Technical Design Specification: Liquid.NET

Part I: Architectural Vision and Core Principles

1.1. Introduction to Liquid Time-Constant Networks in C#

This document provides a comprehensive technical design specification for Liquid.NET, a high-performance C# library for the implementation of Liquid Neural Networks (LNNs). The primary objective of this library is to provide a robust, efficient, and optimizable implementation of Liquid Time-Constant (LTC) models and their associated Neural Circuit Policies (NCPs) within the.NET ecosystem. This specification serves as a definitive blueprint for developing a C# alternative to existing Python-based solutions, with a central focus on identifying and exploiting opportunities for algorithmic and implementation-level optimization.
The core problem addressed by LNNs is the inherent limitation of traditional Recurrent Neural Networks (RNNs) in processing continuous-time signals and their often-opaque internal dynamics. Standard RNN architectures operate on discrete time steps, which can be an unnatural fit for data streams where information arrives at irregular intervals or where the underlying dynamics are continuous. LTCs represent a groundbreaking advancement by modeling neuronal activity as a system of Ordinary Differential Equations (ODEs).[1] This approach is inspired by the neural dynamics observed in biological systems, such as the microscopic roundworm
C. elegans.[2] These models are designed to adapt their internal parameters—specifically their "time constants"—dynamically in response to input, allowing them to fluidly adjust their sensitivity to incoming data.[4] This "liquid" nature makes them exceptionally well-suited for tasks involving time-variable data, such as autonomous driving, financial market analysis, and weather forecasting, where they have demonstrated superior robustness and performance compared to static neural network architectures.[2]
The development of Liquid.NET presents a significant opportunity to leverage the unique performance characteristics of the C# language and the.NET runtime. While the reference implementations are built on PyTorch, a C# version can achieve comparable or even superior performance in specific computational kernels by capitalizing on features unavailable in an interpreted language like Python. The foundational technology for Liquid.NET will be TorchSharp, a.NET library that provides direct, low-level bindings to the native LibTorch C++

library that powers PyTorch.[6] This strategic choice ensures that the core tensor computations—the mathematical bedrock of the network—are executed by the same battle-tested, GPU-accelerated engine used in the official implementation, allowing development to focus on higher-level algorithmic optimizations.

1.2. Design Philosophy

The architecture of Liquid.NET is guided by three fundamental principles, designed to ensure the library is powerful, usable, and future-proof.

Performance-First Architecture

Every architectural and implementation decision will be evaluated through the lens of performance. The central objective is to create a library that not only matches the performance of the Python-based ncps library but is architected to exceed it in computationally intensive areas, most notably the numerical ODE solver. This involves a hybrid strategy: leveraging the pre-optimized native backend of LibTorch for general tensor mathematics while applying targeted, low-level C# optimizations to the explicit computational loops that define the LTC dynamics. This philosophy dictates a careful balance between high-level abstraction and low-level control, ensuring that performance-critical code paths can be finely tuned.

API Fidelity and Idiomatic C#

To ensure a smooth transition for developers familiar with the existing PyTorch ecosystem, the public-facing API of Liquid.NET will closely mirror the structure and naming conventions of the torch-neural-circuit-policies (ncps) library.[8] This fidelity will reduce the learning curve and allow for easier porting of existing models and training scripts. However, this external similarity will not compromise the internal implementation, which will adhere strictly to C# and.NET best practices. The library will employ strong typing, interface-based design, proper memory management patterns for unmanaged resources, and asynchronous programming where applicable, resulting in a codebase that is both familiar to ML practitioners and natural for.NET developers.

Modularity and Extensibility

The library will be designed as a collection of loosely coupled, interchangeable components. Core functionalities such as neuron models, wiring architectures, and ODE solvers will be abstracted behind interfaces. This modular design facilitates extensibility and

experimentation. For example, developers will be able to implement and "plug in" novel ODE solvers to test their performance and accuracy trade-offs. This architecture also paves the way for future expansion, such as the integration of Closed-form Continuous-time (CfC) models—an evolution of LTCs that approximates the ODE solution for faster inference—without requiring a fundamental rewrite of the library's core structure.[10]

1.3. High-Level System Diagram

The architecture of Liquid.NET is organized into four distinct layers, each with a specific responsibility. This layered approach promotes separation of concerns and modularity.

1. **Public API Layer:** This is the outermost layer that users of the library will interact with. It contains the high-level classes that encapsulate the complexity of the underlying models, such as LTC for creating a recurrent sequence model and AutoNCP for generating a neural circuit wiring. This layer is designed for ease of use and integration into larger.NET applications.
2. **Core Model Layer:** This layer contains the internal C# implementation of the neural network components. It includes the LTCCell, which defines the logic for a single recurrent time step, and the LTC class, which orchestrates the processing of entire sequences by repeatedly invoking the cell. This layer is responsible for managing the model's state, parameters, and the forward pass logic.
3. **Numerical Engine Layer:** This layer provides the core numerical capabilities. It is composed of two primary sub-components: the tensor library (TorchSharp) and the swappable ODE solver module. The IOdeSolver interface defines a contract that allows different numerical integration techniques (e.g., Forward Euler, Runge-Kutta, or custom SIMD-accelerated solvers) to be used interchangeably by the LTCCell.
4. **Native Backend:** This is the lowest level of the stack, consisting of the pre-compiled, high-performance LibTorch C++/CUDA library. All tensor operations initiated from the upper layers via TorchSharp are ultimately executed by this native backend, enabling GPU acceleration and leveraging years of optimization from the PyTorch development community.[12]

The strategic selection of TorchSharp as the interface to the native backend is a cornerstone of this design. It provides a direct pathway to the same high-performance computational primitives used by the reference PyTorch implementation.[6] This decision effectively means that the baseline performance for fundamental operations like matrix multiplication and convolutions is already on par with the state of the art. Consequently, the primary opportunity for optimization within the C# implementation does not lie in rewriting these fundamental algorithms. Instead, the most significant performance gains can be realized by optimizing the C# code that
orchestrates these native calls. The most fertile ground for this optimization is the explicit, iterative loop within the numerical ODE solver. While Python's interpreted nature introduces overhead in such loops, a compiled C# implementation can apply low-level techniques, such

as Single Instruction, Multiple Data (SIMD) vectorization, to execute the solver's element-wise operations with far greater efficiency. This hybrid approach—relying on the proven power of LibTorch for heavy lifting and applying targeted C# optimizations to the orchestration logic—is the most direct path to achieving the library's performance goals.

Part II: Mathematical Foundations and Algorithmic Translation

This section deconstructs the mathematical principles that govern Liquid Time-Constant networks and translates them into a clear algorithmic framework. This theoretical foundation is essential for ensuring a correct and robust C# implementation.

2.1. The Liquid Time-Constant (LTC) ODE System

The dynamics of an LTC network are defined by a system of first-order nonlinear Ordinary Differential Equations. The rate of change of the hidden state for each neuron is described by the following governing equation, as presented in the foundational paper [1]:

$$\frac{dx(t)}{dt} = -[\tau^{-1} + f(x(t), I(t), t, \theta)]x(t) + f(x(t), I(t), t, \theta)A$$

Each component of this equation has a distinct role in shaping the network's behavior:

- $x(t) \in \mathbb{R}^N$: Represents the hidden state vector of the N neurons in the network at time t.
- $I(t) \in \mathbb{R}^M$: Represents the external input vector from M features at time t.
- $\tau \in \mathbb{R}^N$: A vector of trainable base time-constants, one for each neuron. This parameter establishes a default rate at which a neuron's state decays toward equilibrium.
- $A \in \mathbb{R}^N$: A vector of trainable bias parameters, representing the target state toward which the neuron's activity "leaks."
- $f(x(t), I(t), t, \theta)$: A nonlinear function, typically parameterized by a small neural network (e.g., a multi-layer perceptron or MLP) with weights θ. This function is the core of the "liquid" mechanism. It takes the current hidden state x(t) and input I(t) and outputs a modulation signal.

The term "liquid" refers to the network's ability to dynamically alter its effective time-constant. By rearranging the governing equation, we can see that the system behaves like a leaky integrator where the leakage rate is not fixed. The term inside the brackets, $\tau_{eff}^{-1} = [\tau^{-1} + f(\dots)]$, represents the inverse of the effective, state-dependent time-constant. Because the function f depends on the current state and input, the network can instantaneously change how quickly it "forgets" or integrates information. For example, in the presence of a strong, salient input, f might output a large value, causing the neuron's state to change very rapidly. Conversely, for a weak or noisy input, f might output a small value, causing the neuron to respond more slowly and effectively filter out the noise. This adaptive temporal sensitivity is a key advantage of LTCs over traditional RNNs with fixed dynamics.[2]

2.2. The Forward Pass as Numerical Integration

The ODE system defines the instantaneous rate of change, $\frac{dx(t)}{dt}$, of the hidden state. To compute the network's state at a future time, $x(t+\Delta t)$, we must integrate this derivative over the time interval Δt. As an analytical, closed-form solution to this integral is generally intractable, we must rely on numerical integration methods.

The simplest numerical integrator is the **Forward Euler method**. It approximates the next state by taking a linear step in the direction of the current derivative:

1. Calculate the derivative at the current time t: $k = \frac{dx(t)}{dt}$
2. Update the state: $x(t+\Delta t) \approx x(t) + \Delta t \cdot k$

While simple to implement, the Forward Euler method can be numerically unstable, especially for stiff ODEs where the solution changes rapidly. The original LTC paper and the official ncps implementation propose a more robust **semi-implicit solver** that offers a better balance between the stability of implicit methods and the computational efficiency of explicit methods.[1] This fused solver has the following update rule:

$$x(t+\Delta t) = \frac{1 + \Delta t \cdot (\tau_1 + g(t))}{x(t) + \Delta t \cdot S(t)}$$

where $S(t)$ and $g(t)$ are intermediate terms derived from the original ODE's components, computed using the state at time t. This formulation avoids the matrix inversion required by fully implicit methods while providing greater stability than the explicit Euler method.

Furthermore, the ncps library introduces a parameter called ode_unfolds, which specifies the number of iterative solver steps to perform within a single external time step Δt.[10] This can be viewed as a form of iterative refinement. If ode_unfolds=6, the solver will apply its update rule six times, each with a smaller internal time step of $6\Delta t$, to compute the final state $x(t+\Delta t)$. This increases computational cost but can significantly improve the accuracy of the numerical integration. The algorithmic pseudo-code for a single forward step of an LTCCell is therefore:

```
function LTCCell_Forward(input_t, state_t-1):
  internal_state = state_t-1
  internal_dt = Δt / ode_unfolds

  for i from 1 to ode_unfolds:
    // Calculate the derivative (or update terms) based on internal_state and input_t
    derivative = ODE_Function(internal_state, input_t, parameters)

    // Apply one step of the numerical solver
    internal_state = ODESolver_Step(internal_state, derivative, internal_dt)
```

```
return internal_state // This is state_t
```

2.3. Backpropagation Through Time (BPTT) for Continuous Systems

Training an LTC network requires computing the gradient of a loss function with respect to the network's trainable parameters (θ, τ, A). This involves propagating gradients "back in time" through the sequence of operations performed during the forward pass, a process known as Backpropagation Through Time (BPTT). The primary challenge in continuous-time models is that the backpropagation must flow through the numerical integration steps of the ODE solver.

Fortunately, modern automatic differentiation (autograd) frameworks like PyTorch and, by extension, TorchSharp, are designed to handle this challenge seamlessly. As long as every operation within the ODE solver is composed of differentiable tensor operations provided by the framework, the entire solver becomes part of the dynamic computation graph.[14] When the .backward() method is called on the final loss value, the autograd engine automatically applies the chain rule to propagate gradients through each solver step.

Conceptually, the gradient flow through a single Forward Euler step can be understood as follows. The update is $x(t+\Delta t) = x(t) + \Delta t \cdot k(x(t), I(t))$. The gradient of the loss with respect to the state at time t is given by the chain rule:

$$\frac{\partial Loss}{\partial x(t)} = \frac{\partial x(t+\Delta t)}{\partial x(t)} \frac{\partial Loss}{\partial x(t+\Delta t)}$$

The Jacobian term $\frac{\partial x(t)}{\partial x(t+\Delta t)}$ can be computed directly from the update rule. The autograd engine performs this calculation automatically for each step in the unfolded solver and for each time step in the sequence, correctly accumulating the gradients for all parameters. This means that from an implementation perspective, as long as the forward pass is constructed correctly using the framework's tools, the backward pass for training is handled automatically.

Part III: C# Library Architecture and Core Components

This section details the concrete software architecture for the Liquid.NET library, specifying the namespaces, class structures, interfaces, and their interrelationships. This blueprint provides a clear and organized structure for the development process.

3.1. Namespace and Project Structure

To ensure a logical and maintainable codebase, the Liquid.NET library will be organized into a hierarchical namespace and corresponding directory structure. This separation of concerns makes the library easier to navigate, test, and extend.

```
Liquid.NET/
├── Liquid.NET.csproj
├── Core/
│   ├── LTCCell.cs
│   ├── LTC.cs
│   └── CfCCell.cs      // Stub for future implementation
├── Wirings/
│   ├── IWiring.cs
│   ├── AutoNCP.cs
│   ├── FullyConnected.cs
│   └── Random.cs
└── Solvers/
    ├── IOdeSolver.cs
    ├── ForwardEulerSolver.cs
    └── SimdEulerSolver.cs  // High-performance implementation
```

- **Liquid.NET.Core**: Contains the primary neural network modules, such as LTCCell (the single-step RNN cell) and LTC (the full recurrent layer for processing sequences).
- **Liquid.NET.Wirings**: Contains classes and interfaces related to defining the neural connectivity. This includes the IWiring interface and concrete implementations like AutoNCP.
- **Liquid.NET.Solvers**: Contains the numerical integration modules. The IOdeSolver interface defines the contract, with concrete implementations providing different trade-offs between performance and accuracy.

3.2. The Wiring Abstraction

The wiring defines the sparsity and structure of the connections between neurons. This is a key feature of Neural Circuit Policies, which use a specific, bio-inspired sparse structure.

IWiring Interface

This interface provides a standardized contract for all wiring architectures, allowing them to be used interchangeably by the LTCCell.

C#

```
namespace Liquid.NET.Wirings
```

```
{
    public interface IWiring
    {
        int TotalUnits { get; }
        int OutputSize { get; }
        int SensorySize { get; }
        int MotorSize { get; }

        // Returns the adjacency matrix defining synaptic connections.
        torch.Tensor GetAdjacencyMatrix();

        // Returns a mask for sensory-to-neuron connections.
        torch.Tensor GetSensoryAdjacencyMatrix();
    }
}
```

AutoNCP Class

This class implements the procedural generation of the sparse wiring diagram inspired by the nervous system of *C. elegans*, as described in the NCP literature.[16]

- **Constructor**: public AutoNCP(int totalUnits, int outputSize)
- **Algorithm**: The constructor will programmatically generate the connectivity graph by:
    1. **Allocating Neurons**: Partitioning the totalUnits into four distinct layers: Sensory, Inter, Command, and Motor neurons based on the principles outlined in the research.[18]
    2. **Connecting Layers**: Creating the synaptic connections (i.e., the adjacency matrix) according to the specified rules: feed-forward connections from Sensory to Inter neurons, from Inter to Command neurons, and from Command to Motor neurons.
    3. **Adding Recurrence**: Implementing recurrent synapses within the Command neuron layer, which is critical for the network's temporal dynamics.
    4. **Enforcing Constraints**: Adhering to fan-in and fan-out constraints to maintain the desired sparsity and biological plausibility.[16]

FullyConnected Class

This class provides a simple, non-sparse implementation of IWiring. It generates a dense adjacency matrix where every neuron is connected to every other neuron. This is useful for creating baseline models to compare against the performance of the structured NCP wiring

and for replicating standard RNN architectures.

3.3. The LTCCell Module

The LTCCell is the heart of the LTC model. It is a custom torch.nn.Module that implements the logic for a single forward pass step, advancing the hidden state by Δt.

C#

```
namespace Liquid.NET.Core
{
   public class LTCCell : torch.nn.Module<torch.Tensor, torch.Tensor, torch.Tensor>
   {
      private readonly IWiring _wiring;
      private readonly IOdeSolver _solver;

      // Learnable parameters
      private readonly torch.nn.Module<torch.Tensor, torch.Tensor> _fusedGate;
      private readonly torch.nn.Parameter _timeConstant;
      private readonly torch.nn.Parameter _bias;
      //... other parameters

      public LTCCell(int inputSize, IWiring wiring, IOdeSolver solver)
         : base(nameof(LTCCell))
      {
         _wiring = wiring;
         _solver = solver;

         // Initialize all learnable torch.nn.Parameter instances here.
         // This includes the weights for the internal MLP, time-constants, and biases.
         // Registering them ensures they are tracked by the optimizer.
         RegisterComponents();
      }

      public override torch.Tensor forward(torch.Tensor input, torch.Tensor hiddenState)
      {
         // The forward pass delegates the core computation to the injected solver.
         return _solver.Solve(this, input, hiddenState);
      }
   }
}
```

- **Parameters**: All learnable weights and biases, such as those for the internal MLP (f), the time-constants (τ), and the leaky-integration biases (A), will be declared as torch.nn.Parameter. This automatically registers them with TorchSharp's autograd engine, making them trainable.
- **forward Method**: The forward method encapsulates the logic for a single time step. Crucially, it does not implement the ODE integration itself. Instead, it delegates this complex task to the IOdeSolver instance that was injected via the constructor. This adheres to the Strategy design pattern, allowing the numerical integration logic to be swapped out independently of the cell's core structure.

3.4. The LTC Recurrent Layer

The LTC class acts as a wrapper around the LTCCell, providing an interface consistent with standard recurrent layers like LSTM or GRU in PyTorch. It processes an entire sequence of inputs over multiple time steps.

C#

```
namespace Liquid.NET.Core
{
   public class LTC : torch.nn.Module<torch.Tensor, torch.Tensor, (torch.Tensor, torch.Tensor)>
   {
      private readonly LTCCell _cell;
      private readonly bool _batchFirst;

      public LTC(int inputSize, IWiring wiring, IOdeSolver solver, bool batchFirst = true)
         : base(nameof(LTC))
      {
         _cell = new LTCCell(inputSize, wiring, solver);
         _batchFirst = batchFirst;
         RegisterComponents();
      }

      public override (torch.Tensor, torch.Tensor) forward(torch.Tensor sequence, torch.Tensor
initialHiddenState = null)
      {
         // 1. Handle input shape based on _batchFirst flag.
         // 2. Initialize hidden state if not provided.
         // 3. Loop over the time dimension of the 'sequence' tensor.
         // 4. In each iteration, call _cell.forward() with the current time-slice
```

```
            //    of the input and the hidden state from the previous step.
            // 5. Collect the output hidden state from each step.
            // 6. Stack the collected outputs into a single tensor.
            // 7. Return the full output sequence and the final hidden state.
        }
    }
}
```

- **forward Method**: This method contains the primary loop that iterates through the time dimension of the input sequence. At each step, it invokes the internal _cell.forward() method to update the hidden state. This design effectively unrolls the recurrence in time, building up a computation graph that autograd can trace for backpropagation. Its return signature, (torch.Tensor, torch.Tensor), which represents the output sequence and the final hidden state, directly mirrors the behavior of PyTorch's built-in RNN layers, ensuring API consistency.[10]

3.5. Table 1: PyTorch ncps to C# Liquid.NET Class Mapping

To facilitate development and provide a clear reference for those familiar with the original Python library, the following table maps the key components of ncps to their proposed equivalents in Liquid.NET. This mapping is a cornerstone of the API fidelity principle, ensuring that the conceptual structure of the library remains consistent across language ecosystems. Adhering to this mapping will significantly reduce the cognitive overhead for developers tasked with porting or referencing the original implementation, thereby accelerating the development process and minimizing the risk of architectural divergence.

PyTorch (ncps) Component	C# (Liquid.NET) Equivalent	Notes
ncps.torch.LTC	Liquid.NET.Core.LTC	The main recurrent layer responsible for processing input sequences.
ncps.torch.LTCCell	Liquid.NET.Core.LTCCell	The single-step RNN cell, containing the core ODE logic. Used internally by LTC.
ncps.wirings.AutoNCP	Liquid.NET.Wirings.AutoNCP	Procedural generator for creating the sparse, bio-inspired NCP wiring structure.
ncps.wirings.FullyConnected	Liquid.NET.Wirings.FullyConnected	A wiring generator for creating dense, fully-connected models for baseline comparisons.
ncps.wirings.Random	Liquid.NET.Wirings.Random	A wiring generator for creating randomly sparse networks.

Part IV: Numerical Engine and Performance Optimization Strategies

This section addresses the user's primary objective: creating a library designed for optimization. It outlines the strategies for building a high-performance numerical backend, focusing on leveraging the strengths of both the native LibTorch engine and the.NET runtime.

4.1. Foundational Tensor Operations with TorchSharp

A foundational mandate for the Liquid.NET library is that all standard tensor operations must be executed via the TorchSharp library. This includes, but is not limited to, matrix multiplications, element-wise arithmetic (addition, multiplication), activation functions, and tensor manipulations (reshaping, concatenation).
The rationale for this mandate is rooted in performance. TorchSharp is a thin wrapper that delegates these operations directly to the underlying LibTorch native library.[6] This C++/CUDA backend has been heavily optimized over many years by a global community and is the same engine that powers PyTorch. Attempting to re-implement these fundamental numerical linear algebra routines in "pure" C# would be a monumental effort that is highly unlikely to match the performance of
LibTorch, especially on GPU hardware. Benchmarks consistently show that for numerical computing, native libraries like Intel MKL (used by LibTorch) or custom CUDA kernels dramatically outperform managed code.[19] By relying on
TorchSharp, Liquid.NET inherits this state-of-the-art performance for its core mathematical building blocks, allowing optimization efforts to be focused on areas where C# can provide a distinct advantage.

4.2. Custom ODE Solver Implementation

The numerical integration of the LTC's ODE system is the most computationally intensive part of the forward pass and presents the greatest opportunity for C#-specific optimization. The design will use an interface-based approach to allow for different solver implementations.

IOdeSolver Interface

This interface defines a simple contract for all ODE solvers, ensuring they can be seamlessly

integrated into the LTCCell.

C#

```
namespace Liquid.NET.Solvers
{
   public interface IOdeSolver
   {
      // Solves the ODE for one time step.
      torch.Tensor Solve(LTCCell cell, torch.Tensor input, torch.Tensor hiddenState);
   }
}
```

ForwardEulerSolver

This class will serve as the baseline solver. It will implement the Forward Euler method using standard, sequential TorchSharp tensor operations. Its purpose is to provide a simple, easy-to-understand reference implementation that is guaranteed to be correct and fully differentiable. It will be the default solver used for debugging and validation.

SimdEulerSolver (Primary Optimization Target)

This class is the centerpiece of the library's performance strategy. It will implement a highly optimized version of the ODE solver that leverages.NET's support for SIMD (Single Instruction, Multiple Data) intrinsics.
The core logic behind this optimization is that the ODE update rule consists of numerous element-wise operations (additions, multiplications, etc.) performed on the hidden state vector. A naive implementation using TorchSharp would involve multiple separate calls into the native LibTorch library for each operation (e.g., one call for multiplication, one for addition). Each of these calls incurs a small but non-trivial overhead from the P/Invoke transition between managed C# code and native C++ code.
The SimdEulerSolver will minimize this overhead by fusing these operations in C#. It will operate directly on the memory underlying the TorchSharp tensors. The process is as follows:

1. **Obtain Memory Pointers**: Use the Tensor.data_ptr() method to get the raw memory addresses of the input and state tensors.
2. **Vectorized Processing**: Within an unsafe C# code block, iterate through the tensor memory in chunks sized to match the hardware's SIMD register width (e.g., Vector<float>.Count, which might be 4, 8, or more elements).[21]
3. **Fused Operations**: For each chunk, load the data into System.Numerics.Vector<T>

registers. Perform the entire sequence of element-wise calculations for the ODE update step using overloaded operators on these vector types. A single Vector<float> addition instruction, for example, performs multiple floating-point additions in parallel.[23]

4. **Write Back Results**: Store the resulting vector back into the memory of the output tensor.
5. **Scalar Fallback**: A final, simple loop will process any remaining elements for tensors whose size is not a multiple of the vector width.

This approach dramatically reduces the number of transitions between managed and native code and maximizes the utilization of modern CPU hardware, offering the potential for a significant speedup in the ODE solver kernel compared to a more naive implementation.[25]

4.3. Memory Management and Tensor Lifetime

A critical consideration when working with TorchSharp is the management of memory. TorchSharp.Tensor objects are managed C# wrappers around blocks of unmanaged, native memory allocated by the LibTorch library. The.NET Garbage Collector (GC) is responsible for managed memory but has no awareness of this underlying native allocation. Consequently, if a Tensor object is simply allowed to go out of scope, the C# wrapper will be collected, but the native memory it points to will be leaked.

To prevent such memory leaks, which would be catastrophic in a training loop, Liquid.NET will enforce a strict memory management policy. The required pattern for any method that creates intermediate tensors (such as the forward pass of any module) is the use of a torch.NewDisposeScope().

C#

```
public override torch.Tensor forward(torch.Tensor input)
{
    using var scope = torch.NewDisposeScope();

    // All intermediate tensors are created within the scope.
    var temp1 = torch.matmul(input, _weights);
    var temp2 = torch.add(temp1, _bias);
    var result = torch.relu(temp2);

    // The final result is explicitly moved out of the scope's control.
    // All other tensors (temp1, temp2) will be disposed automatically
    // when the 'using' block exits.
    return result.MoveToOuterDisposeScope();
}
```

This pattern ensures that all temporary tensors created during a computation are tracked and their corresponding native memory is reliably deallocated. The final result tensor is safely passed to the caller by moving it to the outer scope. Adherence to this pattern is mandatory throughout the library to guarantee stability and prevent memory leaks.

4.4. Benchmarking and Profiling Hooks

To validate the effectiveness of the optimization strategies and to guide future performance tuning, the library's development process will include rigorous benchmarking. The BenchmarkDotNet library, a powerful tool for performance measurement in.NET, will be integrated into the project's test suite.[21]
Specific benchmarks will be designed to measure:

- **Solver Performance**: A micro-benchmark directly comparing the execution time of the ForwardEulerSolver against the SimdEulerSolver on tensors of various sizes.
- **Full Forward Pass**: A benchmark measuring the total time for a forward pass of the LTC layer on a sequence of a given length, allowing for an end-to-end performance evaluation.
- **Cross-Language Comparison**: A benchmark designed to compare the performance of Liquid.NET against the original ncps Python library. This will require careful setup to ensure an apples-to-apples comparison (e.g., same hardware, same model configuration, CPU-only execution to eliminate GPU driver variance).

These benchmarks will provide quantitative data to confirm the benefits of the SIMD-based optimizations and to identify any performance regressions during development.

Part V: Automatic Differentiation and Backpropagation Engine

This section specifies how the Liquid.NET library will handle the crucial process of model training, specifically the computation of gradients via backpropagation. The design prioritizes reliability and correctness by deeply integrating with the underlying framework's capabilities.

5.1. Leveraging TorchSharp's Autograd

The core principle of the training architecture is that Liquid.NET will **not** implement its own backpropagation algorithm. Instead, it will rely entirely on the powerful and highly optimized automatic differentiation engine, autograd, that is built into LibTorch and exposed through TorchSharp.[14]

The autograd engine works by building a computational graph during the forward pass. Every time an operation is performed on a tensor that has its requires_grad property set to true, the engine records that operation and its inputs in a directed acyclic graph (DAG).[15] The leaves of this graph are the model's parameters, and the root is typically the final scalar loss value. When the
.backward() method is called on the loss tensor, autograd traverses this graph in reverse, applying the chain rule at each step to compute the gradient of the loss with respect to every parameter. These gradients are then accumulated in the .grad attribute of each parameter tensor, ready to be used by an optimizer to update the model's weights. By leveraging this built-in system, Liquid.NET avoids the immense complexity and potential for error of a manual gradient implementation and ensures that the gradient calculations are as efficient and numerically stable as those in PyTorch itself.

5.2. Ensuring Differentiability in Custom Code

For the autograd engine to function correctly, every computational step in the forward pass that involves a trainable parameter must be a differentiable operation known to the framework. This principle has a critical implication for the design of custom components, particularly the high-performance SimdEulerSolver.
A naive implementation of the SimdEulerSolver might use unsafe C# to read raw floating-point values from tensor memory, perform calculations using System.Numerics.Vector<T>, and write the results back. While this would be extremely fast for the forward pass, it would be invisible to the autograd engine. The computation graph would be broken at the point where the solver is called, as autograd cannot trace operations happening on raw memory pointers outside of its control. Consequently, calling .backward() would fail, as there would be no path for the gradients to flow back through the solver to the model's parameters.
Therefore, to prioritize correctness and ensure trainability, the initial implementation of all ODE solvers, including SimdEulerSolver, must adhere to a "Golden Rule": **all operations that transform tensor data must be performed using differentiable functions provided by the TorchSharp.torch API.**
This means that the first version of SimdEulerSolver will not operate on raw memory. Instead, it will act as a "smart orchestrator" of TorchSharp operations. It will still loop through the data in chunks, but for each chunk, it will create a small tensor slice and apply the sequence of required torch.mul, torch.add, etc., operations to it. While this introduces some overhead compared to raw SIMD, it keeps the entire computation within the autograd ecosystem, guaranteeing that the model remains differentiable. This design ensures correctness first, providing a solid foundation upon which more advanced, but more complex, optimizations can be built in the future.

5.3. Custom Backpropagation (Advanced/Prospective)

While the initial design relies exclusively on the built-in autograd engine, this specification acknowledges a future path for achieving maximum performance: implementing a custom autograd function. This advanced technique would allow the use of highly optimized, non-differentiable code (like the unsafe SIMD implementation) in the forward pass by manually providing the corresponding backward pass logic.
This would involve defining a C# class that conceptually inherits from torch.autograd.Function. This class would require the implementation of two static methods:

1. **forward()**: This method would contain the performance-critical code. Here, the unsafe SimdEulerSolver that operates on raw memory pointers would be used, providing the fastest possible forward computation.
2. **backward()**: This method would receive the incoming gradient from the next layer in the network (the gradient of the loss with respect to the solver's output). It would be responsible for manually implementing the chain rule to compute the gradient of the loss with respect to the solver's *inputs* (the previous hidden state, the current input, and the model parameters). This requires a rigorous mathematical derivation of the gradients for the ODE solver step.

Implementing a custom autograd.Function is a complex and error-prone task that requires a deep understanding of both the model's mathematics and the autograd framework's internals.[29] However, it is the only way to combine the absolute peak forward-pass performance of
unsafe SIMD code with the ability to train the network. This approach is therefore designated as a prospective, high-impact optimization for a future version of the Liquid.NET library, to be undertaken after the core, fully-differentiable version has been thoroughly tested and validated.

Part VI: Public API Design and Usage Patterns

This final section provides concrete C# code examples to illustrate how a developer would use the Liquid.NET library. These patterns are designed to be intuitive for.NET developers while remaining familiar to those with experience in PyTorch.

6.1. Model Instantiation

Creating a Liquid Time-Constant model is a straightforward process involving the selection of a wiring architecture and the instantiation of the LTC class. The API is designed to be flexible, allowing for the easy creation of both simple, fully-connected models and complex, bio-inspired Neural Circuit Policies.

C#

```
using Liquid.NET.Core;
using Liquid.NET.Wirings;
using Liquid.NET.Solvers;
using static TorchSharp.torch;

// Example 1: Create a small, fully-connected LTC model with 32 hidden units
// and 8 output units. This is analogous to a standard RNN.
var fcWiring = new FullyConnected(units: 32, outputSize: 8);
var solver = new ForwardEulerSolver(); // Use the standard, reliable solver.
var fcModel = new LTC(inputSize: 10, wiring: fcWiring, solver: solver);

Console.WriteLine("Fully-Connected LTC Model Created:");
fcModel.print();

// Example 2: Create a larger, sparsely-wired NCP model with 64 total neurons,
// 4 of which are designated as motor (output) neurons.
var ncpWiring = new AutoNCP(totalUnits: 64, outputSize: 4);
var ncpModel = new LTC(inputSize: 20, wiring: ncpWiring, solver: solver);

Console.WriteLine("\nNeural Circuit Policy (NCP) LTC Model Created:");
ncpModel.print();
```

6.2. The Training Loop

The training process for a Liquid.NET model follows the standard pattern used in modern deep learning frameworks. It involves iterating through a dataset, performing a forward pass, calculating a loss, performing a backward pass to compute gradients, and updating the model's parameters with an optimizer. The following example demonstrates a complete, self-contained training loop.[31]

C#

```
// Assume 'ncpModel' has been created as in the previous example.
// Assume 'inputSequence' and 'targetSequence' are torch.Tensor objects with training data.
// Shape for inputSequence: (batch_size, sequence_length, input_size)
// Shape for targetSequence: (batch_size, sequence_length, output_size)
```

```
var learningRate = 0.001;
var numEpochs = 100;

// Use a standard optimizer from TorchSharp, passing the model's parameters.
var optimizer = optim.Adam(ncpModel.parameters(), lr: learningRate);
var lossFunc = nn.MSELoss(); // Mean Squared Error loss for a regression task.

Console.WriteLine("\nStarting training loop...");
for (int epoch = 0; epoch < numEpochs; epoch++)
{
    // Clear previously computed gradients before the new pass.
    optimizer.zero_grad();

    using var disposeScope = NewDisposeScope();

    // --- Forward Pass ---
    // Pass the input sequence through the model to get the predictions.
    var (prediction, _) = ncpModel.forward(inputSequence);

    // --- Loss Calculation ---
    // Compare the model's prediction with the ground truth target.
    var loss = lossFunc.call(prediction, targetSequence);

    // --- Backward Pass ---
    // Compute gradients of the loss with respect to all model parameters.
    loss.backward();

    // --- Parameter Update ---
    // Adjust the model's weights based on the computed gradients.
    optimizer.step();

    if ((epoch + 1) % 10 == 0)
    {
        Console.WriteLine($"Epoch [{epoch + 1}/{numEpochs}], Loss: {loss.item<float>():F4}");
    }
}
Console.WriteLine("Training finished.");
```

6.3. Inference and Prediction

Once a model has been trained, it can be used to make predictions on new, unseen data. The

inference process is essentially a forward pass without the subsequent backpropagation and optimization steps. For efficiency and correctness, it is crucial to set the model to evaluation mode and to disable gradient tracking.

- model.eval(): This method puts the model in evaluation mode. This is important for layers like Dropout or BatchNorm, which behave differently during training and inference. While the baseline LTC model does not have these layers, it is a critical best practice for building extensible and robust models.
- torch.no_grad(): This context manager informs the autograd engine that it does not need to build a computational graph for the operations within the block. This significantly reduces memory consumption and speeds up computation, as the overhead of tracking operations is eliminated.

C#

```
// Assume 'trainedModel' is an instance of LTC that has been trained.
// Assume 'newData' is a torch.Tensor with new input data.

// Set the model to evaluation mode.
trainedModel.eval();

// Use the 'no_grad' context to disable gradient calculations for performance.
using (no_grad())
{
    using var disposeScope = NewDisposeScope();

    // Perform the forward pass to get the prediction.
    var (prediction, finalState) = trainedModel.forward(newData);

    // Move the result out of the dispose scope to use it.
    var predictionData = prediction.MoveToOuterDisposeScope();

    // The 'predictionData' tensor can now be used.
    // For example, convert it to a C# array for further processing.
    float[,,] predictionArray = predictionData.to_cpu().data<float>().ToArray();

    Console.WriteLine("\nInference complete.");
    Console.WriteLine($"Prediction tensor shape:");
}
```

Works cited

1. Liquid Time-constant Networks, accessed September 8, 2025, https://ojs.aaai.org/index.php/AAAI/article/view/16936/16743
2. Liquid Neural Nets (LNNs) - Medium, accessed September 8, 2025, https://medium.com/@hession520/liquid-neural-nets-lnns-32ce1bfb045a
3. [PDF] Neural circuit policies enabling auditable autonomy - Semantic Scholar, accessed September 8, 2025, https://www.semanticscholar.org/paper/Neural-circuit-policies-enabling-auditable-autonomy-Lechner-Hasani/cebc1e51eb6c17a9bd64353fd59d815fbfa9ff7f
4. Exploring Liquid Time-Constant Networks: A Breakthrough in AI Technology, accessed September 8, 2025, https://blog.dragonscale.ai/liquid-time-constant-networks/
5. Neural circuit policies enabling auditable autonomy | Request PDF - ResearchGate, accessed September 8, 2025, https://www.researchgate.net/publication/344683434_Neural_circuit_policies_enabling_auditable_autonomy
6. dotnet/TorchSharp: A .NET library that provides access to the library that powers PyTorch. - GitHub, accessed September 8, 2025, https://github.com/dotnet/TorchSharp
7. I Built Faster Reinforcement Learning in C# Solo Than Teams Did with Python - Reddit, accessed September 8, 2025, https://www.reddit.com/r/dotnet/comments/1jm7lov/i_built_faster_reinforcement_learning_in_c_solo/
8. Neural Circuit Policies 0.0.1 documentation, accessed September 8, 2025, https://ncps.readthedocs.io/
9. mlech26l/ncps: PyTorch and TensorFlow implementation of NCP, LTC, and CfC wired neural models - GitHub, accessed September 8, 2025, https://github.com/mlech26l/ncps
10. PyTorch (nn.modules) - Neural Circuit Policies 0.0.1 documentation, accessed September 8, 2025, https://ncps.readthedocs.io/en/latest/api/torch.html
11. raminmh/CfC: Closed-form Continuous-time Neural Networks - GitHub, accessed September 8, 2025, https://github.com/raminmh/CfC
12. pytorch/pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration - GitHub, accessed September 8, 2025, https://github.com/pytorch/pytorch
13. raminmh/liquid_time_constant_networks: Code Repository for Liquid Time-Constant Networks (LTCs) - GitHub, accessed September 8, 2025, https://github.com/raminmh/liquid_time_constant_networks
14. Automatic Differentiation with torch.autograd - PyTorch documentation, accessed September 8, 2025, https://docs.pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
15. Understanding Autograd in PyTorch: The Core of Automatic Differentiation - Medium, accessed September 8, 2025, https://medium.com/@sahin.samia/understanding-autograd-in-pytorch-the-core-of-automatic-differentiation-ca9d98da980d
16. Neural Circuit Policies Imposing Visual Perceptual Autonomy - ResearchGate,

accessed September 8, 2025, https://www.researchgate.net/publication/368752388_Neural_Circuit_Policies_Imposing_Visual_Perceptual_Autonomy
17. MIT Open Access Articles Interpretable Autonomous Flight Via Compact Visualizable Neural Circuit Policies, accessed September 8, 2025, https://dspace.mit.edu/bitstream/handle/1721.1/148379/Interpretable_Autonomous_Flight_Via_Compact_Visualizable_Neural_Circuit_Policies_RA_L.pdf?sequence=2&isAllowed=y
18. Neural Circuit Policy : training a autonomous vehicles using models inspired by nervous system. | by Ved Prakash, accessed September 8, 2025, https://ved933409.medium.com/neural-circuit-policy-training-a-autonomous-vehicles-using-models-inspired-by-nervous-system-db79a554ebef
19. Why is matrix multiplication in .NET so slow? - Stack Overflow, accessed September 8, 2025, https://stackoverflow.com/questions/3229442/why-is-matrix-multiplication-in-net-so-slow
20. c# - The speed of .NET in numerical computing - Stack Overflow, accessed September 8, 2025, https://stackoverflow.com/questions/1831353/the-speed-of-net-in-numerical-computing
21. SIMD Accelerated Numeric Types in C# - Code Maze, accessed September 8, 2025, https://code-maze.com/csharp-simd-accelerated-numeric-types/
22. Harnessing the Power of SIMD with System.Numerics.Vectors in .NET | by Nikita N - Medium, accessed September 8, 2025, https://medium.com/@nikitinsn6/harnessing-the-power-of-simd-with-system-numerics-vectors-in-net-cd88cd103f70
23. System.Numeric.Vectors are now accelerated in Mono, accessed September 8, 2025, https://www.mono-project.com/news/2016/12/20/system-numeric-vectors/
24. Understanding Vectors in C# – Coding Bolt, accessed September 8, 2025, https://codingbolt.net/2024/05/25/understanding-vectors-in-c/
25. SIMD Accelerated Numeric Types in C#: Complete Guide 2024 - DEV Community, accessed September 8, 2025, https://dev.to/bytehide/simd-accelerated-numeric-types-in-c-complete-guide-2024-2277
26. Making SIMD Operations in C# Easier | by Christopher Diggins - Medium, accessed September 8, 2025, https://medium.com/@cdiggins/making-simd-operations-in-c-easier-ab1576e336d3
27. A Gentle Introduction to torch.autograd - PyTorch documentation, accessed September 8, 2025, https://docs.pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
28. Breaking Down Backpropagation in PyTorch: A Complete Guide | Medium, accessed September 8, 2025, https://medium.com/@noel.benji/breaking-down-backpropagation-in-pytorch-3762ea107d3a

29. Custom loss function breaking backpropagation in Pytorch - Stack Overflow, accessed September 8, 2025, https://stackoverflow.com/questions/70760976/custom-loss-function-breaking-backpropagation-in-pytorch
30. Backpropagation through non-torch operations in custom layers - PyTorch Forums, accessed September 8, 2025, https://discuss.pytorch.org/t/backpropagation-through-non-torch-operations-in-custom-layers/162935
31. C# Deep Learning Framework TorchSharp: Native Model Training and Image Recognition, accessed September 8, 2025, https://www.whuanle.cn/archives/21760
32. Mastering the Basics of torch.nn: A Comprehensive Guide to PyTorch's Neural Network Module | by Sahin Ahmed, Data Scientist | Medium, accessed September 8, 2025, https://medium.com/@sahin.samia/mastering-the-basics-of-torch-nn-a-comprehensive-guide-to-pytorchs-neural-network-module-9f2d704e8c7f