Neural Implicit Flow: A Comparative Study

Implementation Approaches and Performance Analysis

Christian Beneke

10.02.2025

Outline

Problem Space & Motivation

- High-dimensional spatio-temporal dynamics are computationally challenging
- Traditional methods face limitations:
  - Fixed mesh structures (SVD, CAE)
  - Poor scaling with high-dimensional data
  - Limited ability to capture nonlinear dynamics
- Real-world applications require:
  - Adaptive mesh handling
  - Efficient dimensionality reduction
  - Real-time processing capabilities

Key Challenges in Current Approaches

- Singular Value Decomposition (SVD)
  - Limited to fixed mesh structures
  - Poor scaling with dimensionality
- Convolutional Autoencoders (CAE)
  - Requires uniform grid resolution
  - Struggles with adaptive meshing
- Common Issues
  - High computational overhead
  - Limited expressiveness for complex dynamics
  - Poor generalization to new scenarios

Core Architecture

- Two specialized networks:
  - ShapeNet: Spatial complexity
  - ParameterNet: Temporal evolution
- Key innovations:
  - Mesh-agnostic representation
  - Efficient parameter sharing
  - Adaptive complexity handling

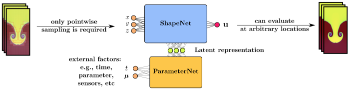

Figure: NIF Architecture

Mathematical Formulation

- Core mapping function:

$$f_\theta : (\mathbf{x}, \mathbf{t}) \mapsto u(\mathbf{x}, \mathbf{t})$$

  where:
  - $\mathbf{x} \in \mathbb{R}^d$: Spatial coordinate
  - $\mathbf{t} \in \mathbb{R}^p$: Temporal/parametric input
  - u: Field value
- Hypernetwork decomposition:

$$f_\theta(\mathbf{x}, \mathbf{t}) = \text{ShapeNet}_{\text{ParameterNet}_{\theta_p}(\mathbf{t})}(\mathbf{x})$$

Implementation Overview

- Three distinct implementations:
  - Upstream Reference (TensorFlow)
  - TensorFlow Functional API
  - PyTorch Implementation
- Key considerations:
  - Framework-specific optimizations
  - Code maintainability
  - Development ergonomics

Upstream Reference Implementation

- Based on original paper implementation
- Key modifications:
  - Migration to modern TensorFlow patterns
  - Improved gradient tape management
  - Enhanced base class structure

TensorFlow Functional API Implementation

- Complete architectural redesign
- Key features:
  - Functional programming principles
  - Improved composability
  - XLA compilation support
- Optimizations:
  - Vectorized weight generation
  - Memory-efficient parameter handling
  - Mixed precision training (FP16/FP32)

PyTorch Implementation

- Modern PyTorch features:
  - Dynamic computation graphs
  - Native autograd functionality
  - Improved debugging capabilities
- Implementation highlights:
  - Flexible activation mapping
  - Efficient static weight layers
  - Comprehensive training logger

Test Cases

Low Frequency Wave

- Domain: $x \in [0, 1]$, $t \in [0, 100]$
- 200 spatial points, 10 timesteps
- Simple periodic traveling wave
- $u(x, t) = \exp(-1000(x - x_0 - ct)^2)$

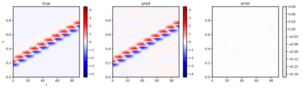

Figure: Low Frequency Example

Test Cases (cont.)

High Frequency Wave

- Same spatial/temporal domain
- Added frequency $\omega = 400$
- Complex oscillatory pattern
- $u(x,t) = \exp(-1000(x - x_0 - ct)^2)\sin(\omega(x - x_0 - ct))$

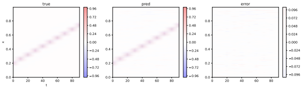

Figure: High Frequency Example

Network Architectures

ParameterNet

- Dense input layer
- Two hidden layers (30 units)
- Single-unit bottleneck
- Adaptive output layer

ShapeNet Variants

- Shortcut (Low frequency)
  - Skip connections
  - Swish activation
- SIREN (High frequency)
  - Sinusoidal activation
  - $\omega_0 = 30$ scaling

Performance Overview

- Low Frequency Results:
  - TF Functional API (Adam): 1.526e-04
  - PyTorch (Adam): 1.549e-04
  - Upstream (AdaBelief): 5.073e-04
- High Frequency Results:
  - PyTorch (Adam): 1.884e-04
  - TF Functional API (Adam): 5.415e-04
  - Upstream: Not available

Processing Speed Comparison

Implementation	Low Freq.	High Freq.
TF Functional	11.5-17K pts/s	11.5-16K pts/s
PyTorch	12-15K pts/s	6-16K pts/s
Upstream	12-17K pts/s	-

Table: Processing Speed (points per second)

Optimizer Impact

- Adam vs AdaBelief comparison:
  - TF Functional API: 62.5% improvement
  - PyTorch: 5.10x improvement (low freq.)
  - PyTorch: 11.77x improvement (high freq.)
- Key findings:
  - Adam: More stable convergence
  - Better final loss values
  - Consistent across implementations

Visual Results - Low Frequency

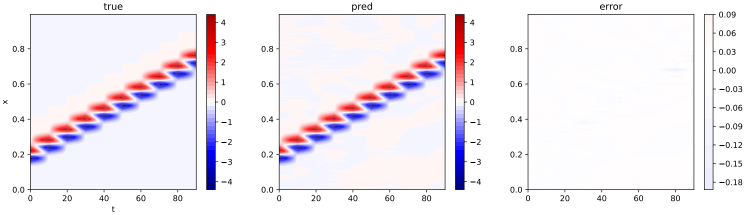

Figure: Low Frequency Predictions (TF Functional API)

Visual Results - High Frequency

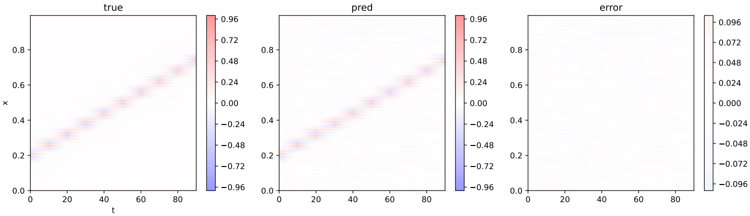

Figure: High Frequency Predictions (TF Functional API)

Key Findings

- Implementation Trade-offs:
  - Modern implementations excel at high-frequency cases
  - TF Functional API: Most consistent performance
  - PyTorch: Best high-frequency accuracy
- Practical Implications:
  - All implementations achieve production-ready speed
  - Adam optimizer consistently outperforms AdaBelief
  - Framework choice impacts development experience

Future Directions

- Technical Improvements:
  - Additional network architectures
  - Memory optimization
  - Computational efficiency
- Application Areas:
  - Fluid dynamics
  - Climate modeling
  - Materials science
- Research Opportunities:
  - Complex spatio-temporal problems
  - Multi-scale phenomena
  - Real-time applications

Thank you for your attention!

Questions?