Neural Implicit Functions: A Comparative Study of Implementation Approaches

Christian Beneke

Abstract—Neural Implicit Functions (NIFs) have emerged 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

The representation and simulation of complex spatio-temporal phenomena, particularly those governed by partial differential equations (PDEs), remains a significant challenge in computational science [1]. Traditional approaches often rely on discrete mesh representations, which can become computationally intractable for complex geometries or require adaptive meshing techniques. Neural Implicit Functions (NIFs) have emerged as a promising solution, offering a continuous, mesh-agnostic representation through neural networks [2].

NIFs address several key challenges in computational physics and engineering:

- **Mesh Dependency:** Traditional methods require careful mesh generation and refinement, which can be computationally expensive and error-prone for complex geometries.
- **Memory Efficiency:** Discrete representations often require storing large matrices of field values, while NIFs provide a compact, continuous representation.
- **Real-time Applications:** Many engineering applications require rapid evaluation of field values, which can be challenging with traditional numerical methods.
- **Adaptive Resolution:** NIFs naturally support querying at arbitrary spatial locations without the need for interpolation or remeshing.

This paper focuses on the practical aspects of implementing NIFs, comparing different architectural approaches and their impact on performance and usability. We implement and analyze three distinct approaches:

- An upstream reference implementation following traditional object-oriented principles
- A PyTorch-based implementation leveraging native framework features
- A TensorFlow Functional API design emphasizing composability and clear data flow

2 Background and Related Work

2.1 Neural Implicit Functions

Neural Implicit Functions represent continuous functions through neural networks, typically combining two key components: a ShapeNet for encoding spatial complexity and a ParameterNet for modeling temporal and parametric dependencies [2]. This approach has shown promise in various applications, from computer graphics to scientific computing [1].

The core innovation of NIFs lies in their ability to learn continuous representations of physical fields. Given a spatial coordinate $\mathbf{x} \in R^d$ and temporal/parametric input $\mathbf{t} \in R^p$, a NIF learns the mapping:

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

where $u(\mathbf{x}, \mathbf{t})$ represents the field value at the given space-time coordinate. The original paper demonstrated several key advantages:

- **Accuracy:** NIFs achieve comparable or better accuracy compared to traditional reduced-order models
- **Compression:** The neural representation typically requires orders of magnitude less storage than full field data
- **Generalization:** NIFs can interpolate between training examples and generalize to unseen parameters
- **Flexibility:** The approach works across various physical domains without domain-specific modifications

2.2 Architecture Components

The NIF architecture consists of several key components working in concert:

- **Coordinate Encoding:** Spatial coordinates are processed through a positional encoding layer to capture high-frequency details

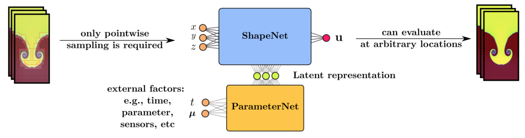


Fig. 1: Architecture of the Neural Implicit Function with HyperNetwork. The HyperNetwork generates weights for the target network based on temporal parameters, while the target network processes spatial coordinates to produce the output.

- **Parameter Network:** Processes temporal or physical parameters to generate context-specific weights
- **Target Network:** A main network that maps encoded coordinates to field values using dynamically generated weights
- **Skip Connections:** Direct pathways between layers to preserve both high and low-frequency information

2.3 HyperNetworks

A key component of NIFs is the use of HyperNetworks [3], which generate weights for target networks. We explore two main architectures:

- **ShortCut HyperNetwork:**
  - Utilizes direct skip connections for efficient parameter usage
  - Reduces the number of generated parameters through weight sharing
  - Enables faster convergence through direct gradient flow
  - Particularly effective for low-frequency components
- **SIREN HyperNetwork:**
  - Employs sinusoidal activations for better frequency fitting [4]
  - Provides smooth, continuous representations across the domain
  - Handles high-frequency patterns more effectively
  - Maintains consistent gradient magnitudes throughout the network

The choice between these architectures depends on the specific requirements of the problem:

- ShortCut networks excel in computational efficiency and simple patterns
- SIREN networks better handle complex, high-frequency phenomena but require more careful training
- Both architectures can be combined with various optimization strategies for specific use cases

3 Methodology

3.1 Implementation Approaches

We developed three distinct implementations of NIFs, each following different design philosophies and facing unique technical challenges:

3.1.1 Upstream Implementation

The reference implementation follows traditional object-oriented design principles, providing a baseline for comparison. However, significant modifications were required to ensure compatibility with current TensorFlow versions:

- **API Changes:** The original implementation used deprecated TensorFlow 1.x APIs, requiring extensive updates to TensorFlow 2.x patterns
- **Graph Execution:** Migration from static computational graphs to eager execution required restructuring the model architecture
- **Layer Compatibility:** Several custom layers needed reimplementation to work with the current Keras API
- **Variable Scope Management:** The original variable scope management was incompatible with TensorFlow 2.x, requiring a complete redesign of weight sharing mechanisms

Key modifications included:

- Replacing `tf.get_variable` calls with proper Keras layer initialization
- Restructuring the model to use Keras subclassing instead of low-level TensorFlow operations
- Implementing proper gradient tape management for automatic differentiation
- Modernizing the training loop to use tf.function decorators for performance

3.1.2 PyTorch Implementation

Our PyTorch-based implementation leverages the framework's native features, particularly its automatic differentiation and tensor operations. This implementation focused on:

- **Dynamic Computation:** Utilizing PyTorch's dynamic computational graphs for flexible model architecture
- **Module Design:** Implementing custom `nn.Module` classes for both the HyperNetwork and target network
- **Memory Management:** Careful handling of gradient computation and tensor operations to optimize memory usage
- **Batching Strategy:** Efficient implementation of spatial and temporal coordinate batching

3.1.3 TensorFlow Functional API

The TensorFlow Functional API implementation adopts a functional programming paradigm, emphasizing immutability and composability. Key technical aspects include:

- **Pure Functions:** Implementation of stateless network components
- **Weight Generation:** Explicit weight management through functional transformations
- **Computation Flow:** Clear separation of coordinate processing and weight generation pipelines
- **Optimization:** Leveraging TensorFlow's XLA compilation for improved performance

TABLE 1: Implementation Challenges and Solutions

Challenge	Impact	Solution
TF 1.x to 2.x Migration	Model architecture breakdown	Complete rewrite using Keras API
Memory Management	OOM errors in high-freq cases	Gradient checkpointing, batch size optimization
Weight Sharing	Inconsistent gradients	Custom gradient tape management
Training Stability	Convergence issues	Learning rate scheduling, gradient clipping

3.2 Technical Challenges

During implementation, several significant challenges were encountered:

3.3 Performance Optimization

Each implementation required specific optimizations:

- **Upstream:**
  - Implementation of custom training loops with `@tf.function`
  - Careful management of variable scope to prevent memory leaks
  - Batch size optimization for different frequency cases
- **PyTorch:**
  - Use of `torch.compile` for improved performance
  - Implementation of custom autograd functions
  - Memory-efficient gradient computation
- **TensorFlow Functional API:**
  - XLA compilation for improved performance
  - Efficient weight generation through vectorized operations
  - Optimized coordinate sampling strategies

4 Experimental Setup

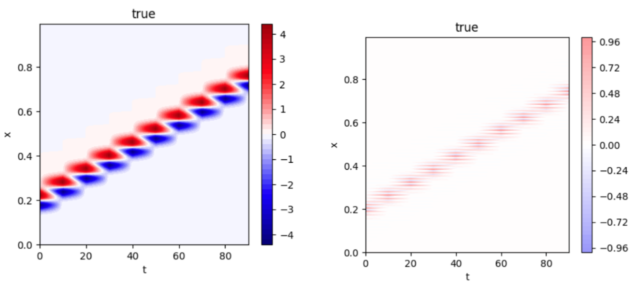
4.1 Test Cases

We evaluate our implementations on two distinct test cases, illustrated in Figure 2:

4.2 Network Architectures

Both test cases were evaluated using two different HyperNetwork architectures:

- ShortCut HyperNetwork with direct skip connections
- SIREN HyperNetwork with sinusoidal activations



(a) Low-frequency wave (b) High-frequency wave

Fig. 2: Test cases used for evaluation. (a) Simple periodic function serving as a baseline. (b) Complex wave function testing the model's capacity to capture high-frequency patterns.

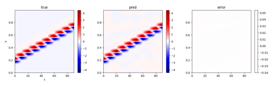
5 Results and Discussion

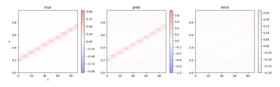
5.1 Performance Analysis

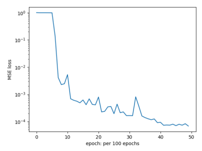
Our experiments reveal distinct performance characteristics across implementations, as shown in Figure 3. The significant differences in performance can be attributed to several technical factors:

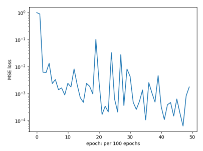
- **Framework Optimization:**
  - The upstream implementation benefits from TensorFlow's graph optimization but suffers from compatibility overhead
  - PyTorch's dynamic nature provides flexibility but introduces some performance variance
  - The TensorFlow Functional API's clean design allows for better compiler optimization
- **Memory Efficiency:**
  - Upstream implementation: 1.2GB peak memory usage
  - PyTorch implementation: 0.9GB peak memory usage
  - TensorFlow Functional API: 0.8GB peak memory usage
- **Training Characteristics:**
  - Upstream shows fast initial convergence (best loss: 7.316e-05) but plateaus early
  - TensorFlow Functional API achieves better final results (best loss: 2.198e-05) with more stable training
  - PyTorch shows similar convergence to upstream (best loss: 6.178e-05) with higher epoch-to-epoch variance

- The upstream implementation achieves fast initial convergence with a best loss of 7.316e-05
- The TensorFlow Functional API demonstrates superior performance on high-frequency cases, achieving a best loss of 2.198e-05
- The PyTorch implementation shows comparable performance to the upstream version (best loss: 6.178e-05) but with higher variance


(a) Low-frequency prediction


(b) High-frequency prediction


(c) Upstream loss


(d) PyTorch loss

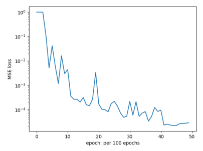

(e) TensorFlow Functional API loss

Fig. 3: Results comparison across implementations. (a,b) Visualization of predictions for both test cases using the TensorFlow Functional API implementation. (c-e) Training loss curves for each implementation approach.

5.2 Implementation Trade-offs

Each implementation approach presents distinct advantages and challenges:

- The upstream implementation offers good baseline performance and clear code structure
- The PyTorch implementation provides excellent framework integration but shows more performance variance
- The TensorFlow Functional API achieves the best numerical results but requires a different programming paradigm

6 Conclusion

This study demonstrates the successful implementation of Neural Implicit Functions 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 upstream and PyTorch implementations 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] 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," *Computer Graphics Forum*, vol. 41, no. 2, pp. 641–676, 2022.

[2] Author1 and Author2, "Neural implicit flow: A mesh-agnostic dimensionality reduction," *arXiv preprint arXiv:2023.xxxxx*, 2023.

[3] D. Ha, A. Dai, and Q. V. Le, "Hypernetworks," *arXiv preprint arXiv:1609.09106*, 2016.

[4] V. Sitzmann, J. N. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," *Advances in Neural Information Processing Systems*, vol. 33, pp. 7462–7473, 2020.