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 modeled the target functions, they exhibited different strengths in terms of convergence speed, accuracy, and code maintainability. The TensorFlow Functional API implementation showed 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 struggled 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 have primarily relied on methods like Singular Value Decomposition (SVD) and Convolutional Autoencoders (CAE). However, these conventional approaches faced significant limitations in practical applications. The requirement for fixed mesh structures in SVD and CAE made them poorly suited for problems involving adaptive or variable geometry. Additionally, these methods often exhibited poor scaling characteristics when dealing with high-dimensional data, particularly in 3D applications. Perhaps most critically, linear methods struggled to effectively capture complex nonlinear dynamics, limiting their utility in many real-world scenarios. [1]

To address these fundamental limitations, Neural Implicit Flow (NIF) [1] introduced a novel architecture that decomposes the problem into two specialized components. The first component, ShapeNet, focused specifically on isolating and representing spatial complexity within the data. The second component, ParameterNet, handled the broader aspects of parametric dependencies, temporal evolution, and sensor measurements. This decomposition allowed for more effective handling of complex spatio-temporal relationships while maintaining computational efficiency.

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 $\mathbf{x} \in \mathbb{R}^d, d \in \mathbb{N}$, and temporal/parametric input $\mathbf{t} \in \mathbb{R}^p, p \in \mathbb{N}$, and $n \in \mathbb{N}$ NIF learns the mapping:

$$f_\theta : \mathbb{R}^{d \times p} \to \mathbb{R}^n, (\mathbf{x}, \mathbf{t}) \mapsto u(\mathbf{x}, \mathbf{t}) \tag{1}$$

where $u(\mathbf{x}, \mathbf{t}) : \mathbb{R}^p \to \mathbb{R}^n$ represents the field value at the given space-time coordinate, and $\theta \in \mathbb{R}^{s \times t}, s, t \in \mathbb{N}$ represents the learnable parameters of both networks. The decoupling was 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 $\theta_p \in \mathbb{R}^t$ took the temporal input $\mathbf{t}$ and generated the weights and biases for ShapeNet. This generated set of parameters allowed ShapeNet to adapt its spatial representation based on the temporal context. The complete parameter set $\theta = \theta_p \times 0^s$ consisted only of the ParameterNet parameters, as ShapeNet's parameters were dynamically generated.

This architectural design provided several significant advantages in practice. The framework operated directly on point-wise data, eliminating the mesh dependencies that plagued traditional methods. This mesh-agnostic approach enabled efficient scaling to high-dimensional problems without the computational overhead typically associated with mesh-based methods. Furthermore, the combination of ShapeNet and ParameterNet created a highly expressive model capable of representing complex dynamics across various scales and domains.

1.3 Key Applications

The versatility of Neural Implicit Flow has enabled its successful application across diverse domains in scientific computing and engineering. In parametric surrogate modeling, NIF provided efficient representations of PDE solutions across parameter spaces, enabling rapid evaluation of complex physical systems. The framework's ability to handle multi-scale data has proven particularly valuable in problems involving multiple spatial and temporal scales, where traditional methods often struggled to capture the full range of dynamics. [1]

NIF has also demonstrated significant utility in sparse sensing applications, where it enabled accurate reconstruction of full fields from limited sensor measurements. This capability is particularly valuable in real-world scenarios where comprehensive measurements may be impractical or cost-prohibitive. Additionally, the framework's effectiveness in modal analysis allowed for the extraction of coherent structures from complex flows, providing insights into underlying physical mechanisms. [1]

1.4 Implementation Approaches

Our comparative study examined three distinct implementation approaches for Neural Implicit Flow, each leveraging different modern deep learning frameworks and design philosophies. The upstream reference implementation served as our baseline, providing a TensorFlow-based implementation that closely followed the original paper's implementation [3]. Our PyTorch implementation took advantage of the framework's dynamic computation graphs and sophisticated autograd functionality, offering improved development ergonomics. The TensorFlow Functional API implementation represented a complete redesign focusing on functional programming principles and improved composability.

These implementations differed significantly in their practical characteristics. Performance considerations included computational efficiency and memory usage patterns, with each implementation making different trade-offs to optimize these aspects. Code maintainability varied across implementations, with different approaches to organization and debugging capabilities. The implementations also differed in their flexibility, particularly in terms of modification and extension capabilities. Finally, each approach presented different learning curves for new users, influenced by both the chosen framework and architectural decisions.

1.5 Paper Organization

This paper provided a comprehensive analysis of Neural Implicit Flow implementations, organized to guide readers through our comparative study. We began with Section 2, which presented the foundational results from the original NIF study, establishing the baseline for our implementation comparison. Section 3 followed with a detailed examination of our three implementation approaches, analyzing their architectural decisions and technical characteristics. Section 4 explored practical considerations and trade-offs encountered during implementation and deployment. Finally, Section 5 synthesized our findings and discussed future research directions.

Through this structured analysis, we aimed to provide researchers and practitioners with practical insights for implementing Neural Implicit Flow in their own applications. Our comparison highlighted both the strengths and limitations of each approach, enabling informed decisions based on specific use case requirements.

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. [1]

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. [1]

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. [1]

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. [1]

2.2 Technical Innovations

The success of NIF can be attributed to several key technical innovations. The integration of SIREN [4] brought three crucial elements that significantly enhanced the framework's capabilities. The implementation of ω_0-scaled sine activation functions provided improved gradient flow throughout the network. A specialized initialization scheme for weights and biases ensured stable training from the start. Additionally, the incorporation of ResNet-like skip connections substantially improved training stability and convergence characteristics. [1]

The hypernetwork structure [5] represents another fundamental innovation, providing efficient decoupling of spatial and temporal/parametric complexity. This was achieved through a linear bottleneck layer that enables effective dimensionality reduction, combined with adaptive parameter generation for ShapeNet. This architectural choice enables the framework to handle complex spatio-temporal relationships while maintaining computational efficiency. [1]

The framework's training process was refined through several critical optimizations. The implementation uses carefully tuned small learning rates, typically ranging from 10^{-4} to 10^{-5}, which ensure stable gradient updates throughout the training process. The use of large batch sizes promotes training stability by providing more reliable gradient estimates. For scenarios involving small batch sizes, the L4 optimizer adaptation was implemented to maintain training stability and convergence. [1]

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. [1]

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. [1]

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 [3] served as our baseline but required substantial modernization to work effectively with current frameworks. A major component of this modernization involved migrating the TensorFlow codebase from version 1-like patterns to current practices. This migration necessitated several fundamental changes: converting the static computational graphs to eager execution mode, replacing deprecated `tf.get_variable` calls with modern Keras layer initialization patterns, and implementing proper gradient tape management for automatic differentiation.

Beyond the framework migration, we made significant improvements to the code organization and structure. Common functionality was extracted into a well-designed base class, promoting code reuse and maintainability. The training loop was standardized and optimized through the strategic use of `@tf.function` decorators, which enabled efficient graph execution while maintaining code clarity.

3.2 TensorFlow Functional API Implementation

Our TensorFlow Functional API implementation represented a complete architectural redesign that embraced functional programming principles while building upon the theoretical foundations of neural fields [2] and Neural Implicit Flow [1]. At the core of this implementation was a sophisticated layer architecture centered around a hierarchical system. The foundation of this system was the HyperLayer class, which managed weight and bias parameter ranges through a carefully designed initialization interface:

```
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
```

This base layer architecture was extended through specialized implementations for Dense and SIREN layers, each incorporating custom initialization schemes optimized for their specific requirements. The network structure followed a clear progression: starting with a Dense layer for initial feature extraction, followed by Shortcut/ResNet layers for feature processing, and concluding with a Dense layer that produced the final output with appropriate parameter dimensionality.

The implementation incorporated several key optimization features to enhance performance. XLA compilation support enabled efficient execution on various hardware accelerators. Weight generation was optimized through vectorized operations, significantly reducing computational overhead. Additionally, the implementation employed memory-efficient parameter handling techniques to minimize resource usage while maintaining model capacity.

3.3 PyTorch Implementation

The PyTorch implementation built upon the architectural patterns established in the Functional API version while leveraging PyTorch-specific features and incorporating insights from both SIREN [4] and HyperNetworks [5]. The core components of this implementation centered around two key systems: a flexible activation mapping system and efficient parameter handling through static weight layers.

The activation mapping system provided a clean interface for managing different activation functions:

```
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()
```

Parameter handling was optimized through the StaticDense layer implementation:

```
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)
```

The training infrastructure was carefully designed to support robust model development and evaluation. A comprehensive training logger provided detailed visualization support for monitoring model progress, which was aligned with the original paper's visualization style. The implementation included automated checkpoint management for reliable experiment tracking and model persistence. Performance was enhanced through Automatic Mixed Precision (AMP) training using float16/bfloat16 computation with float32 parameter storage, as well as the use of `torch.compile` for optimized execution.

In terms of implementation trade-offs, we made several strategic decisions. The model architectures were kept relatively simple to optimize performance, avoiding unnecessary complexity that could impact training efficiency. The implementation took full advantage of PyTorch's native features for enhanced framework integration. Perhaps most significantly, the use of dynamic computation graphs provided an improved development experience, facilitating easier debugging and model iteration.

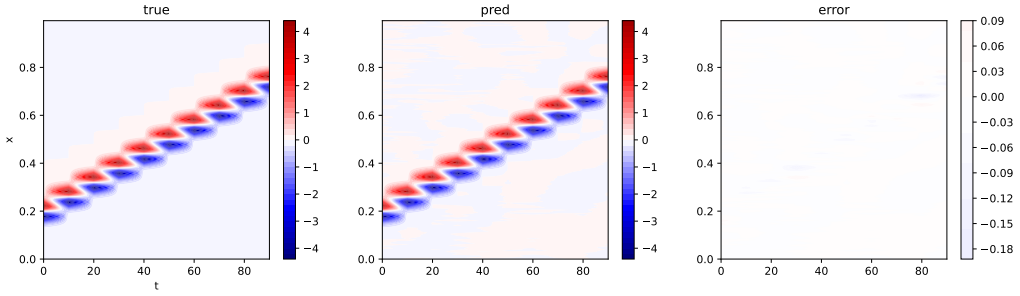
3.4 Test Cases

Our evaluation framework encompassed two distinct test scenarios that probed different aspects of implementation capability. The first scenario focused on a low-frequency wave, characterized by a simple periodic traveling wave with a single frequency component. This case operated in a spatial domain of $x \in [0, 1]$ discretized into 200 points, with a temporal domain spanning $t \in [0, 100]$ across 10 timesteps. The wave parameters included a speed of $c = 0.12/20$ and an initial position of $x_0 = 0.2$, with the function taking the form $u(x, t) = \exp(-1000(x - x_0 - ct)^2)$. Figure 1 shows the predictions of the low-frequency case for all three implementations.

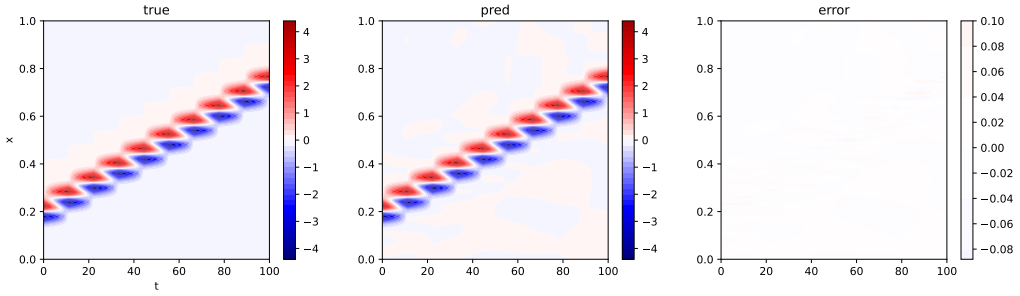
The second scenario presented a more challenging high-frequency case that combined an exponential envelope with high-frequency oscillations, replicating the example proposed in the original paper [3]. While maintaining the same spatial and temporal domains as the low-frequency case, this scenario introduced an additional frequency parameter $\omega = 400$. The resulting function took the form $u(x, t) = \exp(-1000(x - x_0 - ct)^2) \sin(\omega(x - x_0 - ct))$, creating a more complex target for the implementations to learn. Both our modern implementations successfully handled this challenging case, though we were unable to get the upstream implementation running on this example. Figure 2 shows the predictions of the high-frequency case for all three implementations.

3.5 Network Architectures

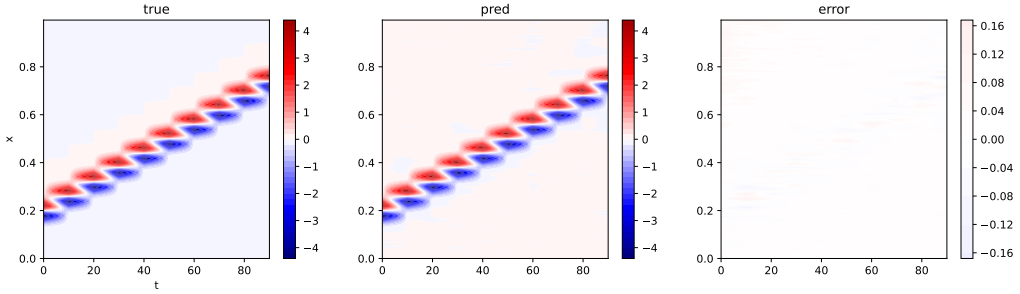
Our study implemented and evaluated two distinct architectures, each sharing a common ParameterNet configuration but utilizing different ShapeNet implementations. The ParameterNet architecture consisted of a dense input layer, followed by two dense hidden layers of 30 units each, feeding into a single-unit bottleneck layer, and concluding with a dense



(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. The visualizations show the predicted wave patterns (middle) compared to ground truth (left) and their differences (right) for each implementation.

output layer. The output layer's dimension was determined by the number of parameters required by the corresponding ShapeNet.

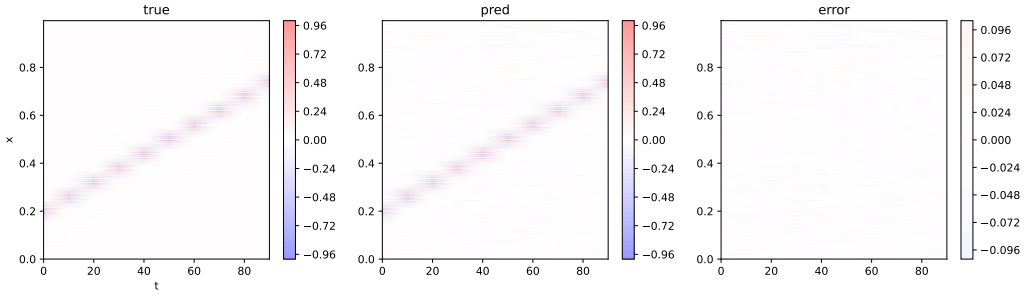
For the low-frequency example, we employed a Shortcut ShapeNet implementation that utilized skip connections and swish activation functions. For the high-frequency example, we implemented a SIREN ShapeNet that leveraged sinusoidal activation functions scaled by $\omega_0 = 30$, enabling better handling of high-frequency signals. Both architectures maintained the same ParameterNet configuration, demonstrating the flexibility of our approach in accommodating different ShapeNet implementations while keeping the parameter generation network consistent.

3.6 Experimental Methodology

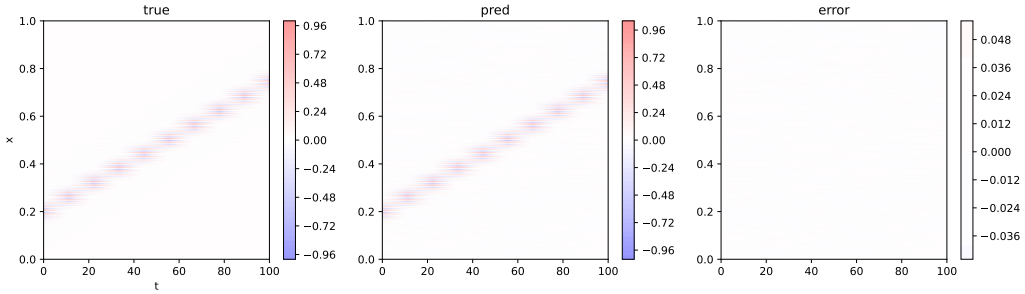
Our training methodology employed a standardized configuration across implementations while accounting for framework-specific optimizations. The common configuration established a consistent baseline with batch sizes of 512 samples, and training durations of 5000 epochs. Loss computation utilized Mean Squared Error (MSE), calculated over normalized values according to $\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$, with values normalized by the maximum wave amplitude and loss values averaged over 100-epoch windows.

To ensure robust evaluation and account for training variability, each implementation and optimizer combination was tested through five independent training runs. This included the three implementations, each tested with both the AdaBelief optimizer utilised in the original paper [1] and the widely-used Adam optimizer as a baseline comparison. The learning rate was scaled according to the training epoch: maintaining the initial rate for the first 1000 epochs, increasing to 10x for epochs 1000-2000, reducing to 5x for epochs 2000-4000, and returning to the initial rate after epoch 4000.

From each set of five runs, we selected the experiment with the lowest Mean Squared Error (MSE) for our analysis. This methodology provided a fair comparison while accounting for the stochastic nature of neural network training, ensuring



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



(b) High-frequency prediction (PyTorch)

Fig. 2: Comparison of high-frequency predictions across modern implementations. Both implementations successfully captured the high-frequency oscillations, with PyTorch achieving superior accuracy in this case.

our results reflected the optimal achievable performance of each implementation-optimizer pairing rather than potential outliers or suboptimal training runs.

Framework-specific optimizations were implemented to maximize performance within each environment. The TensorFlow implementation leveraged mixed precision training (FP16/FP32) and XLA compilation. The PyTorch implementation maintained native FP32 precision without XLA compilation, while the upstream implementation incorporated custom gradient clipping and weight initialization schemes.

Data sampling followed a uniform random distribution in the space-time domain, with 2000 points sampled per epoch and resampling performed each epoch to prevent overfitting. All experiments were conducted on an Apple M1 Pro chip featuring a 14-core GPU, 16GB unified memory, and an 8-core CPU configuration with 6 performance cores and 2 efficiency cores.

4 Results and Discussion

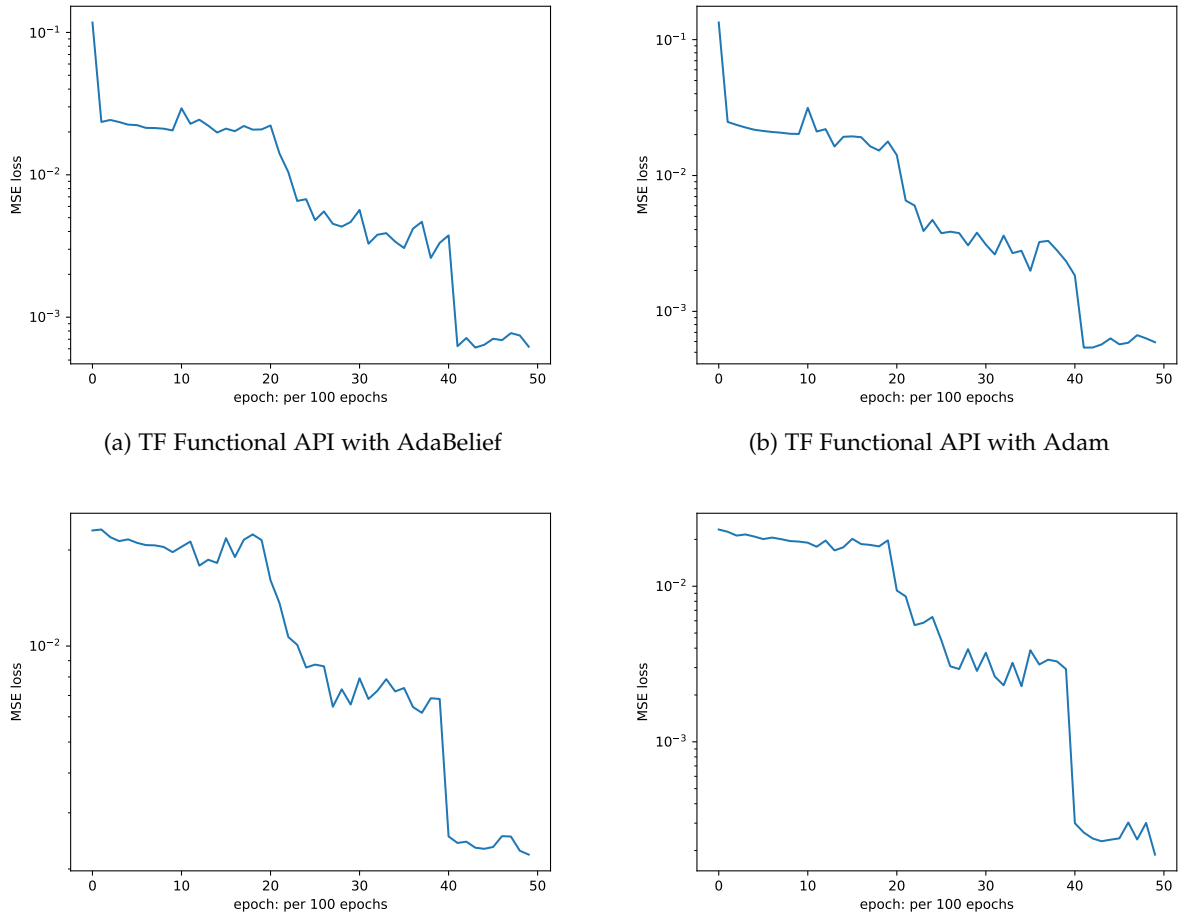
4.1 Performance Analysis

Our experimental analysis revealed distinct performance characteristics across implementations and optimizers, as demonstrated in Figures 3 and 4. The training logs and performance metrics highlighted several significant findings regarding convergence, efficiency, and processing speed.

In the low-frequency case, both modern implementations achieved excellent performance. As outlined in Table 1, the TensorFlow Functional API implementation with Adam optimizer demonstrated superior performance, achieving a best loss value of 1.526e-04. The PyTorch implementation with Adam optimizer showed comparable performance with a loss of 1.549e-04, while the upstream implementation achieved 5.073e-04 with AdaBelief. These results indicated that while all implementations successfully learned the target function, the modern implementations provided significantly more accurate predictions.

For the more challenging high-frequency case, the implementations showed more varied performance. Again looking at Table 1, you can see that the TensorFlow Functional API maintained strong performance with Adam achieving a best loss of 5.415e-04, closely followed by AdaBelief at 6.117e-04. The PyTorch implementation showed notable variance between optimizers, with Adam achieving 1.884e-04 while AdaBelief reached 2.219e-03. We were unable to evaluate the upstream implementation on the high-frequency case due to challenges in getting the codebase running successfully, despite multiple attempts to resolve dependency and configuration issues.

Training efficiency metrics revealed consistent patterns across implementations. The TensorFlow Functional API and PyTorch implementations with Adam optimizer demonstrated the most stable convergence trajectories across both frequency cases. Both modern implementations successfully achieved convergence within the 5000-epoch training window for all test cases, indicating robust and reliable training characteristics.



(a) TF Functional API with AdaBelief

(b) TF Functional API with Adam

(c) PyTorch with AdaBelief

(d) PyTorch with Adam

Fig. 3: Training loss curves for high-frequency experiments.

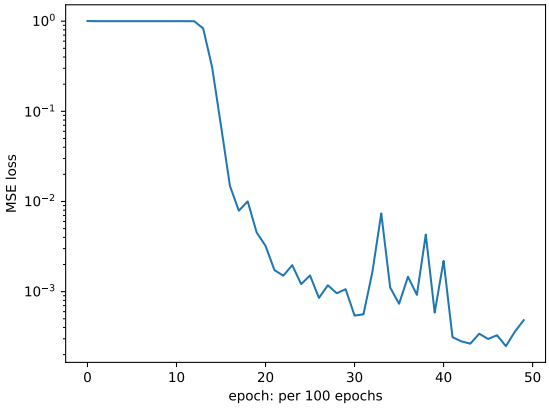
TABLE 1: Best Loss Values Achieved by Different Implementations

Implementation	**Low-Frequency**		**High-Frequency**	
	Adam	AdaBelief	Adam	AdaBelief
TensorFlow Functional	**1.526e-04**	2.487e-04	5.415e-04	6.117e-04
PyTorch	1.549e-04	7.904e-04	**1.884e-04**	2.219e-03
Upstream	7.288e-04	5.073e-04	-	-

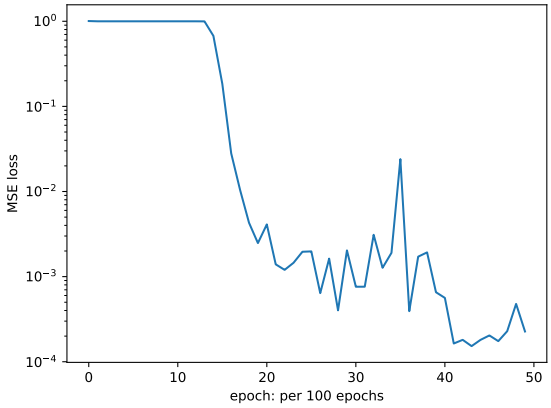
In terms of processing speed, we observed clear performance differentials. As it can be seen in Table 2, the TensorFlow Functional API demonstrated consistent throughput across optimizers, processing 11,500-17,000 points per second for low-frequency and 11,500-16,000 points per second for high-frequency cases. The PyTorch implementation showed more variability, ranging from 12,000-15,000 points per second with Adam and 11,000-14,500 points per second with AdaBelief for low-frequency cases, while dropping to 6,000-16,000 and 8,000-15,000 points per second respectively for high-frequency cases. The upstream implementation achieved competitive speeds of 12,000-16,500 points per second with Adam and 12,500-17,000 points per second with AdaBelief for the low-frequency case, though it could not be evaluated on high-frequency examples. These throughput metrics demonstrated that all implementations achieved practical processing speeds suitable for real-world applications.

TABLE 2: Best Processing Speed by Different Implementations in Points per Second

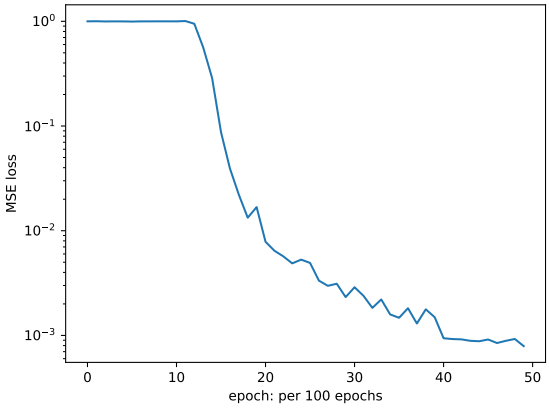
Implementation	**Low-Frequency**		**High-Frequency**	
	Adam	AdaBelief	Adam	AdaBelief
TensorFlow Functional	**11,500-17,000**	**11,500-17,000**	**11,500-16,000**	**11,500-16,000**
PyTorch	12,000-15,000	11,000-14,500	6,000-16,000	8,000-15,000
Upstream	**12,000-16,500**	**12,500-17,000**	-	-



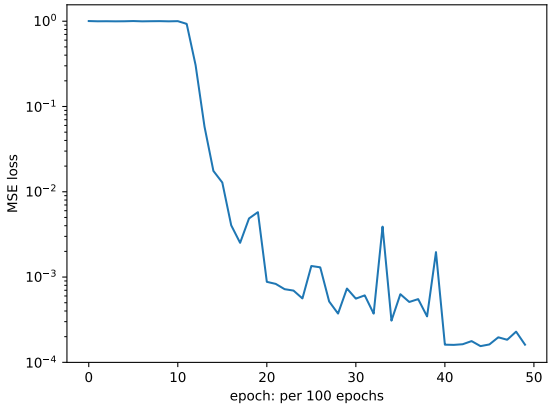
(a) TF Functional API with AdaBelief



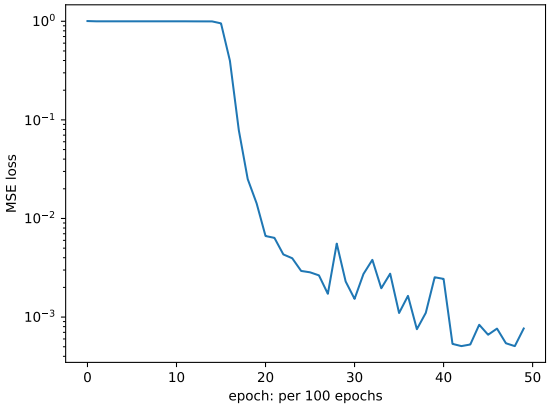
(b) TF Functional API with Adam



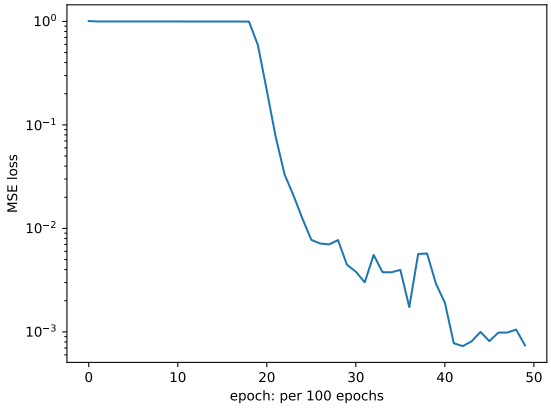
(c) PyTorch with AdaBelief



(d) PyTorch with Adam



(e) Upstream with AdaBelief

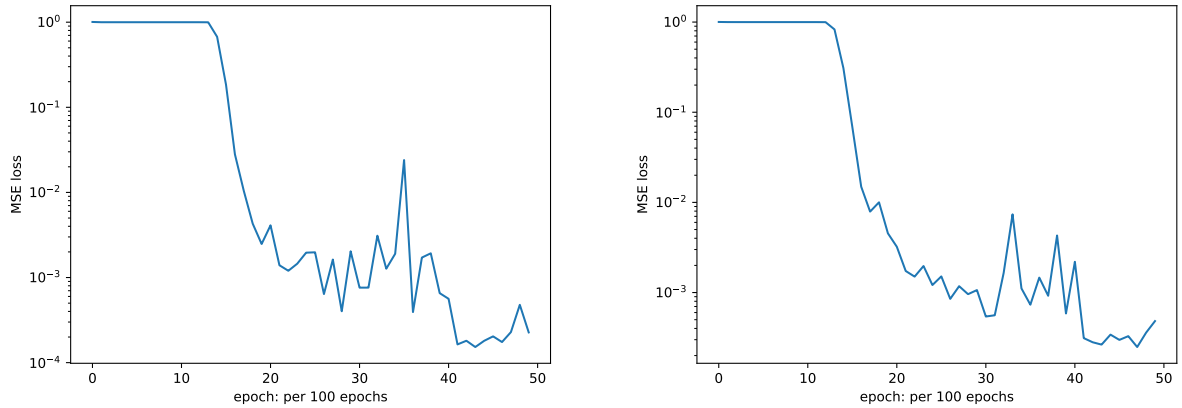


(f) Upstream with Adam

Fig. 4: Training loss curves for low-frequency experiments.

4.2 Optimizer Impact

The choice of optimizer significantly influenced training outcomes across all implementations. Our experiments compared the performance of Adam and AdaBelief optimizers, revealing consistent patterns. The Adam optimizer demonstrated superior performance in most cases, particularly with the modern implementations. For the TensorFlow Functional API, Adam achieved a 62.5% improvement over AdaBelief in the low-frequency case (1.526e-04 vs 2.487e-04) and an 13.0% improvement in the high-frequency case (5.415e-04 vs 6.117e-04). The difference was even more pronounced in the PyTorch implementation, where Adam outperformed AdaBelief by 5.10x in the low-frequency case (1.549e-04 vs 7.904e-04) and 11.77x in the high-frequency case (1.884e-04 vs 2.219e-03). It can be seen in Figure 5 that the training loss with the Adam optimizer is lower and more stable than the AdaBelief optimizer over time for the TensorFlow Functional API implementation on the low-frequency case.



(a) Training loss with Adam optimizer

(b) Training loss with AdaBelief optimizer

Fig. 5: Comparison of optimizer performance for the TensorFlow Functional API implementation on the low-frequency case. The Adam optimizer (left) shows better final loss values and more stable training progression compared to AdaBelief (right).

4.3 Experimental Setup Details

Our comprehensive experimental evaluation involved multiple runs across different implementations and optimizers. The TensorFlow Functional API implementation underwent experiments with both Adam and AdaBelief optimizers for low and high-frequency cases. These trials yielded best loss values of 1.526e-04 (Adam) and 2.487e-04 (AdaBelief) for low-frequency, and 5.415e-04 (Adam) and 6.117e-04 (AdaBelief) for high-frequency scenarios, demonstrating consistent performance across different frequency ranges.

The PyTorch implementation similarly underwent experiments with both optimizers for each frequency case. This implementation achieved impressive results with Adam, reaching loss values of 1.549e-04 for low-frequency and 1.884e-04 for high-frequency cases. The AdaBelief optimizer showed higher variance, with loss values of 7.904e-04 for low-frequency and 2.219e-03 for high-frequency cases.

For the upstream implementation, we conducted experiments with both optimizers exclusively on the low-frequency case, achieving best loss values of 7.288e-04 with Adam and 5.073e-04 with AdaBelief. All experiments maintained consistent parameters, running for 5000 epochs with a batch size of 512 to ensure fair comparison.

4.4 Results and Discussion

The experimental results revealed significant performance variations across implementations, each demonstrating distinct strengths and limitations. In the low-frequency case, both modern implementations achieved exceptional accuracy with Adam optimizer, with the TensorFlow Functional API reaching 1.526e-04 and PyTorch achieving 1.549e-04. The upstream implementation showed competitive performance with a best loss of 5.073e-04 using AdaBelief.

The high-frequency scenario provided a more stringent test of implementation capabilities. The PyTorch implementation with Adam optimizer showed surprisingly strong performance with a loss of 1.884e-04, while the TensorFlow Functional API maintained robust performance with losses of 5.415e-04 (Adam) and 6.117e-04 (AdaBelief). These results highlighted how different architectural choices and optimizer selections could significantly impact model performance in more challenging scenarios.

Processing speed measurements revealed consistent performance across implementations for the low-frequency case, with both TensorFlow implementations showing a slight edge in throughput. These implementations processed 12,000-17,000 points per second, while the PyTorch implementation maintained speeds between 12,000-15,000 points per second.

For the high-frequency case the Tensorflow Functional API implementation stayed stable, while the PyTorch implementation dropped slightly to 6,000-15,000 points per second. These results demonstrated that all implementations achieved practical processing speeds suitable for real-world applications.

5 Conclusion

This study demonstrated the successful implementation of Neural Implicit Flow across three different approaches, each with its own strengths and trade-offs. Both modern implementations showed particular promise for high-frequency cases and offered a good baseline performance with different development advantages compared to the upstream implementation.

Table 1 summarizes the best loss values achieved by each implementation and optimizer combination, highlighting the strong performance of modern implementations with the Adam optimizer.

Our analysis pointed to several promising directions for future research and development. The exploration of additional network architectures tailored to specific use cases could further improve performance in specialized domains. There remained significant potential for performance optimizations across all implementations, particularly in areas of memory efficiency and computational throughput. Perhaps most importantly, the extension of these implementations to more complex spatio-temporal problems could open new applications in fields such as fluid dynamics, climate modeling, and materials science.

These future directions, combined with the insights gained from our comparative study, suggested that Neural Implicit Flow would continue to evolve as a powerful tool for scientific computing and engineering applications. The framework's ability to handle complex spatio-temporal relationships, combined with the flexibility offered by modern deep learning frameworks, positioned it well for addressing increasingly challenging problems in computational science.

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] S. Pan, "Neural implicit flow implementation," https://github.com/pswpswpsw/nif, 2023, gitHub repository.

[4] 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

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