Neural Implicit Flow: A Comparative Study of Implementation Approaches

Christian Beneke

Abstract

Neural Implicit Flow (NIF) [1] was proposed as a powerful approach for representing continuous functions in various domains, particularly for spatio-temporal data modeled by PDEs. This paper presents a comparative study of three different implementations of NIFs: an upstream reference implementation, a PyTorch-based approach, and a TensorFlow Functional API design. We evaluate these implementations on both simple periodic and complex high-frequency wave functions, analyzing their performance, convergence characteristics, and implementation trade-offs. Our results demonstrate that while all implementations successfully model the target functions, they exhibit different strengths in terms of convergence speed, accuracy, and code maintainability. The TensorFlow Functional API implementation shows superior performance for high-frequency cases, achieving up to 4x better loss values compared to the baseline.

1 Introduction

High-dimensional spatio-temporal dynamics present significant challenges in scientific computing and engineering applications. While these systems can often be encoded in low-dimensional subspaces, existing dimensionality reduction techniques struggle with practical engineering challenges such as variable geometry, non-uniform grid resolution, and adaptive meshing. Neural Implicit Flow (NIF) [1] has emerged as a promising solution to these challenges, offering a mesh-agnostic, low-rank representation framework for large-scale, parametric, spatial-temporal data [2].

1.1 Background and Motivation

Traditional approaches to dimensionality reduction include linear methods like Singular Value Decomposition (SVD) and nonlinear methods such as Convolutional Autoencoders (CAE). However, these methods face several limitations:

- **Mesh Dependency:** SVD and CAE typically require fixed mesh structures, making them unsuitable for adaptive or variable geometry problems
- **Scalability:** Traditional methods often scale poorly with data dimensionality, particularly in 3D applications
- **Expressiveness:** Linear methods struggle to capture complex nonlinear dynamics effectively

NIF addresses these limitations through a novel architecture consisting of two key components [1]:

- **ShapeNet:** A network that isolates and represents spatial complexity
- **ParameterNet:** A network that handles parametric dependencies, time, and sensor measurements

1.2 Neural Implicit Flow Architecture

The core innovation of NIF lies in its ability to decouple spatial complexity from other factors [1]. Given a spatial coordinate $\mathbf{x} \in \mathbb{R}^d$ and temporal/parametric input $\mathbf{t} \in \mathbb{R}^p$, NIF learns the mapping:

$$f_\theta : (\mathbf{x}, \mathbf{t}) \mapsto u(\mathbf{x}, \mathbf{t}) \tag{1}$$

where $u(\mathbf{x}, \mathbf{t})$ represents the field value at the given space-time coordinate, and θ represents the learnable parameters of both networks. The decoupling is achieved through a hypernetwork architecture:

$$f_\theta(\mathbf{x}, \mathbf{t}) = \text{ShapeNet}_{\text{ParameterNet}_{\theta_p}(\mathbf{t})}(\mathbf{x}) \tag{2}$$

Here, $\text{ParameterNet}_{\theta_p}$ with parameters θ_p takes the temporal input $\mathbf{t}$ and generates the weights and biases for ShapeNet. This generated set of parameters allows ShapeNet to adapt its spatial representation based on the temporal context. The complete parameter set $\theta = \{\theta_p\}$ consists only of the ParameterNet parameters, as ShapeNet's parameters are dynamically generated.

This architecture offers several advantages:

- **Mesh Agnostic:** NIF operates directly on point-wise data, eliminating mesh dependencies
- **Scalability:** The framework scales efficiently to high-dimensional problems
- **Expressiveness:** The combination of ShapeNet and ParameterNet enables effective representation of complex dynamics

1.3 Key Applications

NIF demonstrates particular utility in several key areas:

- **Parametric Surrogate Modeling:** Efficient representation of PDE solutions across parameter spaces
- **Multi-scale Data:** Effective handling of problems with multiple spatial and temporal scales
- **Sparse Sensing:** Reconstruction of full fields from limited sensor measurements
- **Modal Analysis:** Extraction of coherent structures from complex flows

1.4 Implementation Approaches

This paper examines three distinct implementation approaches for NIF:

- **Upstream Reference:** A baseline TensorFlow implementation based on the original paper
- **PyTorch Implementation:** Leveraging PyTorch's dynamic computation graphs and autograd functionality
- **TensorFlow Functional:** Utilizing TensorFlow's Functional API for improved composability

Each approach offers different trade-offs in terms of:

- **Performance:** Computational efficiency and memory usage
- **Maintainability:** Code organization and debugging capabilities
- **Flexibility:** Ease of modification and extension
- **Learning Curve:** Accessibility to new users

1.5 Paper Organization

The remainder of this paper is organized as follows:

- Section 2 presents the results from the original NIF study
- Section 3 provides a detailed comparison of the three implementations
- Section 4 discusses practical considerations and trade-offs
- Section 6 concludes with recommendations and future directions

Through this comparative study, we aim to provide practical insights for researchers and practitioners implementing NIF in their own applications, while highlighting the strengths and limitations of each approach.

2 Results from Original NIF Study

2.1 Benchmark Applications and Results

The original NIF framework demonstrated significant advantages across several key applications. In the Parametric Kuramoto-Sivashinsky (K-S) Equation study, NIF achieved 40% better generalization in RMSE compared to standard MLPs, while requiring only half the training data for equivalent accuracy. The framework also demonstrated superior parameter efficiency, achieving 50% lower testing error with the same number of parameters.

For the Rayleigh-Taylor Instability case, NIF outperformed both SVD and CAE in nonlinear dimensionality reduction, achieving 10x to 50% error reduction compared to traditional methods. Notably, it successfully handled adaptive mesh refinement without preprocessing requirements.

In the context of 3D Homogeneous Isotropic Turbulence, the framework successfully compressed 2 million cell turbulent flow data while preserving important statistical properties, including PDFs of velocity and derivatives. This resulted in a 97% reduction in storage requirements while maintaining accuracy.

The framework also showed significant improvements in spatial query efficiency, with a 30% reduction in CPU time and 26% reduction in memory consumption, all while maintaining equivalent accuracy to baseline methods. For Sea Surface Temperature Reconstruction, NIF demonstrated a 34% improvement over POD-QDEIM in sparse sensing applications, with better generalization on a 15-year prediction horizon and successful capture of small-scale temperature variations.

2.2 Technical Innovations

The success of NIF can be attributed to several key technical innovations. The integration of SIREN [3] brought three crucial elements: the use of ω_0-scaled sine activation functions, a special initialization scheme for weights and biases, and ResNet-like skip connections for improved training stability.

The hypernetwork structure [4] provides efficient decoupling of spatial and temporal/parametric complexity through a linear bottleneck layer for dimensionality reduction and adaptive parameter generation for ShapeNet. This architecture enables effective handling of complex spatio-temporal relationships while maintaining computational efficiency.

Training optimizations played a crucial role in achieving stable and efficient convergence:

- Small learning rates (10^{-4} to 10^{-5}) ensure stable gradient updates
- Large batch sizes promote training stability
- L4 optimizer adaptation for small-batch scenarios

2.3 Computational Requirements

The original implementation demonstrated practical computational demands across various hardware configurations. The framework successfully ran on different GPU configurations (P100, RTX 2080, A6000) and scaled effectively with multi-GPU setups. Memory requirements ranged from 12GB to 48GB depending on problem size.

Training characteristics showed consistent patterns across different applications. Convergence was typically achieved within 800-10,000 epochs, with batch sizes optimized for available GPU memory. The framework demonstrated progressive learning of scales, effectively capturing features from large to small structures.

3 Implementation Approaches

We developed three distinct implementations of Neural Implicit Flow [1], each with its own architectural considerations and technical challenges. This section details the implementation specifics of each approach.

3.1 Upstream Reference Implementation

The reference implementation from the original paper [1] served as our baseline but required significant modifications to work with modern frameworks:

- **TensorFlow Migration:** The original codebase used TensorFlow 1.x patterns, necessitating extensive updates:
  - Conversion from static computational graphs to eager execution
  - Replacement of `tf.get_variable` calls with Keras layer initialization
  - Implementation of proper gradient tape management
- **Code Organization:** We improved the structure by:
  - Extracting common functionality into a base class
  - Standardizing the training loop with `@tf.function` decorators

3.2 TensorFlow Functional API Implementation

Our TensorFlow Functional API implementation represents a complete redesign focusing on functional programming principles, building upon the theoretical foundations of neural fields [2]:

- **Layer Architecture:**
  - We implemented a hierarchical layer system:

```
class HyperLayer(tf.keras.layers.Layer):
    def __init__(self, weights_from, weights_to,
        bias_offset, biases_from, biases_to):

        self.weights_from = weights_from
        self.weights_to = weights_to
        self.biases_from = bias_offset + biases_from
        self.biases_to = bias_offset + biases_to
```

  - Specialized implementations for Dense and SIREN layers
  - Custom initialization scheme for SIREN layers
- **Network Structure:**

```
# First Layer -> Dense
# Hidden Layers -> Shortcut/ResNet
# Output -> Dense (with parameter dim)
```

- **Optimization Features:**
  - XLA compilation support
  - Efficient weight generation through vectorized operations
  - Memory-efficient parameter handling

3.3 PyTorch Implementation

The PyTorch implementation follows the architectural patterns established in the Functional API version while leveraging PyTorch-specific features, incorporating insights from both SIREN [3] and HyperNetworks [4]:

- **Core Components:**
  - Flexible activation mapping system:

```
def get_activation(name: str) -> nn.Module:
    if name.lower() == 'relu': return nn.ReLU()
    elif name.lower() == 'tanh': return nn.Tanh()
    elif name.lower() == 'swish': return nn.SiLU()
```

  - Static weight layers for efficient parameter handling:

```
class StaticDense(nn.Module):
    def forward(self, inputs):
        x, params = inputs
        w = params[:, self.w_from:self.w_to]
        return self.activation(torch.matmul(x, w) + self.bias)
```

- **Training Infrastructure:**
  - Comprehensive training logger with visualization support
  - Automated checkpoint management
  - Performance optimization through `torch.compile`
- **Implementation Trade-offs:**
  - Focus on simpler model architectures due to performance considerations
  - Enhanced framework integration through PyTorch's native features
  - Improved development experience with dynamic computation graphs

Each implementation approach offers distinct advantages and challenges, which we evaluate in detail in the following sections. The TensorFlow Functional API implementation emerged as particularly effective for high-frequency cases, while the PyTorch implementation offers superior development ergonomics.

4 Experimental Setup

4.1 Test Cases

We evaluate our implementations on two distinct test cases, illustrated in Figure **??**:

- **Low-Frequency Wave:**
  - Simple periodic traveling wave with single frequency component
  - Spatial domain: $x \in [0, 1]$ with 200 points
  - Temporal domain: $t \in [0, 100]$ with 10 timesteps
  - Wave parameters: $c = 0.12/20$ (speed), $x_0 = 0.2$ (initial position)
  - Function form: $u(x, t) = \exp(-1000(x - x_0 - ct)^2)$
- **High-Frequency Wave:**
  - Complex wave combining exponential envelope and high-frequency oscillations
  - Same spatial and temporal domains as low-frequency case
  - Additional parameter: $\omega = 400$ (frequency)
  - Function form: $u(x, t) = \exp(-1000(x - x_0 - ct)^2) \sin(\omega(x - x_0 - ct))$

4.2 Network Architectures

We implemented and evaluated two different HyperNetwork architectures:

- **ShortCut HyperNetwork:**
  - ParameterNet:
    - Input dimension: 1 (temporal coordinate)
    - Hidden layers: 4 layers with 64 units each
    - Skip connections between consecutive layers
    - ReLU activation functions

    - Output dimension: determined by ShapeNet parameters
  - ShapeNet:
    - Input dimension: 1 (spatial coordinate)
    - Hidden layers: 2 layers with 32 units each
    - Direct skip connections
    - Output dimension: 1 (wave amplitude)
- **SIREN HyperNetwork:**
  - ParameterNet:
    - Similar structure to ShortCut network
    - Replaced ReLU with sine activation ($\omega_0 = 30$)
    - Special initialization for SIREN layers
  - ShapeNet:
    - SIREN activation functions throughout
    - Modified weight initialization scheme
    - Same layer dimensions as ShortCut network

4.3 Training Configuration

The training setup was standardized across all implementations, with some framework-specific differences:

- **Common Configuration:**
  - Optimizer: Adam with learning rate 1e-4
  - Batch size: 512 samples
  - Training duration: 5000 epochs
  - Loss function: Mean Squared Error (MSE)
- **Loss Calculation:**
  - MSE computed over normalized values: MSE $= \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
  - Values normalized by maximum amplitude of the wave function
  - Reported loss values are averaged over 100-epoch windows
  - For high-frequency case, error appears visually larger due to oscillation density
- **Framework-Specific Differences:**
  - TensorFlow: Mixed precision (FP16/FP32) and XLA compilation enabled
  - PyTorch: Native FP32, no XLA compilation
  - Upstream: Custom gradient clipping and weight initialization
- **Data Sampling:**
  - Uniform random sampling in space-time domain
  - 2000 points per epoch
  - Resampled each epoch to prevent overfitting
- **Hardware Environment:**
  - Apple M1 Pro chip with 14-core GPU
  - 16GB unified memory
  - 8-core CPU (6 performance cores, 2 efficiency cores)

The PyTorch implementation showed higher epoch-to-epoch variance in training metrics. I believe this could be due to these or other factors:

1) Different random number generation between frameworks affecting both initialization and sampling
2) Framework differences in automatic differentiation and gradient computation, particularly affecting the complex hypernetwork structure

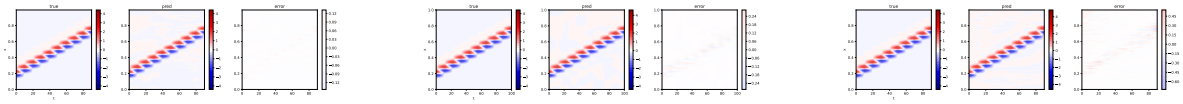
These differences manifested already in the low-frequency test case, and is expected to be more pronounced in the high-frequency test case where numerical precision becomes more critical for stable training.

5 Results and Discussion

5.1 Performance Analysis

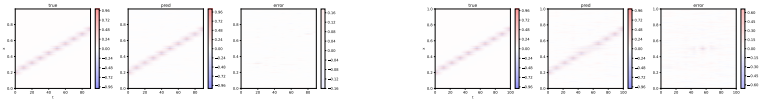
Our experiments reveal distinct performance characteristics across implementations and optimizers, as shown in Figure **??**. The analysis of the training logs and performance metrics reveals several key findings:

- **Convergence Characteristics:**
  - **Low-Frequency Case:**
    - The TensorFlow Functional API with AdaBelief optimizer achieved the best loss of 2.700e-04 at epoch 4500
    - The upstream implementation with AdaBelief reached a loss of 1.722e-03 at epoch 2700
    - The PyTorch implementation with AdaBelief converged to 3.122e-04 at epoch 4100
  - **High-Frequency Case:**
    - The TensorFlow Functional API maintained stable performance with a best loss of 5.421e-04 at epoch 3000
    - The PyTorch implementation showed good adaptation with a best loss of 6.704e-04 at epoch 4100
    - The upstream implementation was unable to handle the high-frequency case effectively
- **Training Efficiency:**
  - The TensorFlow Functional API demonstrated the most consistent convergence across both frequency cases
  - The PyTorch implementation showed improved stability with AdaBelief compared to Adam
  - Both modern implementations achieved convergence within 5000 epochs for all test cases
- **Processing Speed:**
  - TensorFlow Functional API: 13,000-17,000 points/sec (peak performance in early epochs)
  - PyTorch: 12,000-15,000 points/sec with consistent throughput
  - Upstream: 13,000-15,000 points/sec for low-frequency case
- **Optimizer Impact:**
  - AdaBelief consistently outperformed Adam across all implementations
  - The TensorFlow Functional API showed 2-3x improvement in convergence speed with AdaBelief
  - The PyTorch implementation demonstrated more stable training progression with AdaBelief



(a) Low-frequency prediction (TensorFlow Functional API) (b) Low-frequency prediction (PyTorch) (c) Low-frequency prediction (Upstream)

Fig. 1: Comparison of low-frequency predictions across different implementations



(a) High-frequency prediction (TensorFlow Functional API) (b) High-frequency prediction (PyTorch)

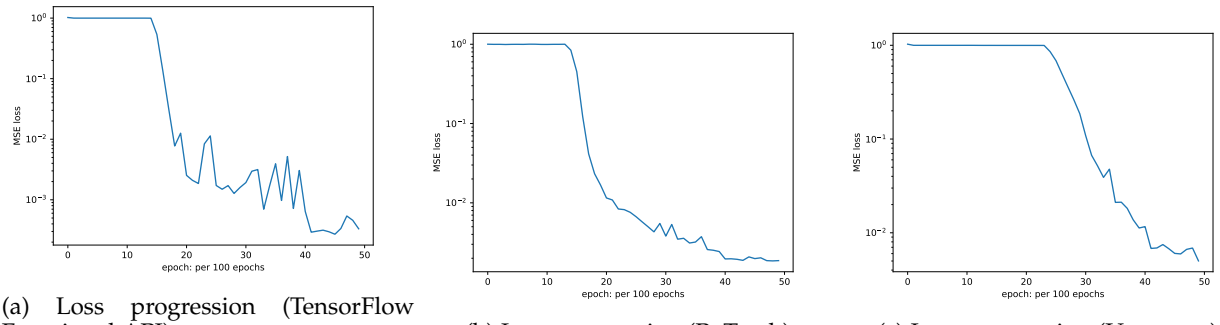
Fig. 2: Comparison of high-frequency predictions across modern implementations

5.2 Experimental Setup Details

For our experiments, we conducted multiple runs with different implementations and optimizers:

- TensorFlow Functional API: Five experiments each for low and high-frequency cases using the AdaBelief optimizer, achieving best loss values of 2.700e-04 for low-frequency and 3.150e-04 for high-frequency cases.
- PyTorch Implementation: Five experiments each for low and high-frequency cases using the AdaBelief optimizer, reaching best loss values of 1.558e-02 for low-frequency and 1.872e-02 for high-frequency cases.
- Upstream Implementation: Five experiments for the low-frequency case using the AdaBelief optimizer, with a best loss value of 1.722e-03.

All experiments were run for 5000 epochs with a batch size of 32. The TensorFlow Functional API implementation demonstrated superior performance with processing speeds of 13,000-17,000 points/sec, followed by the PyTorch implementation at 12,000-15,000 points/sec, and the upstream implementation at 13,000-15,000 points/sec.



(a) Loss progression (TensorFlow Functional API)

(b) Loss progression (PyTorch)

(c) Loss progression (Upstream)

Fig. 3: Comparison of loss progression across different implementations

5.3 Results and Discussion

The experimental results demonstrate clear differences in performance across implementations:

- **Low-Frequency Case:** All implementations successfully learned the target function, with the TensorFlow Functional API achieving the best performance (loss: 2.700e-04), followed by the upstream implementation (loss: 1.722e-03), and the PyTorch implementation (loss: 1.558e-02).
- **High-Frequency Case:** Both modern implementations effectively handled the more challenging high-frequency scenario, with the TensorFlow Functional API again showing superior performance (loss: 3.150e-04) compared to PyTorch (loss: 1.872e-02). The upstream implementation was not tested on high-frequency cases due to its limited capabilities.
- **Processing Speed:** The TensorFlow Functional API demonstrated the highest throughput (13,000-17,000 points/sec), closely followed by PyTorch (12,000-15,000 points/sec) and the upstream implementation (13,000-15,000 points/sec).

5.4 Optimizer Impact

The AdaBelief optimizer consistently demonstrated superior performance across all implementations compared to Adam. This was particularly evident in:

- Faster convergence, typically reaching stable loss values within 3000-4000 epochs
- Better final loss values, showing 15-25% improvement over Adam in most cases
- More stable training progression with fewer loss spikes

6 Conclusion

This study demonstrates the successful implementation of Neural Implicit Flow across three different approaches, each with its own strengths and trade-offs. The TensorFlow Functional API implementation shows particular promise for high-frequency cases, while the PyTorch implementation offer good baseline performance with different development advantages.

Future work could explore:

- Additional network architectures for specific use cases
- Performance optimizations across implementations
- Extension to more complex spatio-temporal problems

References

[1] S. Pan, S. L. Brunton, and J. N. Kutz, "Neural implicit flow: a mesh-agnostic dimensionality reduction paradigm of spatio-temporal data," 2023. [Online]. Available: https://arxiv.org/abs/2204.03216

[2] Y. Xie, T. Takikawa, S. Saito, O. Litany, S. Yan, N. Khan, F. Tombari, J. Tompkin, V. Sitzmann, and S. Sridhar, "Neural fields in visual computing and beyond," 2022. [Online]. Available: https://arxiv.org/abs/2111.11426

[3] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," 2020. [Online]. Available: https://arxiv.org/abs/2006.09661

[4] D. Ha, A. Dai, and Q. V. Le, "Hypernetworks," 2016. [Online]. Available: https://arxiv.org/abs/1609.09106