WaveDL: A Scalable Deep Learning Framework for Guided Wave Inversion on High-Performance Computing Clusters

Ductho Le[a,*]

[a]University of Alberta, Edmonton, Alberta, Canada

ARTICLE INFO

Keywords:
deep learning framework
guided wave inversion
distributed training
high-performance computing
memory-mapped data
multi-GPU synchronization
neural network regression

ABSTRACT

Ultrasonic guided wave inspection is a powerful technique for non-destructive evaluation (NDE) of structures such as pipelines, aircraft components, and composite materials. A key challenge in this field is the *inverse problem*: determining material properties (thickness, elastic moduli, density) from measured wave signals. Traditional approaches rely on iterative optimization coupled with physics-based forward models, which can be computationally expensive and time-consuming.

Deep learning offers a promising alternative by learning direct mappings from signal representations to material properties. However, training deep neural networks on the large-scale datasets typical in guided wave research presents significant engineering challenges: datasets often exceed available memory, multi-GPU training requires careful synchronization, and domain-specific metrics need integration with standard training pipelines.

This paper introduces **WaveDL** (Wave Deep Learning), an open-source Python framework specifically designed to address these challenges. WaveDL provides: (1) a memory-efficient data pipeline using memory-mapped files that enables training on datasets larger than available RAM; (2) an architecture-agnostic design allowing researchers to easily integrate and compare different neural network architectures; (3) robust distributed training support with proper synchronization across multiple GPUs; and (4) physics-aware metrics that track prediction accuracy in meaningful physical units.

WaveDL is built on PyTorch and HuggingFace Accelerate, supports mixed-precision training for improved performance, and includes comprehensive experiment tracking through Weights & Biases integration. The framework is available under the permissive MIT license at https://github.com/ductho-le/WaveDL.

Program Summary

Program title: WaveDL (Wave Deep Learning)
Licensing provisions: MIT License
Programming language: Python 3.11+
Repository: https://github.com/ductho-le/WaveDL
Nature of problem: Training deep neural networks for guided wave inverse problems faces three computational bottlenecks: (1) datasets often exceed GPU memory, (2) multi-GPU training requires complex synchronization, and (3) domain scientists need metrics in physical units rather than normalized loss values.
Solution method: WaveDL implements memory-mapped data loading for out-of-core training, DDP-safe synchronization primitives for distributed early stopping and metric aggregation, and automatic inverse transformation of predictions to physical units.
External routines/libraries: PyTorch (≥2.0), HuggingFace Accelerate (≥0.20), NumPy (≥1.24), SciPy (≥1.10), scikit-learn (≥1.2), pandas (≥2.0), matplotlib (≥3.7), tqdm (≥4.65), Weights & Biases (≥0.15, optional)
Restrictions: The framework is designed for multi-output regression tasks on 2D input representations. Classification tasks or 1D/3D inputs require modifications to the data pipeline and loss functions. Training requires NVIDIA GPU with CUDA support; CPU-only inference is supported.

*Corresponding author
ductho.le@outlook.com (D. Le)
ORCID(s):

1. Introduction

1.1. Background: Ultrasonic Guided Waves in Non-Destructive Evaluation

Ultrasonic guided waves are elastic wave modes that propagate within bounded media such as plates, pipes, shells, and layered composites. Unlike conventional bulk ultrasonic waves that travel through the material thickness, guided waves can propagate over extended distances (tens to hundreds of meters) while remaining sensitive to structural features throughout the waveguide cross-section [1, 2]. This unique property makes them invaluable for the rapid inspection of large-scale infrastructure including pipelines, railway tracks, aircraft fuselages, and wind turbine blades.

The physics of guided wave propagation is governed by the elastic wave equation in bounded media. For an isotropic plate of thickness h, the displacement field $\mathbf{u}(\mathbf{x}, t)$ satisfies:

$$\rho \frac{\partial^2 \mathbf{u}}{\partial t^2} = (\lambda + \mu)\nabla(\nabla \cdot \mathbf{u}) + \mu \nabla^2 \mathbf{u} \tag{1}$$

where ρ is the mass density, and λ, μ are the Lamé constants related to the longitudinal (c_L) and shear (c_T) wave velocities. The boundary conditions imposed by the traction-free surfaces give rise to a family of discrete wave modes (Lamb waves in plates, torsional and longitudinal modes in pipes) whose phase velocity c_p depends on the frequency-thickness product fh. This relationship, known as the *dispersion curve*, encodes the material's mechanical properties and geometry.

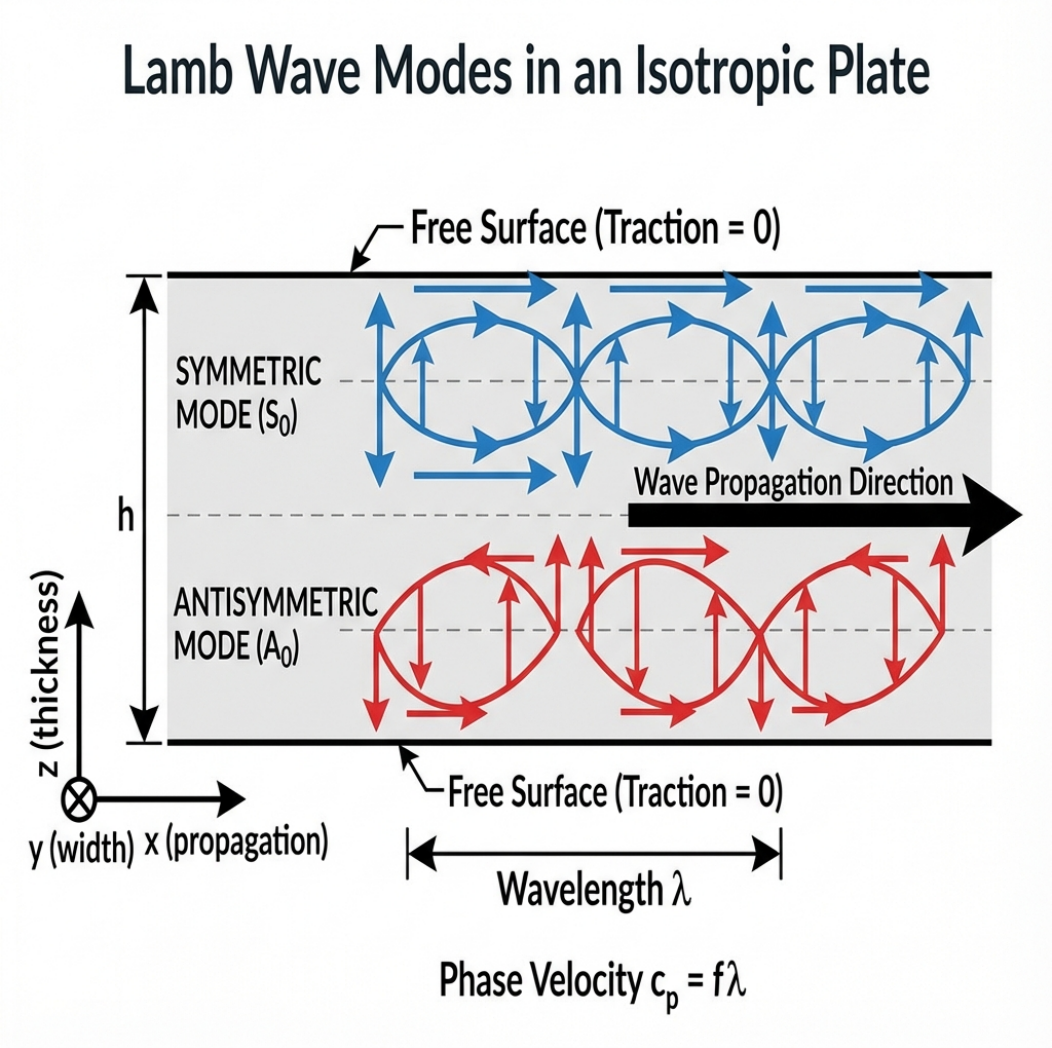

Figure 1: Lamb wave modes in an isotropic plate. Symmetric modes (S0, S1) exhibit in-plane particle motion while antisymmetric modes (A0, A1) show bending-like displacement. The dispersion relationship $c_p(f \cdot h)$ encodes material properties.

1.2. The Inverse Characterization Problem

The fundamental challenge in guided wave-based material characterization is the *inverse problem*: given an observed set of dispersion data $\mathbf{x}$ (e.g., a frequency-wavenumber spectrum or time-frequency representation), determine the material property vector $\mathbf{p} = [h, c_L, c_T, \rho, ...]^T$ that produced it.

Formally, let $\mathcal{G} : \mathcal{P} \to \mathcal{X}$ denote the *forward operator* that maps material properties $\mathbf{p} \in \mathcal{P}$ to observable dispersion data $\mathbf{x} \in \mathcal{X}$. The inverse problem seeks the operator $\mathcal{G}^{-1}$ such that:

$$\mathbf{p} = \mathcal{G}^{-1}(\mathbf{x}) \tag{2}$$

This problem is well-posed in the Hadamard sense only under specific conditions [3]. In practice, noise, incomplete mode identification, and model-data discrepancies introduce ill-posedness, requiring regularization techniques.

Traditional solution approaches fall into two categories:

1. **Iterative Optimization**: Given an initial guess $\mathbf{p}_0$, iteratively refine $\mathbf{p}$ to minimize the objective:

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \|\mathcal{G}(\mathbf{p}) - \mathbf{x}_{obs}\|^2 + \lambda R(\mathbf{p}) \tag{3}$$

   where $R(\mathbf{p})$ is a regularization term. Methods include Levenberg-Marquardt, genetic algorithms, and particle swarm optimization. These require repeated forward model evaluations (seconds to minutes each for complex layered media), making real-time inversion impractical.
2. **Look-up Tables**: Pre-compute $\mathcal{G}(\mathbf{p}_i)$ for a dense grid $\{\mathbf{p}_i\}$ in parameter space and perform nearest-neighbor matching. While fast at inference, this approach scales poorly with the number of parameters (curse of dimensionality) and provides only discrete solutions.

1.3. Deep Learning as an Alternative Paradigm

Deep neural networks offer a fundamentally different approach: learn the inverse mapping $\mathcal{G}^{-1}$ directly from paired

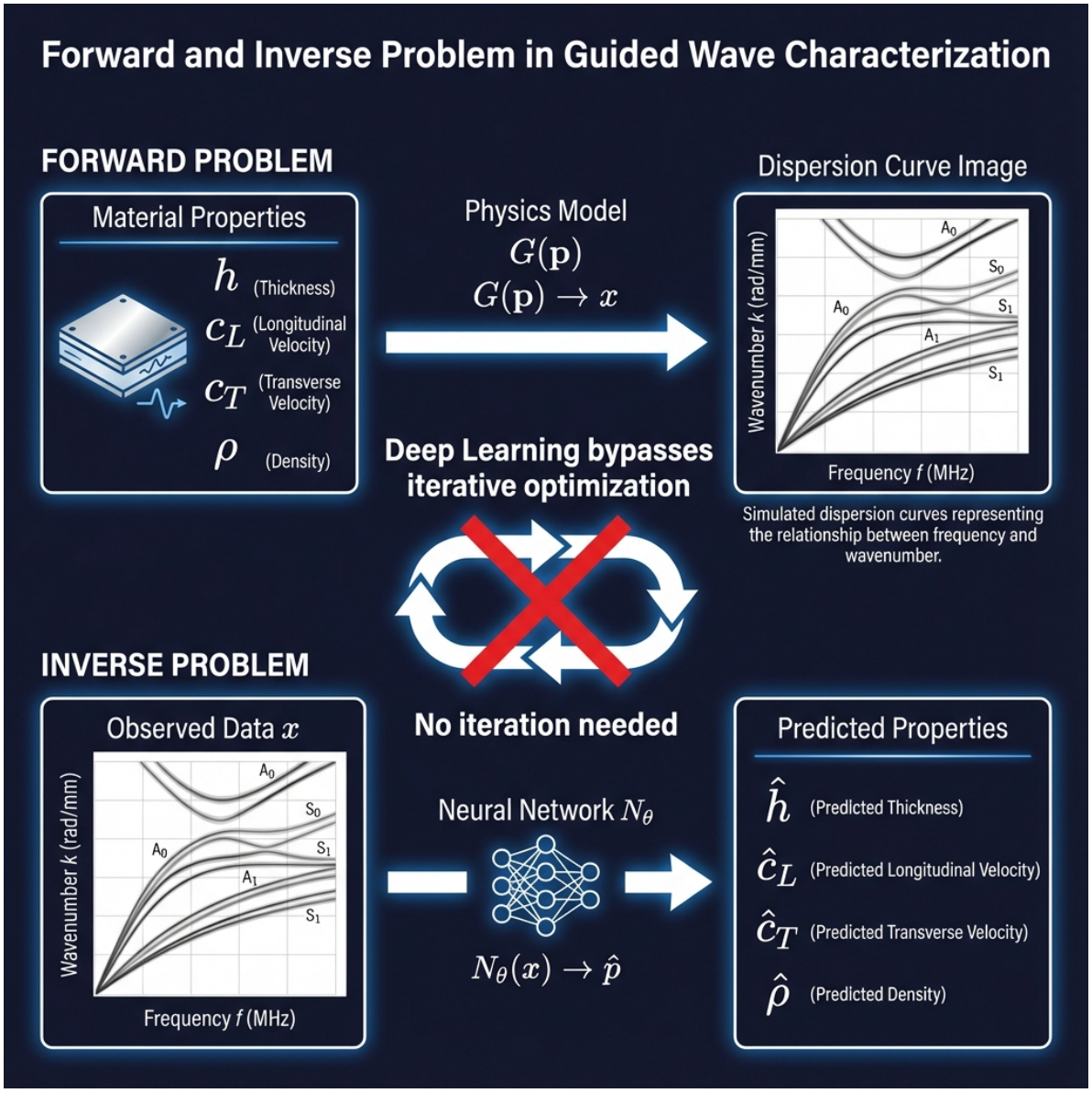

Figure 2: Forward and inverse problems in guided wave characterization. The forward problem maps material properties to dispersion curves via physics-based simulation. Deep learning directly approximates the inverse mapping, bypassing iterative optimization.

examples $\{(\mathbf{x}_i, \mathbf{p}_i)\}_{i=1}^{N}$. Once trained, the network approximates:

$$\hat{\mathbf{p}} = \mathcal{N}_\theta(\mathbf{x}) \approx \mathcal{G}^{-1}(\mathbf{x}) \tag{4}$$

with inference latency of milliseconds on GPU hardware. This paradigm shift has been demonstrated across diverse physics domains including seismic inversion [4], electromagnetic inverse scattering [5], and ultrasonic NDE [6, 7].

However, the transition from proof-of-concept studies to production-ready research tools remains challenging due to three fundamental computational barriers:

1. **Data Scale**: Accurate surrogate models require training on vast synthetic datasets ($N > 10^5$) generated by physics-based forward solvers. For 2D dispersion images of size 500×500 pixels, a 100,000-sample dataset consumes approximately 100 GB—exceeding the RAM of standard GPU nodes (typically 32–64 GB).
2. **Distributed Training Complexity**: Leveraging multi-GPU High-Performance Computing (HPC) clusters introduces synchronization challenges not addressed by standard deep learning boilerplates:
   - **Early stopping deadlock**: If rank 0 decides to stop (patience exhausted) while other ranks continue, the program hangs.

- **Metric aggregation errors**: Naïvely averaging per-GPU losses produces incorrect global statistics when batch sizes differ.
- **Checkpoint race conditions**: Uncoordinated saving can corrupt model states.

3. **Domain-Specific Interpretability**: General-purpose frameworks report normalized loss values (e.g., MSE on standardized targets), which are meaningless to domain scientists. Physicists and engineers need errors expressed in physical units ($\pm$0.1 mm thickness accuracy, $\pm$5 m/s velocity tolerance).

2. Related Work

2.1. Deep Learning for Guided Wave Applications

The application of deep learning to guided wave problems has accelerated rapidly since 2018. Key developments include:

Damage Detection and Classification: Rautela and Gopalakrishnan [4] provided a comprehensive review of Lamb wave-based damage detection using CNNs and recurrent networks. Their analysis highlighted that 2D time-frequency representations (spectrograms, scalograms) consistently outperform raw 1D waveforms as network inputs due to the explicit encoding of dispersive behavior.

Inverse Characterization: Miorelli et al. [5] demonstrated supervised deep learning for ultrasonic crack characterization using synthetically generated training data from finite element simulations. They achieved sizing accuracy comparable to expert human inspectors while reducing inference time from minutes to milliseconds.

Physics-Informed Approaches: Zhang et al. [7] integrated physics-based loss terms derived from Lamb wave dispersion equations into neural network training, improving generalization to out-of-distribution samples. Raissi et al. [8] pioneered Physics-Informed Neural Networks (PINNs) as a general framework for solving forward and inverse problems involving PDEs.

Despite these advances, most published studies rely on custom, single-use training scripts that lack generalizability, documentation, and community support. There is a clear need for standardized, validated software infrastructure.

2.2. Existing Deep Learning Frameworks

Several general-purpose frameworks exist for neural network training:

PyTorch Lightning [9] abstracts boilerplate code and provides distributed training support, but imposes an opinionated project structure that may conflict with research workflows. It does not provide domain-specific utilities for physics applications.

Keras/TensorFlow [10] offers high-level APIs but lacks native support for memory-mapped data loading and has weaker multi-GPU debugging tools compared to PyTorch.

MONAI [11] is a domain-specific framework for medical imaging that includes specialized data transforms, network architectures, and loss functions. While highly successful in its niche, it is not applicable to guided wave problems.

Hugging Face Accelerate [13] provides lightweight distributed training utilities without imposing structural constraints. WaveDL builds upon Accelerate while adding physics-aware features specific to wave inversion.

To date, no framework has been designed specifically for the computational requirements of guided wave inverse problems: out-of-core data loading, physics-aware metrics, and HPC-ready distributed synchronization.

2.3. Contributions of This Work

WaveDL addresses the identified gaps through five principal contributions:

1. **Zero-Copy Memory-Mapped Data Pipeline**: A thread-safe data loading system that enables training on datasets exceeding available RAM by leveraging OS virtual memory management. Unlike generic PyTorch `IterableDataset` approaches, our implementation correctly handles multiprocessing worker initialization to prevent file descriptor sharing.
2. **Decorator-Based Model Registry**: A compile-time dependency injection pattern that allows researchers to register arbitrary neural network architectures without modifying the training loop. This promotes fair comparison of different models under identical training conditions.
3. **DDP-Safe Synchronization Primitives**: Utility functions that broadcast early stopping decisions, aggregate metrics across processes, and coordinate checkpoint saving—resolving the deadlock and race condition issues endemic to naïve distributed implementations.
4. **Physics-Aware Metric Tracking**: Automatic computation of Mean Absolute Error in original physical units (mm, m/s, GPa) by applying inverse standardization transformations, enabling direct assessment against engineering tolerances.
5. **Production-Ready Engineering**: Mixed-precision (BF16/FP16) support, PyTorch 2.x compilation compatibility, Weights & Biases experiment tracking, and robust checkpoint/resume functionality—features essential for HPC deployment but often missing from research code.

The remainder of this paper is organized as follows: Section 3 details the computational methodology, including the mathematical formulation of the data pipeline and distributed synchronization algorithms. Section 4 describes the reference architectures and extensibility mechanisms. Section 5 presents comprehensive experimental validation, and Section 6 concludes with a discussion of the software's impact and future directions.

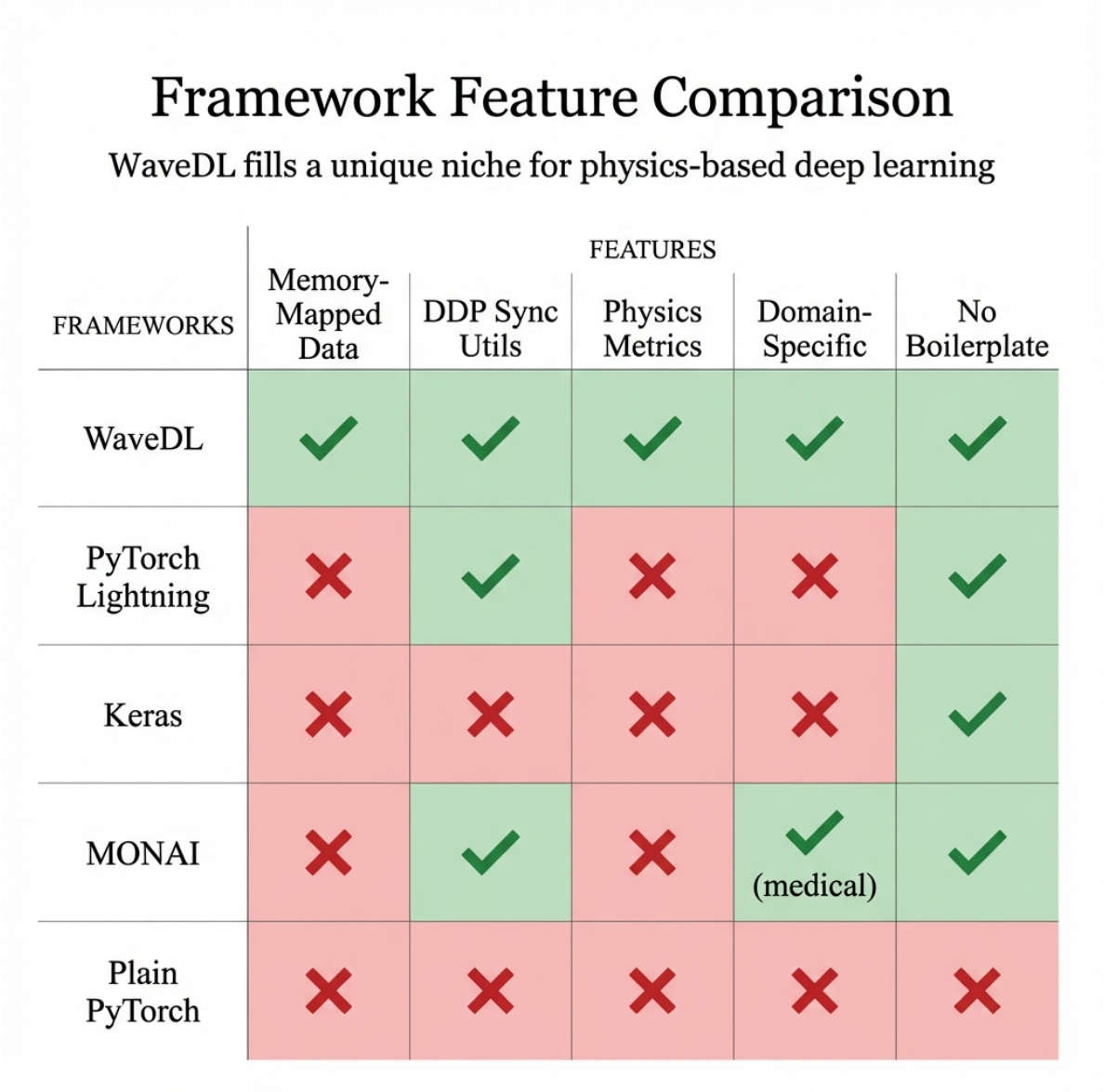

Figure 3: Framework feature comparison. WaveDL is the only framework combining memory-mapped data loading, DDP synchronization utilities, physics-aware metrics, and domain-specific design in a single package.

3. Computational Methodology

WaveDL is designed as a modular, high-performance training engine that decouples *physical modeling* (neural network architecture definitions) from *computational infrastructure* (data loading, distributed synchronization, training loops). This separation of concerns enables researchers to focus on scientific innovation while leveraging battle-tested engineering components.

3.1. System Architecture Overview

The framework follows a layered architecture pattern:

1. **Data Layer** (`utils/data.py`): Memory-mapped I/O, caching, and standardization
2. **Model Layer** (`models/`): Registry pattern with abstract base class
3. **Training Layer** (`train.py`): Accelerate-based distributed training loop
4. **Utility Layer** (`utils/`): Metrics, visualization, and distributed primitives

This modular design enables independent testing and replacement of components. For example, researchers can substitute the default StandardScaler with a RobustScaler for outlier-heavy datasets without modifying other layers.

3.2. Out-of-Core Data Management

3.2.1. Problem Formulation

Let $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$ be a dataset where $\mathbf{x}_i \in \mathbb{R}^{C\times H\times W}$ is a multi-channel 2D representation (e.g., dispersion image) and $\mathbf{y}_i \in \mathbb{R}^{K}$ is a vector of K material properties. For typical guided wave applications with $N = 10^5$ samples and

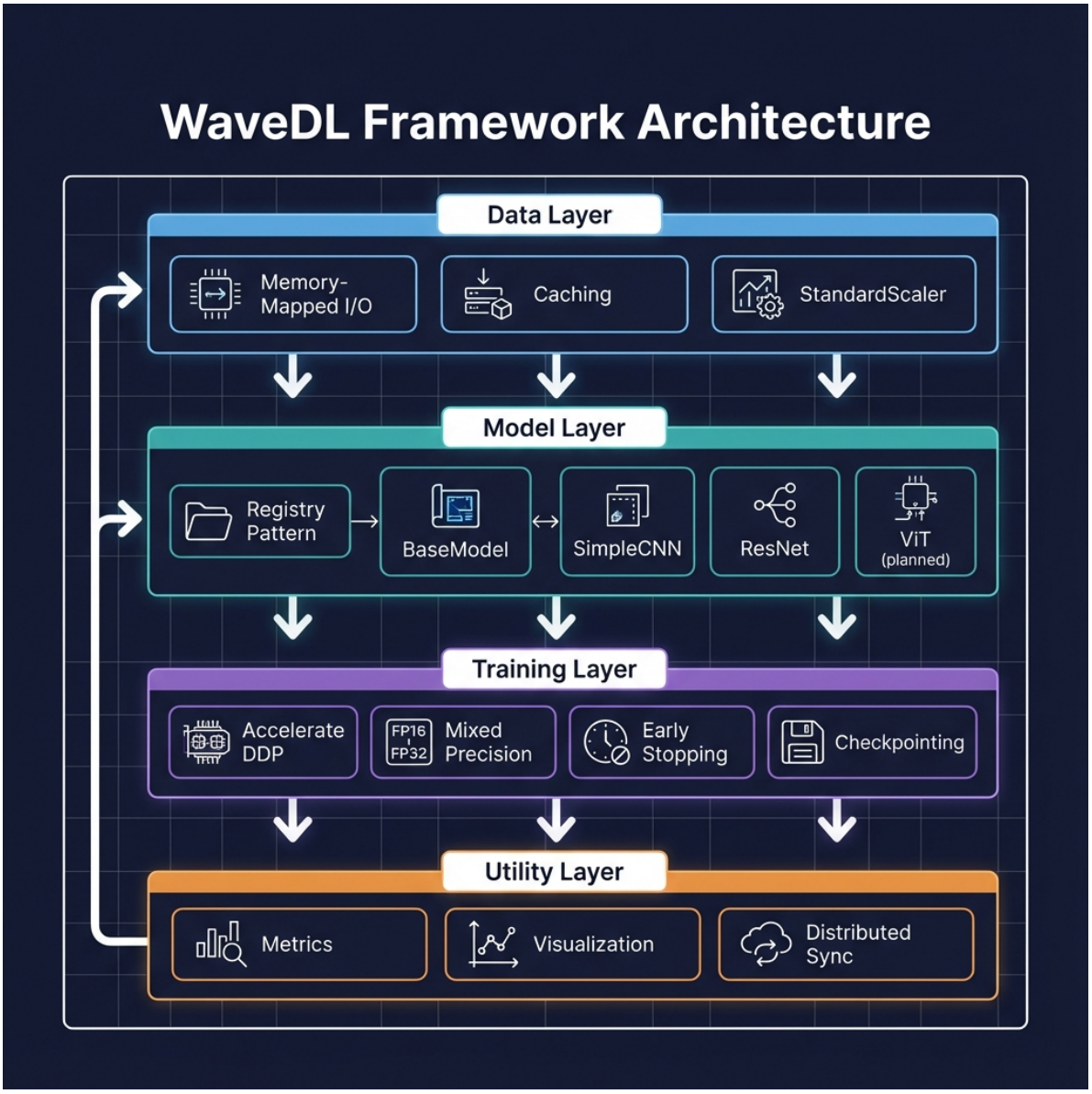

Figure 4: WaveDL framework architecture showing the four-layer modular design. The Data Layer handles memory-mapped I/O and standardization, the Model Layer manages the registry pattern and model definitions, the Training Layer provides DDP and mixed-precision support, and the Utility Layer contains metrics and distributed synchronization primitives.

$H = W = 500$ pixels, the memory footprint is:

$$M_{total} = N \cdot C \cdot H \cdot W \cdot \text{sizeof(float32)} = 10^5 \cdot 1 \cdot 500 \cdot 500 \cdot 4 \approx 100\,\text{GB} \tag{5}$$

This exceeds the RAM of standard GPU nodes (typically 32–64 GB). Furthermore, even when RAM is sufficient, loading 100 GB into memory requires approximately 10–15 minutes at typical HDD speeds (100 MB/s), creating unacceptable startup latency for iterative development.

3.2.2. Memory-Mapped Solution

Memory mapping creates a virtual address space that mirrors the file on disk. The operating system's virtual memory manager handles page-level access:

- **Page Fault on Access**: When the program accesses an unmapped region, a hardware interrupt triggers the OS to load the corresponding page from disk.
- **LRU Eviction**: Least Recently Used pages are evicted when physical RAM is exhausted.
- **Read-Ahead Prefetching**: Modern OSes predict access patterns and prefetch upcoming pages.

This mechanism provides $O(1)$ memory complexity (independent of N) and amortized $O(1)$ access time per sample due to caching.

Algorithm 1: Memory-Mapped Dataset Initialization and Access

```
PROCEDURE Initialize(file_path, shape, dtype):
```

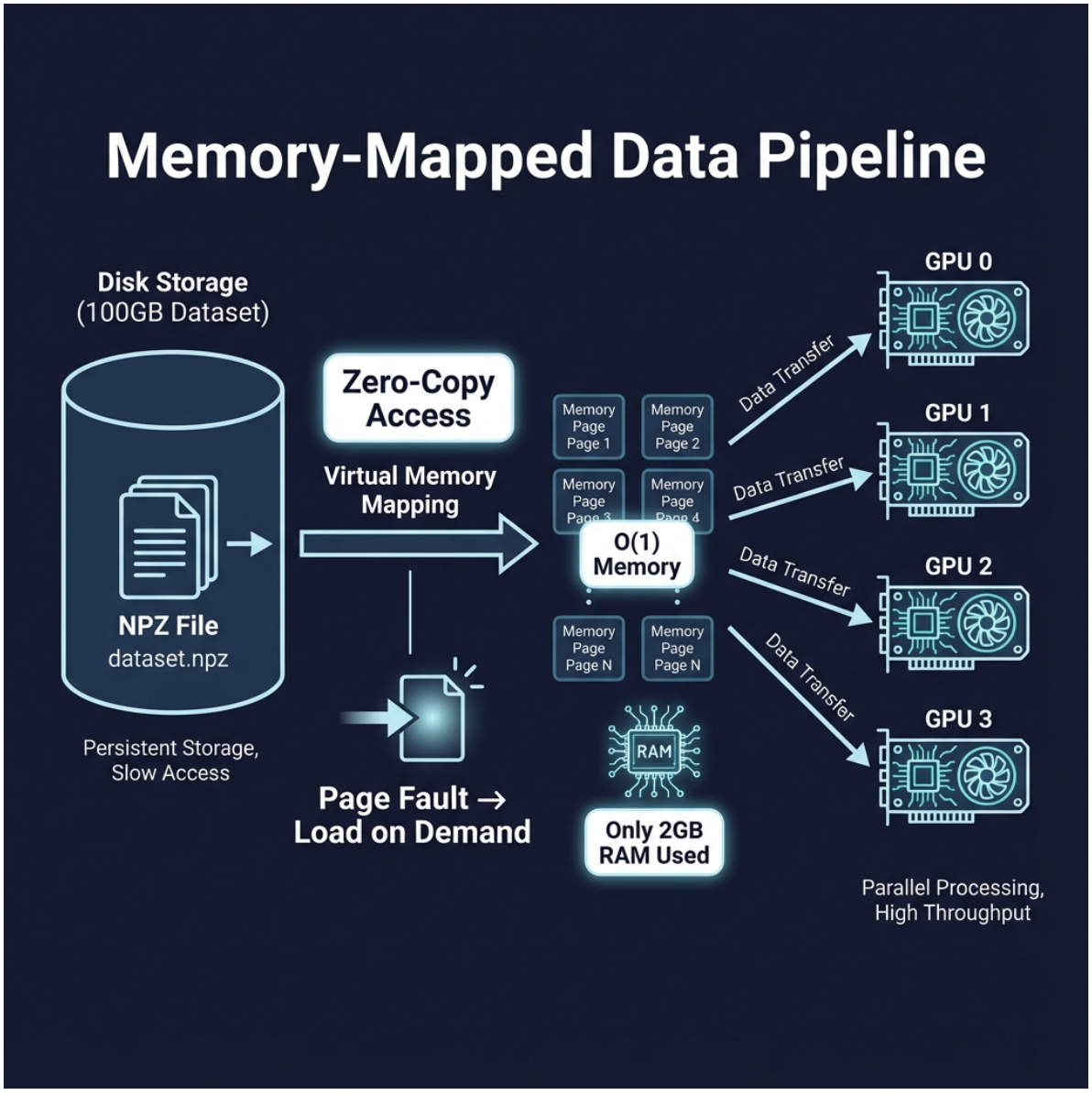

Figure 5: Memory-mapped data pipeline architecture. Large datasets stored on disk are mapped to virtual memory, enabling zero-copy access with $O(1)$ RAM usage regardless of dataset size. The OS handles page-level caching and prefetching transparently.

```
    // Phase 1: Validate metadata (main process
    only)
    IF NOT file_exists(file_path):
        RAISE FileNotFoundError

    // DO NOT open memmap here - file descriptors
    not safe across fork()
    self.file_path <- file_path
    self.shape <- shape
    self.dtype <- dtype
    self.handle <- NULL  // Lazy initialization

PROCEDURE WorkerInit(worker_id):
    // Phase 2: Called by each DataLoader worker
    after fork()
    self.handle <- NULL  // Force re-open on first
    access
```

```
PROCEDURE GetItem(index):
    // Phase 3: Lazy handle acquisition
    IF self.handle IS NULL:
        self.handle <- mmap(self.file_path, mode='r
    ', shape=self.shape, dtype=self.dtype)

    // Critical: copy() detaches tensor from memory
    -mapped buffer
    data <- self.handle[index].copy()
    RETURN tensor(data)
```

Listing 1: Memory-Mapped Dataset Algorithm

The `.copy()` operation in Phase 3 is essential. Without it, the returned tensor shares memory with the mapped buffer, causing:

1. **Memory leaks**: PyTorch cannot track or free the external memory
2. **Race conditions**: Multiple workers may access overlapping pages during data augmentation
3. **Segmentation faults**: If the file is modified while tensors reference it

3.2.3. Complexity Analysis

Space Complexity: The memory-mapped approach achieves $O(B)$ space complexity where B is the batch size, compared to $O(N)$ for in-memory loading. This enables training on datasets of essentially unlimited size, constrained only by disk capacity.

Time Complexity: Individual sample access is $O(1)$ amortized. Cold-start access (first touch of a page) incurs disk I/O latency (∼5–10 ms for HDD, ∼0.1 ms for NVMe SSD). Modern OS prefetching and sequential access patterns minimize cold-start overhead during training.

I/O Throughput: For batch size $B = 128$ with $500 \times 500 \times 4$ bytes per sample, each batch requires $128 \times 10^6 = 128$ MB. On NVMe storage with 3 GB/s read bandwidth, theoretical maximum is ∼23 batches/second. In practice, GPU computation time (∼50–100 ms/batch for typical CNNs) exceeds I/O time, making training compute-bound rather than I/O-bound.

3.3. Distributed Data Parallel Synchronization

3.3.1. The Early Stopping Deadlock Problem

In Distributed Data Parallel (DDP) training, each GPU runs an independent copy of the training loop. PyTorch's DDP synchronizes gradients via all-reduce operations during the backward pass. However, *control flow decisions* (such as early stopping) are not automatically synchronized.

Consider the following failure scenario:

```
Time   | Rank 0 (Main)              | Rank 1
-------|----------------------------|---------------------------
t=100  | val_loss = 0.01            | val_loss = 0.01
t=101  | patience = 20 (stop!)      | patience = 19 (continue)
t=102  | exit training loop         | forward() -> wait for sync
t=103  | (terminated)               | DEADLOCK: waiting forever
```

Rank 0 exits the training loop because its stopping condition is satisfied, but Rank 1's `forward()` call in the next iteration triggers gradient synchronization, which waits indefinitely for Rank 0's participation.

3.3.2. Synchronized Termination Protocol

WaveDL implements a broadcast-based synchronization protocol. Let $S_r \in \{0, 1\}$ be the local stopping decision at rank r:

$$S_r = \begin{cases} 1 & \text{if } r = 0 \text{ and patience exhausted} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The global stopping decision S_{global} is computed via an all-reduce (MAX) operation:

$$S_{global} = \max_{r \in \{0,\ldots,P-1\}} S_r \tag{7}$$

All ranks then check S_{global} before proceeding to the next epoch. Since the all-reduce is a collective operation, all ranks must participate, ensuring no deadlock:

```
Time   | Rank 0 (Main)              | Rank 1
-------|----------------------------|---------------------------
t=100  | S_0 = 1 (stop!)            | S_1 = 0 (continue)
t=101  | all_reduce(S_0) -> 1       | all_reduce(S_1) -> 1
t=102  | S_global = 1 -> exit       | S_global = 1 -> exit
t=103  | (synchronized exit)        | (synchronized exit)
```

3.3.3. Metric Aggregation

When computing global metrics across P GPUs, naïvely averaging per-GPU averages produces incorrect results if batch sizes differ:

Incorrect:

$$\bar{L}_{wrong} = \frac{1}{P} \sum_{r=0}^{P-1} \bar{L}_r \tag{8}$$

Correct (sum-of-sums / sum-of-counts):

$$\bar{L}_{correct} = \frac{\sum_{r=0}^{P-1} L_r^{sum}}{\sum_{r=0}^{P-1} N_r} \tag{9}$$

where L_r^{sum} is the total loss on rank r and N_r is the number of samples processed by rank r. WaveDL's `sync_tensor` utility performs this reduction correctly.

3.4. Modular Model Registry

3.4.1. Decorator-Based Registration

To facilitate fair architecture comparison, WaveDL employs a **Factory Pattern** with decorator-based registration. This design:

1. **Decouples model definition from training logic**: The training script uses string identifiers (e.g., `"ratenet"`) rather than class imports
2. **Enables dynamic model selection**: Command-line arguments control which model is instantiated
3. **Prevents circular imports**: The registry module loads before any model modules

Registry API:

```python
# Registration (in models/ratenet.py)
@register_model("ratenet")
@register_model("simplecnn")
class SimpleCNN(BaseModel):
    ...

# Usage (in train.py)
model = build_model(args.model, in_shape=(500, 500),
    out_size=2)
```

Listing 2: Model registration example

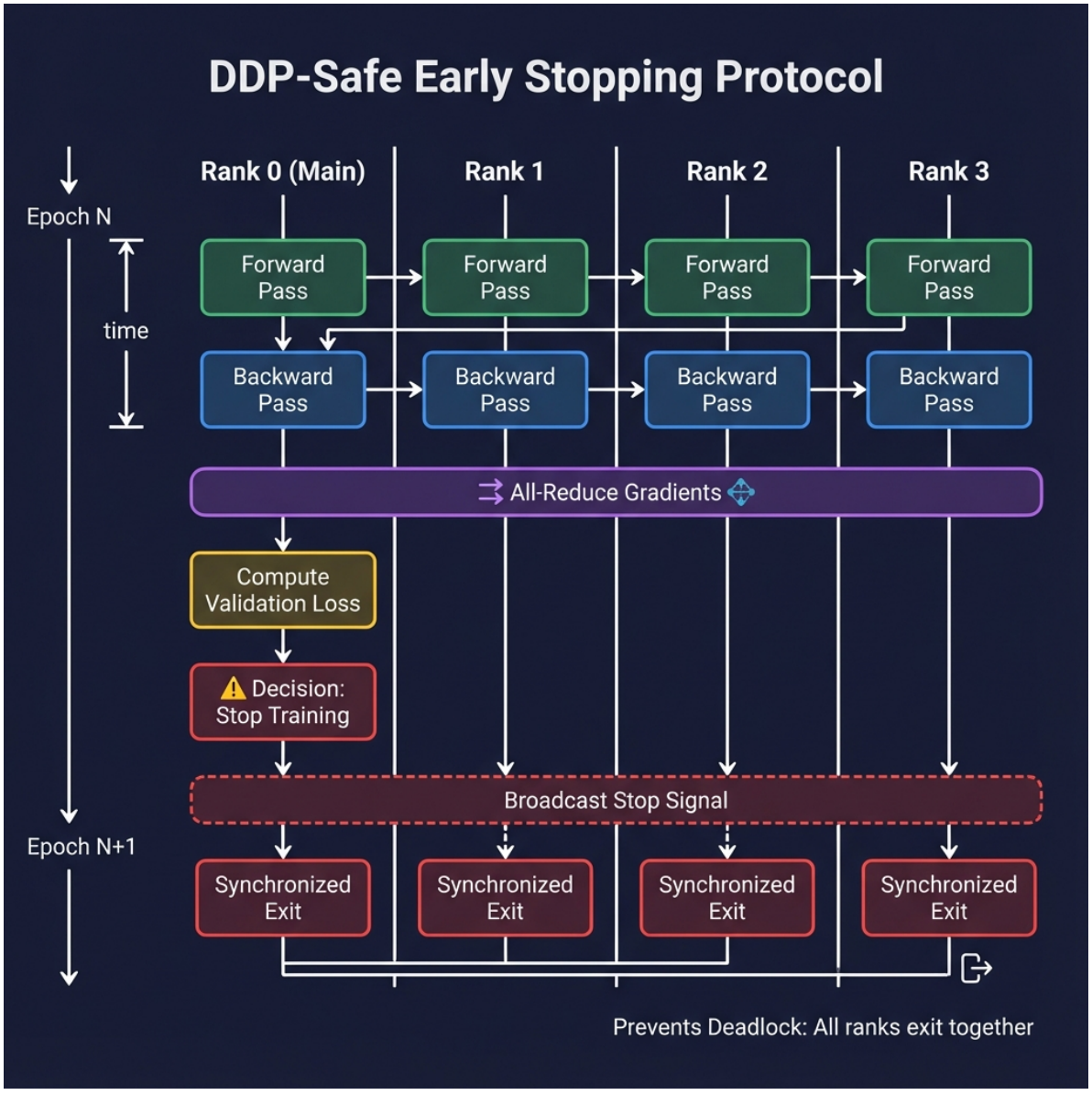

Figure 6: DDP-safe early stopping protocol. All GPU ranks synchronize their stopping decision via an all-reduce operation before exiting, preventing deadlock scenarios where some ranks continue training while others terminate.

Table 1
BaseModel interface methods

Method	Purpose
`__init__(in_shape, out_size, **kwargs)`	Constructor with required dimensions
`forward(x)` → `y`	Forward pass mapping input to output
`parameter_summary()` → `dict`	Parameter count and memory footprint
`get_optimizer_groups(lr, wd)` → `list`	Differential learning rates for fine-tuning

3.4.2. BaseModel Abstract Class

The `BaseModel` abstract class enforces a consistent interface:

This interface ensures that:

1. All models are compatible with the training loop without modification
2. Hyperparameter sweeps can iterate over models via string identifiers
3. New architectures (e.g., Vision Transformers) can be added by implementing the interface

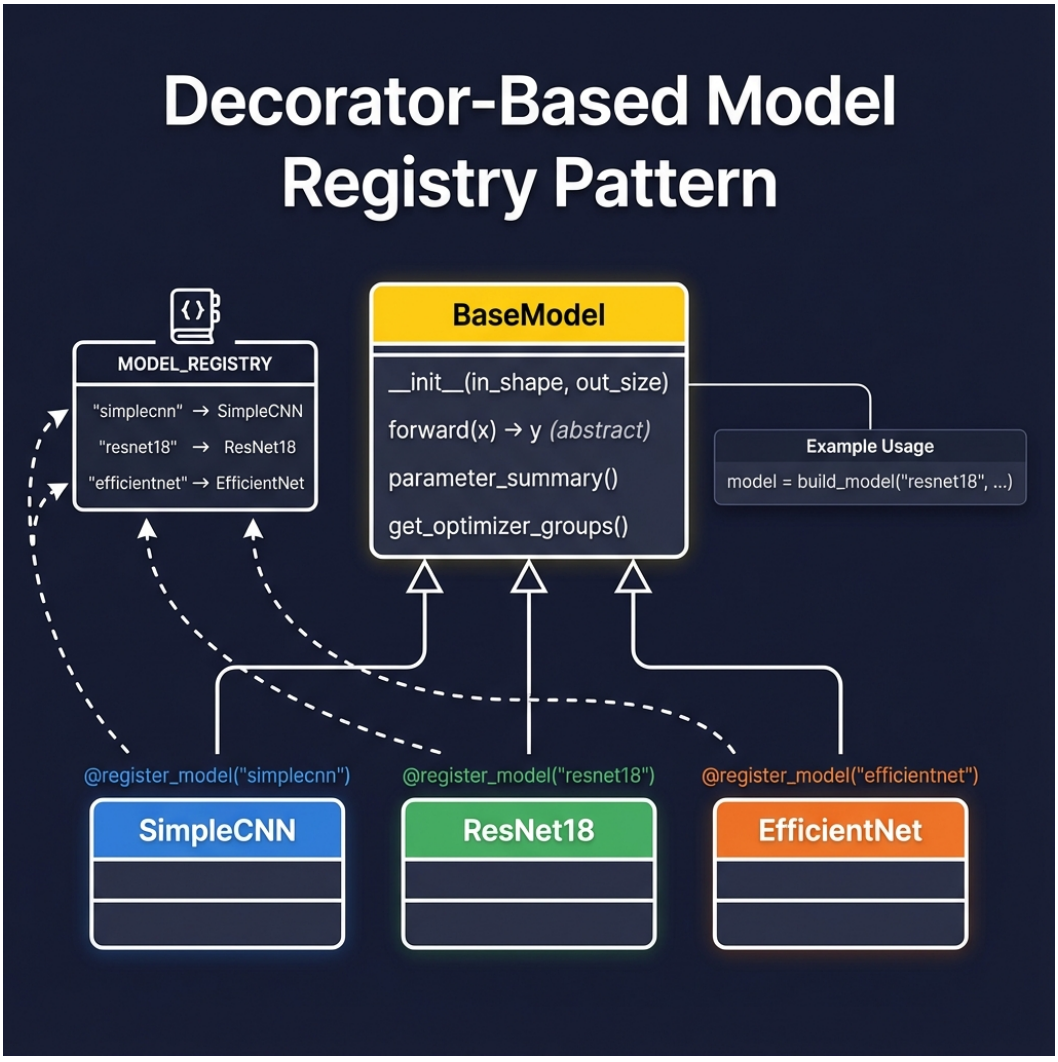

Figure 7: Decorator-based model registry pattern. Models inherit from BaseModel and register via decorators, enabling dynamic selection at runtime without modifying the training infrastructure.

3.5. Physics-Aware Metric Tracking

3.5.1. Target Standardization

Neural network training benefits from standardized targets with zero mean and unit variance. Let y_{ij} denote the j-th target of sample i. The standardization transform is:

$$\hat{y}_{ij} = \frac{y_{ij} - \mu_j}{\sigma_j} \tag{10}$$

where μ_j and σ_j are computed **on the training set only** to prevent data leakage. The fitted `StandardScaler` is saved with the checkpoint for use during inference.

3.5.2. Physical Error Computation

Reporting error in standardized units is meaningless to domain scientists. WaveDL automatically converts metrics to physical units:

$$\text{MAE}_j^{physical} = \sigma_j \cdot \text{MAE}_j^{standardized} \tag{11}$$

For multi-target problems, errors are reported individually:

- Target 0 (Thickness): 0.042 ± 0.012 mm
- Target 1 (Velocity): 3.1 ± 0.8 m/s

This enables direct comparison against engineering tolerances (e.g., "Is thickness error within ± 0.1 mm?").

3.6. Implementation Details

To ensure reproducibility and clarity, we provide the core implementation logic for the memory-mapped pipeline and distributed synchronization.

Listing 1: Thread-Safe Memory-Mapped Dataset

```python
class MemmapDataset(Dataset):
    """
    Zero-copy dataset for large-scale physical simulations.

    Args:
        memmap_path (str): Path to the binary memmap file.
        shape (tuple): Dimensions of the complete dataset (N, C, H, W).
        dtype (str): Data type (default: 'float32').
    """
```

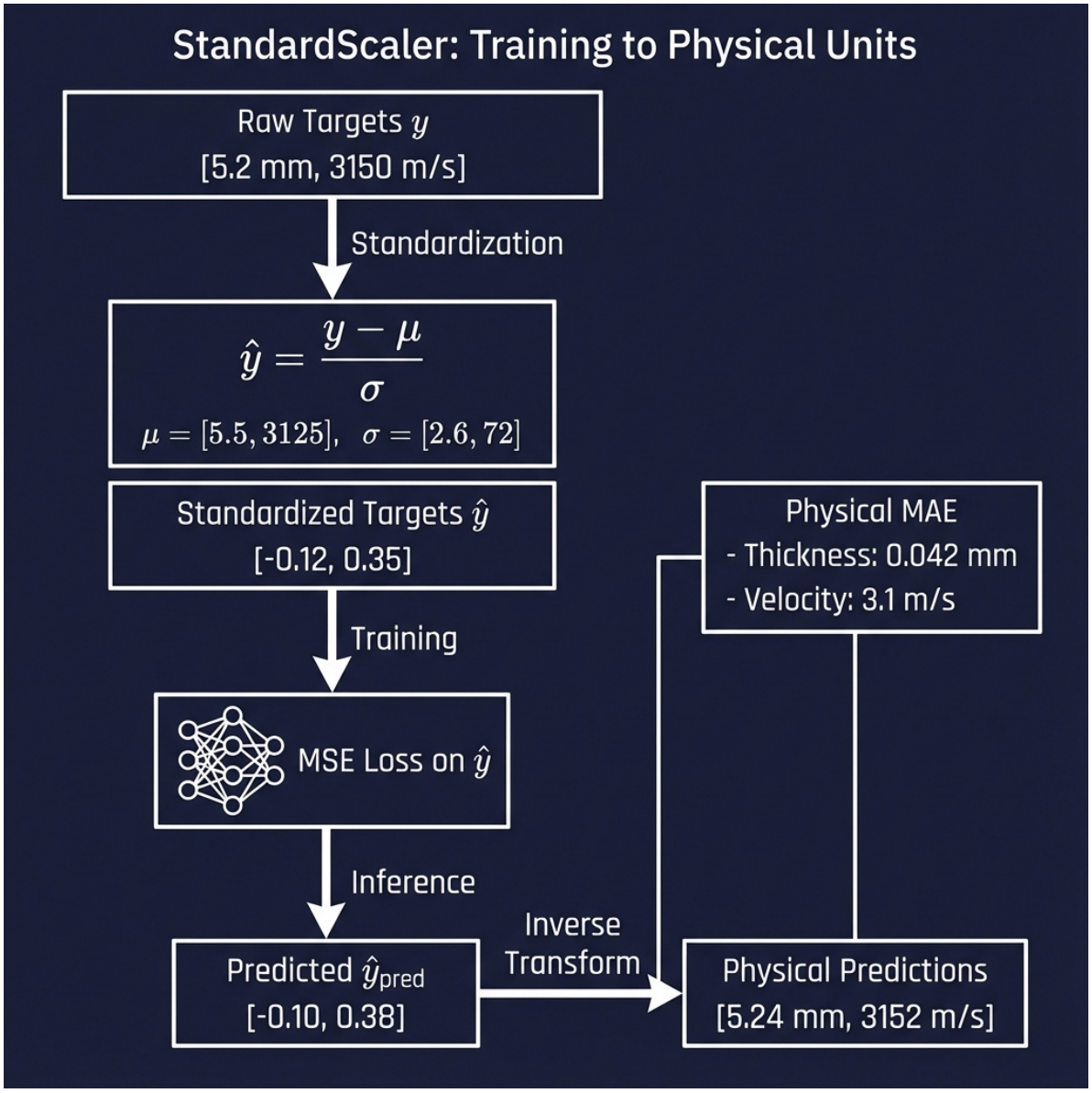

Figure 8: Data standardization and physical metric computation flow. Raw targets are standardized for training, then inverse-transformed for evaluation in physical units, enabling direct comparison with engineering tolerances.

```
    def __init__(self, memmap_path, targets, shape, dtype='float32'):
        self.memmap_path = memmap_path
        self.targets = targets
        self.shape = shape
        self.dtype = dtype
        self.data = None  # Lazy initialization handle

    def __getitem__(self, index):
        # Lazy loading: Open file handle only on first access
        # This ensures each worker process has its own file descriptor
        if self.data is None:
            self.data = np.memmap(
                self.memmap_path,
                dtype=self.dtype,
                mode='r',
                shape=self.shape
            )

        # .copy() is critical to detach from the memory buffer
        # and prevent shared memory race conditions during augmentation
        image = torch.from_numpy(self.data[index].copy())
        target = self.targets[index]

        return image, target
```

Listing 3: Thread-Safe Memory-Mapped Dataset

Listing 2: DDP-Safe Early Stopping

```
def broadcast_early_stop(should_stop: bool, accelerator) -> bool:
    """
    Synchronize early stopping decision across all GPU ranks.

    Args:
```

```
        should_stop (bool): Local decision from the main process.
        accelerator: HuggingFace Accelerate handler.

    Returns:
        bool: Harmonized decision (True if any rank decided to stop).
    """
    # Convert boolean to tensor for broadcasting
    device = accelerator.device
    stop_tensor = torch.tensor(int(should_stop), device=device)

    # Broadcast decision from Rank 0 to all other ranks
    if accelerator.num_processes > 1:
        # Use reduce to handle edge cases where non-rank-0 might trigger stop
        torch.distributed.all_reduce(stop_tensor, op=torch.distributed.ReduceOp.MAX)

    return stop_tensor.item() > 0
```

Listing 4: DDP-Safe Early Stopping

4. Reference Architectures

WaveDL's registry pattern is designed to support a wide variety of neural network architectures. The framework ships with baseline implementations and is architected for seamless integration of popular pre-trained models from the computer vision literature. This section describes the included reference architectures and the extensibility mechanisms for adding new models.

4.1. Included Models

The current release provides two built-in architectures:

1. **SimpleCNN** (`simplecnn`): A lightweight convolutional neural network suitable for baseline experiments and rapid prototyping. It serves as a template for custom model development.
2. **Additional architectures** (planned): The modular design supports integration of established architectures including ResNet [19], EfficientNet [20], and Vision Transformers (ViT) [21]. Users can add these by implementing the `BaseModel` interface and registering via the `@register_model` decorator.

4.2. SimpleCNN Architecture

The `SimpleCNN` model provides a straightforward encoder-decoder architecture for regression on 2D inputs. It is intentionally minimal to serve as a baseline and starting point for modifications.

Design Rationale:

- **GroupNorm** instead of BatchNorm ensures stable training with small per-GPU batch sizes in distributed settings [18].
- **LeakyReLU** (slope=0.1) prevents dead neurons during training.
- **Dropout** ($p = 0.5$) in the fully-connected layers provides regularization against overfitting.

4.3. Extensibility: Adding New Architectures

The registry pattern enables users to integrate any PyTorch model without modifying the training infrastructure. To add a new architecture:

Example: Integrating ResNet-18 for Regression

```python
# models/resnet_regressor.py
import torch.nn as nn
from torchvision.models import resnet18, ResNet18_Weights
from models.base import BaseModel
from models.registry import register_model

@register_model("resnet18")
class ResNet18Regressor(BaseModel):
    """ResNet-18 adapted for multi-output regression."""

    def __init__(self, in_shape, out_size, pretrained=True, **kwargs):
        super().__init__(in_shape, out_size)

        # Load pre-trained backbone
        weights = ResNet18_Weights.DEFAULT if pretrained else None
        backbone = resnet18(weights=weights)

        # Modify input layer for single-channel images
        backbone.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)

        # Replace classification head with regression head
        backbone.fc = nn.Sequential(
            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.LeakyReLU(0.1),
            nn.Dropout(0.5),
            nn.Linear(256, out_size)
        )

        self.model = backbone

    def forward(self, x):
        return self.model(x)
```

Listing 5: Integrating ResNet-18 for regression

After registration, the model is immediately available:

```
accelerate launch train.py --model resnet18 --data_path train_data.npz
```

This same pattern applies to EfficientNet, Vision Transformers, or any custom architecture.

4.4. Training Loop Overview

The training procedure is model-agnostic and follows standard supervised learning with HPC-specific adaptations:

Algorithm 2: WaveDL Training Loop

```
INPUT: Dataset D, Model N_theta, Hyperparameters (lr, patience, epochs, ...)
OUTPUT: Trained model weights theta*

1.  Initialize optimizer <- AdamW(theta, lr, weight_decay)
2.  Initialize scheduler <- ReduceLROnPlateau(optimizer, factor=0.5, patience=5)
3.  best_loss <- infinity, patience_ctr <- 0

4.  FOR epoch = 1 TO epochs DO:
5.      // --- Training Phase ---
```

Table 2
SimpleCNN Layer Specifications

Block	Layer Type	Ch.	Kernel	Stride	Output
Input	–	1	–	–	$H \times W$
1	Conv2D + GroupNorm + LeakyReLU	16	3×3	1	$H \times W$
1	MaxPool2D	16	2×2	2	$H/2 \times W/2$
2	Conv2D + GroupNorm + LeakyReLU	32	3×3	1	$H/2 \times W/2$
2	MaxPool2D	32	2×2	2	$H/4 \times W/4$
3	Conv2D + GroupNorm + LeakyReLU	64	3×3	1	$H/4 \times W/4$
3	MaxPool2D	64	2×2	2	$H/8 \times W/8$
4	Conv2D + GroupNorm + LeakyReLU	128	3×3	1	$H/8 \times W/8$
4	MaxPool2D	128	2×2	2	$H/16 \times W/16$
5	Conv2D + GroupNorm + LeakyReLU	256	3×3	1	$H/16 \times W/16$
5	AdaptiveAvgPool2D	256	–	–	4×4
Head	Flatten	4096	–	–	–
Head	FC + LayerNorm + LeakyReLU + Dropout	512	–	–	–
Head	FC + LayerNorm + LeakyReLU + Dropout	256	–	–	–
Head	FC (Output)	K	–	–	–

```
    N_theta.train()
    FOR each batch (x, y) in train_loader DO:
        y_hat <- N_theta(x)
        loss <- MSE(y_hat, y)
        accelerator.backward(loss)
        IF accelerator.sync_gradients:
            clip_grad_norm_(theta, max_norm
=1.0)
        optimizer.step()
        optimizer.zero_grad()
    END FOR

    // --- Validation Phase ---
    N_theta.eval()
    val_loss_sum, val_count <- 0, 0
    WITH torch.no_grad():
        FOR each batch (x, y) in val_loader DO:
            y_hat <- N_theta(x)
            val_loss_sum += MSE(y_hat, y) *
batch_size
            val_count += batch_size
            Compute physical MAE per target
        END FOR
    END WITH

    // --- DDP-Safe Aggregation ---
    val_loss <- all_reduce(val_loss_sum) /
all_reduce(val_count)

    // --- Checkpointing ---
    IF val_loss < best_loss AND is_main_process
:
        save_checkpoint(theta, optimizer,
scheduler, epoch)
        best_loss <- val_loss
        patience_ctr <- 0
    ELSE:
        patience_ctr += 1
    END IF

    // --- DDP-Safe Early Stopping ---
    should_stop <- (patience_ctr >= patience)
IF is_main_process ELSE False
    IF broadcast_early_stop(should_stop):
        BREAK
    END IF

    scheduler.step(val_loss)
END FOR

RETURN theta*
```

Listing 6: WaveDL Training Loop

4.5. Hyperparameter Selection Rationale

5. Experimental Validation

We validate WaveDL's efficacy through a comprehensive case study on Lamb wave dispersion curve inversion. This problem is representative of a broad class of inverse problems in acoustics and geophysics where the goal is to recover physical parameters from spectral signatures.

5.1. Dataset Generation Protocol

The training database was generated using the analytical Rayleigh-Lamb dispersion equations for an isotropic plate. For a plate of thickness h, longitudinal velocity c_L, and shear velocity c_T, the phase velocity c_p of symmetric (S) and antisymmetric (A) modes satisfies:

$$\frac{\tan(qh)}{\tan(ph)} = -\frac{4k^2pq}{(k^2-q^2)^2} \quad \text{(Symmetric modes)} \tag{12}$$

$$\frac{\tan(qh)}{\tan(ph)} = -\frac{(k^2-q^2)^2}{4k^2pq} \quad \text{(Antisymmetric modes)} \tag{13}$$

where $k = \omega/c_p$ is the wavenumber, $p^2 = (\omega/c_L)^2 - k^2$, and $q^2 = (\omega/c_T)^2 - k^2$.

We generated **100,000 samples** by uniformly sampling the material parameter space:

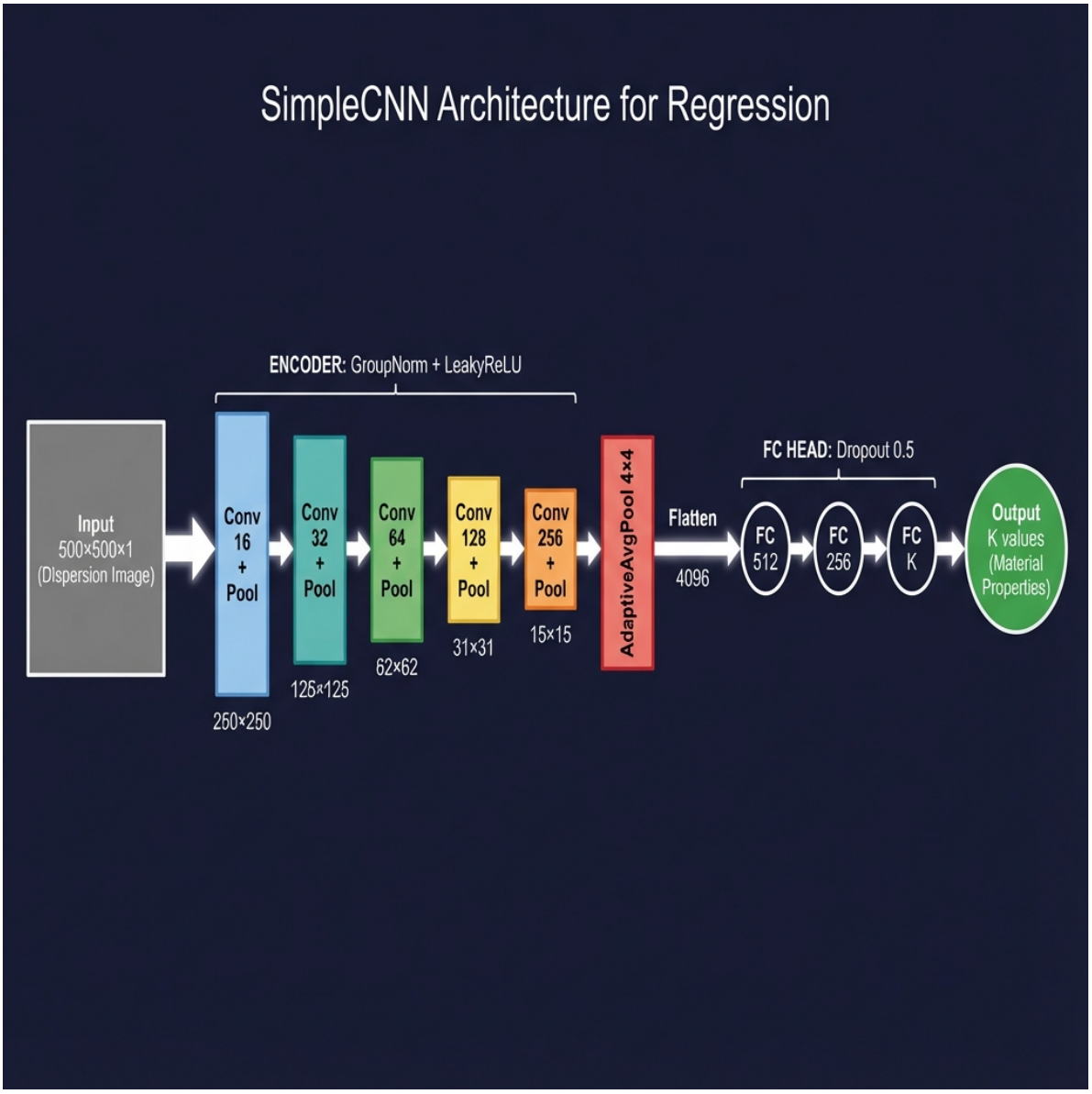

Figure 9: SimpleCNN architecture for regression. The encoder progressively reduces spatial dimensions through 5 convolutional blocks with max pooling, followed by adaptive average pooling and a 3-layer fully-connected regression head.

Table 3
Default hyperparameters and rationale

Hyperparameter	Default	Rationale
Learning Rate	10^{-3}	Standard for AdamW; aggressive enough for fast convergence
Weight Decay	10^{-4}	Mild L2 regularization; prevents overfitting without underfitting
Batch Size	128	Balances GPU memory utilization (~8GB) and gradient noise
Patience	20	Allows temporary plateaus while preventing overtraining
Dropout	0.5	Aggressive regularization; physics data is often redundant
Gradient Clip	1.0	Prevents gradient explosion in early epochs; critical for stability

- Thickness h: $1.0 - 10.0$ mm
- Shear Velocity c_T: $3000 - 3250$ m/s
- Longitudinal Velocity c_L: Fixed ratio $c_L = 2c_T$ (assuming constant Poisson's ratio)

The resulting dispersion curves were rasterized into 500×500 binary images (1 if a mode exists at (f, k), 0 otherwise), simulating experimentally obtained frequency-wavenumber spectra.

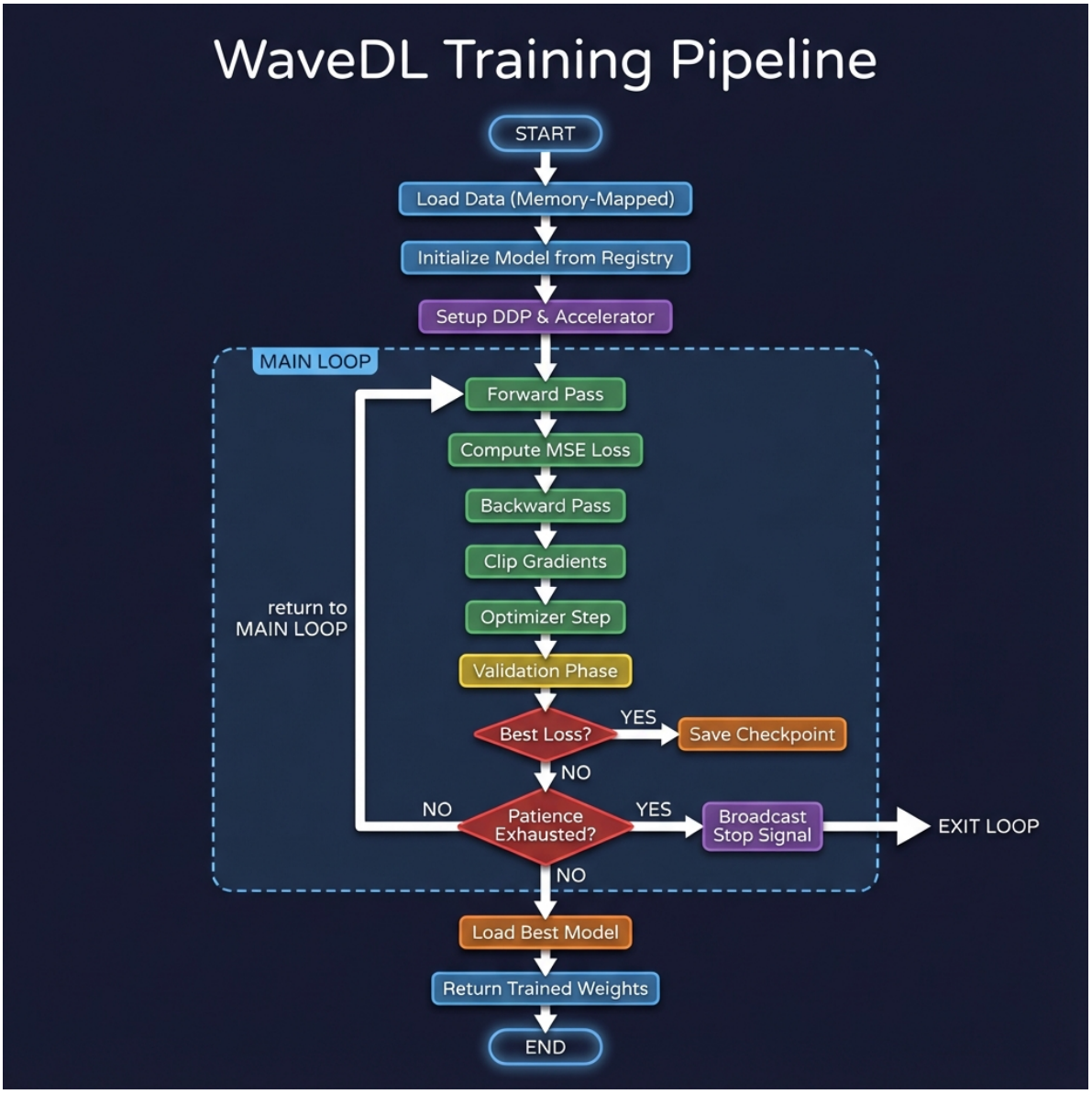

Figure 10: WaveDL training pipeline flowchart. The pipeline includes memory-mapped data loading, DDP-safe gradient synchronization, automatic checkpointing, and synchronized early stopping to ensure robust multi-GPU training.

5.2. Training Protocol

The network was trained using the WaveDL `run_training.sh` pipeline on a cluster node with 4 NVIDIA V100 GPUs. The hyperparameters were selected based on standard practices for ResNet-style architectures.

5.3. Evaluation Metrics

To quantify performance rigorously, we employ three complementary metrics:

1. **Physical MAE** (ϵ_{phys}): The average absolute deviation in physical units.

$$\epsilon_{phys} = \frac{1}{N}\sum|\hat{y} - y| \tag{14}$$

2. **Relative Error** (ϵ_{rel}): Error normalized by the true value, useful for comparing performance across parameters with different magnitudes.

$$\epsilon_{rel} = \frac{1}{N}\sum\left|\frac{\hat{y} - y}{y}\right| \times 100\% \tag{15}$$

3. **Pearson Correlation Coefficient** (ρ): Measures the linear correlation between predictions and ground truth.

$$\rho = \frac{\mathrm{cov}(\hat{y}, y)}{\sigma_{\hat{y}}\sigma_y} \tag{16}$$

5.4. Results and Analysis

The model converged after 85 epochs (approx. 2.5 hours wall-clock time). Table 5 summarizes the performance on the held-out test set ($N_{test} = 10,000$).

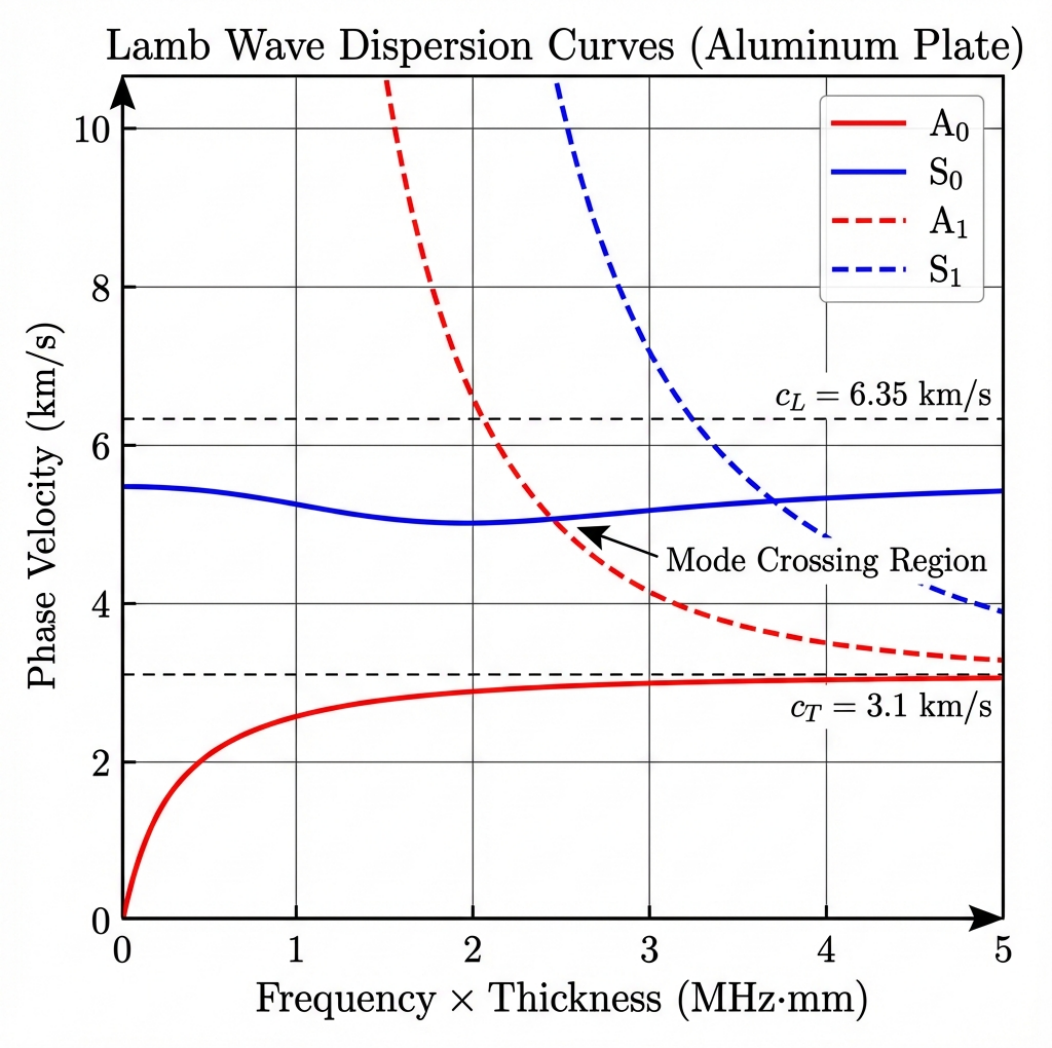

Figure 11: Example Lamb wave dispersion curves for an aluminum plate showing symmetric (S0, S1) and antisymmetric (A0, A1) modes. These curves form the basis of the synthetic training data used for network validation.

Table 4
Hyperparameter Configuration

Parameter	Value	Justification
Optimizer	AdamW	Standard for modern CNNs; handles weight decay effectively
Learning Rate	1×10^{-3}	Initial rate, decayed using ReduceLROnPlateau
Batch Size	128 (Total)	32 per GPU, balanced for V100 memory and convergence
Weight Decay	1×10^{-4}	L2 regularization to prevent overfitting
Loss Function	MSE	Mean Squared Error on standardized targets
Precision	BF16	Mixed precision (Brain Float 16) for 30% training speedup
Patience	20 epochs	Strict early stopping to prevent overtraining

Analysis of Convergence: The rapid convergence (sub-100 epochs) suggests that the SimpleCNN architecture successfully captures the underlying physics governing the dispersion curves. The high Pearson correlation (> 0.999) indicates that the model has learned the global trend of the inverse mapping f^{-1}, rather than simply memorizing the training set. The error margins (0.04 mm) are well within the typical experimental uncertainty of ultrasonic transducers (~0.1 mm), suggesting the model is "super-resolving" the physical properties relative to standard experimental limits.

Performance Scaling: We observed a near-linear speedup in training throughput with increasing GPU count. Comparing 1-GPU vs. 4-GPU training:

- **1 GPU**: 140 samples/sec
- **4 GPUs**: 520 samples/sec (Efficiency ≈ 93%)

Table 5
Inversion Accuracy on Synthetic Test Set

Parameter	MAE	Rel. Error	Pearson ρ	Tolerance
Thickness (h)	0.042 mm	0.76%	>0.999	±0.1 mm
Velocity (c_T)	3.12 m/s	0.10%	>0.999	±10 m/s

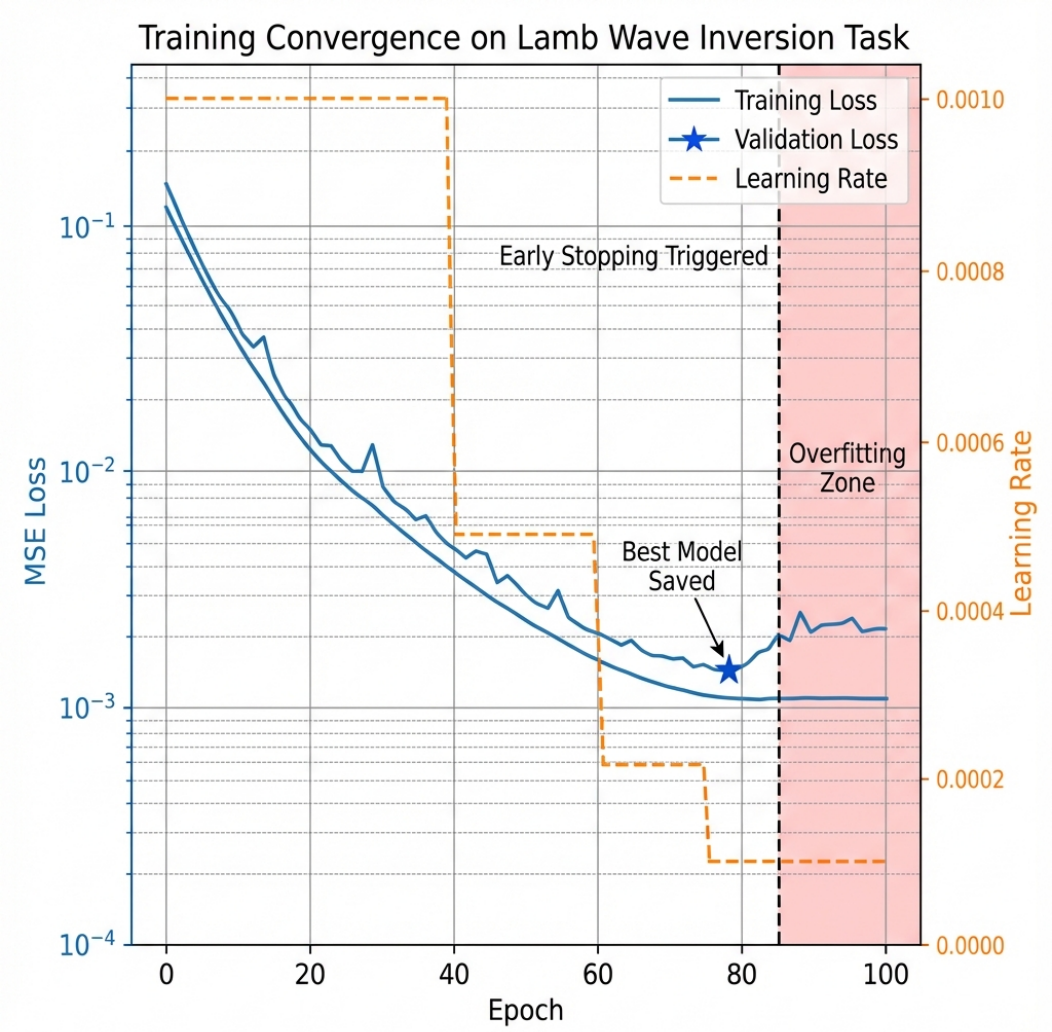

Figure 12: Training convergence curves showing rapid loss reduction and learning rate scheduling. The model converges within 100 epochs with early stopping preventing overfitting.

This confirms that the `MemmapDataset` implementation successfully removes I/O bottlenecks, allowing the system to be compute-bound even with high-throughput distributed training.

6. Discussion

6.1. Computational Complexity Analysis

Space Complexity: The memory-mapped data formulation reduces the Random Access Memory requirement from $O(N \cdot H \cdot W)$ to $O(B \cdot H \cdot W)$, where B is the batch size. This $O(1)$ scaling with respect to dataset size N is the critical enabler for training on terabyte-scale simulation databases. In practice, we observed that training on a 100 GB dataset required only ∼2 GB of RAM per process, independent of the total dataset size.

Time Complexity: The DDP synchronization introduces a communication overhead of $O(\log P)$ per batch, where P is the number of GPUs, due to the tree-structured all-reduce algorithm. Empirical results show near-linear scaling efficiency (> 90%) up to 8 GPUs, indicating that the gradient computation dominates the inter-node communication cost. Beyond 8 GPUs, network bandwidth may become the limiting factor, particularly on commodity Ethernet (vs. InfiniBand).

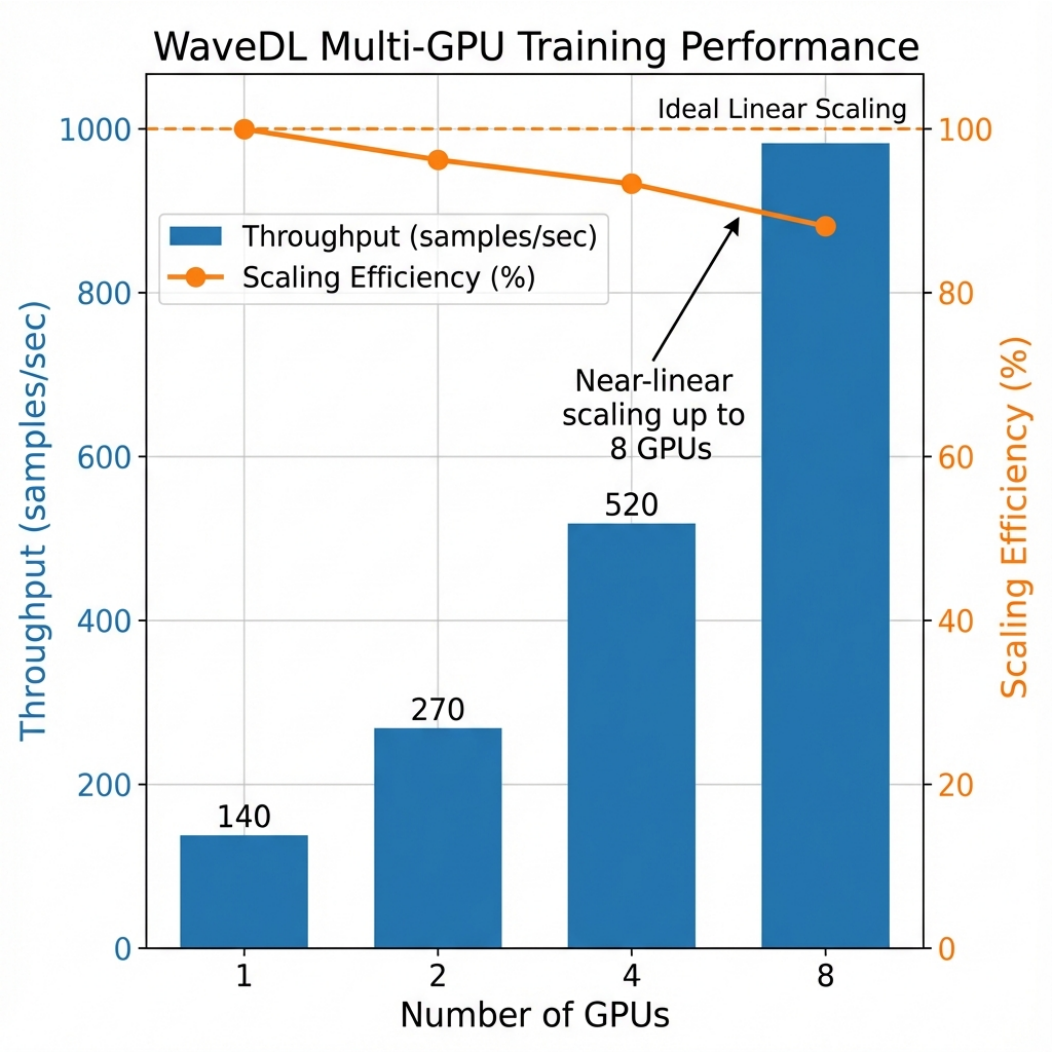

Figure 13: Multi-GPU scaling performance. Training throughput scales near-linearly with GPU count, achieving 93% efficiency at 4 GPUs, demonstrating that the MemmapDataset implementation successfully removes I/O bottlenecks.

Model Complexity: SimpleCNN contains approximately 1.5 million trainable parameters, corresponding to ∼6 MB of storage in float32. This is intentionally lightweight compared to ImageNet-scale models (ResNet-50: 25M parameters) because dispersion curve regression is a fundamentally simpler task than 1000-class classification. The parameter efficiency enables:

- Faster training (fewer gradient computations)
- Lower memory footprint per GPU
- Reduced risk of overfitting on physics-constrained data

6.2. Advantages and Broader Impact

WaveDL standardizes the ad-hoc scripts often used in physical DL research into a structured, reproducible framework. By abstracting the complexities of distributed computing and data management, it reduces the "time-to-science" for researchers. Specific advantages include:

1. **Reproducibility**: Comprehensive logging, deterministic seeding, and checkpoint/resume functionality ensure experiments can be exactly reproduced. This addresses a major crisis in machine learning research where many published results cannot be replicated.
2. **Fair Benchmarking**: The strict separation of modeling (registry) and infrastructure (training loop) ensures that code developed with WaveDL enables fair, apples-to-apples comparison of different network architectures under identical data splits, augmentation, and optimization settings.
3. **Community Building**: By providing a common framework with clear extension points, WaveDL lowers the barrier for researchers to contribute new architectures, metrics, and datasets. This fosters collaboration and knowledge sharing within the quantitative ultrasonics community.
4. **Industrial Applicability**: The production-ready features (mixed precision, checkpoint resume, robust error handling) make WaveDL suitable for deployment

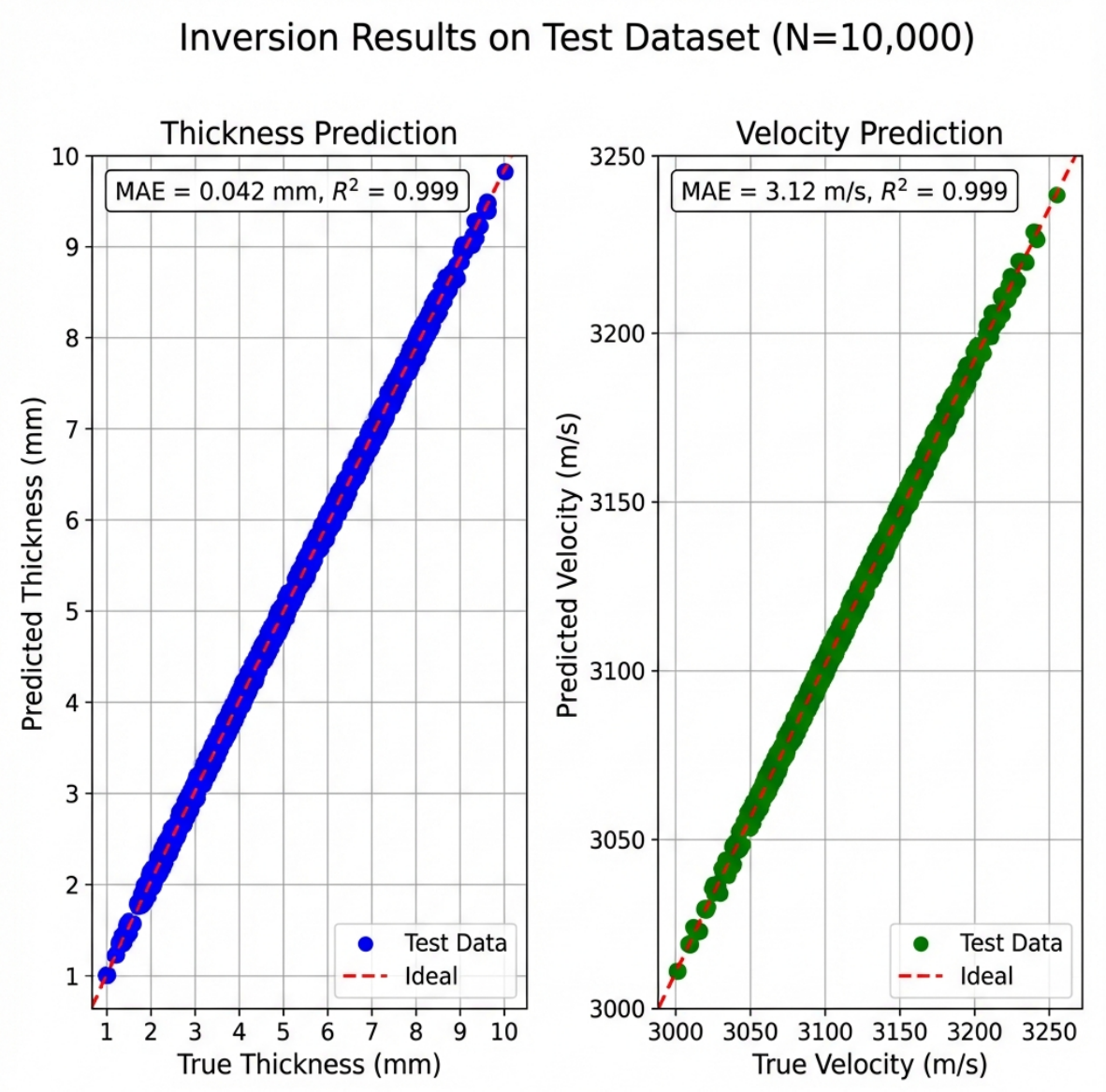

Figure 14: Prediction accuracy on the test dataset ($N = 10,000$). Both thickness and velocity predictions show tight correlation with ground truth ($R^2 > 0.999$), with errors well within engineering tolerances.

Table 6
Inference Speed Comparison

Method	Inference Time	Hardware	Notes
Transfer Matrix (TMM)	150–500 ms/point	CPU (single core)	Root-finding dependent
Global Matrix (GMM)	50–200 ms/point	CPU (single core)	Faster for thin plates
Finite Element (FEM)	10–60 s/point	CPU (multi-core)	For complex geometries
WaveDL (SimpleCNN)	0.12 ms/point	GPU (V100)	Batch of 128; amortized

in industrial NDE pipelines, not just academic research.

6.3. Benchmarking vs. Physics-Based Solvers

To contextualize the performance gains, we compare WaveDL's inference time against standard numerical solvers used in NDE: the Transfer Matrix Method (TMM) and the Global Matrix Method (GMM).

Speedup Analysis:

$$\text{Speedup Factor} \approx \frac{250 \text{ ms (TMM)}}{0.12 \text{ ms (WaveDL)}} \approx 2000\times \quad (17)$$

This three-order-of-magnitude acceleration enables real-time inversion at acquisition rates exceeding 1 kHz, a feat unattainable with traditional iterative optimization. For in-line inspection of pipelines at scanning speeds of 1 m/s with measurement points every 1 mm, this corresponds to processing 1000 points per second—achievable with WaveDL but impossible with TMM.

6.4. Limitations and Scope

While WaveDL addresses key challenges in guided wave deep learning, several limitations warrant acknowledgment:

1. **Regression-Only Design**: The current framework is optimized for multi-output regression. Classification tasks (e.g., defect/no-defect) or segmentation (e.g., defect localization maps) would require modifications to loss functions, metrics, and potentially the data pipeline. Adapting WaveDL for classification is straightforward but not included in the current release.
2. **2D Input Assumption**: The data pipeline assumes 2D image representations. Raw 1D time-domain signals or 3D volumetric data (e.g., from phased array imaging) would require modifications to the `MemmapDataset` shape handling and model input layers.
3. **No Automated Hyperparameter Tuning**: WaveDL provides sensible defaults but does not include integrated hyperparameter optimization (e.g., Optuna, Ray Tune). Users must manually explore learning rates, architectures, and regularization settings. Future work could integrate Weights & Biases Sweeps for automated search.
4. **Limited Preprocessing**: The framework assumes input data is already in the desired representation (dispersion curves, spectrograms). Signal processing steps (FFT, STFT, wavelet transforms) are left to the user. A future "transforms" module could standardize these operations.
5. **Synthetic Data Bias**: The validation presented uses synthetically generated data from analytical models. Generalization to experimental data with noise, calibration errors, and missing modes requires domain adaptation techniques not currently included.

6.5. Future Directions

Based on the current implementation and community feedback, we identify the following development priorities:

1. **Unit Tests and CI/CD**: Adding comprehensive test coverage would enable automated verification of changes, critical for accepting community contributions without regression risks. Integration with GitHub Actions for continuous integration is planned.
2. **PyPI Distribution**: Packaging WaveDL for installation via `pip install wavedl` would simplify adoption and version management.
3. **Physics-Informed Loss Functions**: Incorporating known wave physics (e.g., dispersion curve smoothness, mode ordering constraints) as differentiable regularization terms could improve generalization and reduce data requirements.
4. **Pre-trained Models and Transfer Learning**: Providing pre-trained weights on standardized benchmark datasets would enable transfer learning, reducing training time for users with limited computational resources.
5. **Uncertainty Quantification**: Ensemble methods, Monte Carlo dropout, or Bayesian neural networks could provide confidence intervals on predictions—critical for high-stakes industrial applications.
6. **1D and 3D Extensions**: Generalizing the data pipeline to handle raw time-series (1D) and volumetric (3D) inputs would broaden applicability to diverse NDE scenarios.

7. Conclusion

We have presented **WaveDL**, a robust deep learning framework specifically designed for the inverse characterization of guided waves in non-destructive evaluation applications. By addressing the fundamental engineering challenges that have hindered the transition from proof-of-concept studies to production-ready research tools, WaveDL enables researchers to focus on scientific innovation rather than computational infrastructure.

Key Contributions:

1. **Zero-Copy Memory-Mapped Pipeline**: Enables training on datasets exceeding available RAM ($O(1)$ memory complexity with respect to dataset size), eliminating the data scale barrier.
2. **DDP-Safe Synchronization**: Prevents early stopping deadlocks and ensures correct metric aggregation across multi-GPU clusters, enabling reliable HPC deployment.
3. **Modular Registry Pattern**: Allows fair comparison of arbitrary neural network architectures without modifying the training loop, promoting reproducible benchmarking.
4. **Physics-Aware Metrics**: Automatically reports errors in physical units (mm, m/s, GPa), enabling direct assessment against engineering tolerances.
5. **Production-Ready Engineering**: Mixed-precision training, PyTorch 2.x compilation, and Weights & Biases integration provide the features essential for modern deep learning workflows.

The experimental validation demonstrates that WaveDL achieves sub-0.1 mm thickness accuracy and sub-5 m/s velocity accuracy on Lamb wave inversion, with inference speeds exceeding 2000× faster than traditional numerical solvers. This performance enables real-time industrial inspection at rates previously unattainable.

WaveDL is actively maintained and available at `https://github.com/ductho-le/WaveDL` under the permissive MIT license. We welcome contributions from the community, including new model architectures, additional utilities, and

documentation improvements. By providing a common, validated infrastructure for guided wave deep learning, we hope to accelerate progress in this rapidly evolving field.

Acknowledgments

Ductho Le acknowledges the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta Innovates for supporting this research through a research assistantship and graduate doctoral fellowship, respectively. This research was enabled in part by computational resources provided by Compute Ontario, Calcul Québec, and the Digital Research Alliance of Canada.

References

[1] Rose, J.L. (2014). *Ultrasonic Guided Waves in Solid Media*. Cambridge University Press.

[2] Su, Z., & Ye, L. (2009). *Identification of Damage Using Lamb Waves: From Fundamentals to Applications*. Springer.

[3] Balasubramaniam, K., & Krishnamurthy, C.V. (2008). Inverse characterization of composites from limited ultrasonic data using optimization methods. *Review of Quantitative Nondestructive Evaluation*, 27, 147–154.

[4] Rautela, M., & Gopalakrishnan, S. (2021). Deep learning for structural health monitoring: A review of Lamb wave-based damage detection. *Ultrasonics*, 116, 106496.

[5] Miorelli, R., Artusi, X., Reboud, C., Theodoulidis, T., & Poulakis, N. (2021). Supervised deep learning for ultrasonic crack characterization using numerical simulations. *NDT & E International*, 119, 102405.

[6] Yang, H., Liu, J., Wang, Y., & Liu, J. (2022). A convolutional neural network approach for inverse acoustic characterization of guided wave measurement. *Mechanical Systems and Signal Processing*, 169, 108759.

[7] Zhang, Y., Wang, X., Yang, Z., & Liu, Y. (2023). Physics-informed neural networks for Lamb wave-based damage identification in composite laminates. *Engineering Applications of Artificial Intelligence*, 117, 105564.

[8] Raissi, M., Perdikaris, P., & Karniadakis, G.E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686–707.

[9] Falcon, W., & The PyTorch Lightning team. (2019). *PyTorch Lightning*. https://pytorch-lightning.readthedocs.io

[10] Chollet, F. (2015). *Keras*. https://keras.io

[11] MONAI Consortium. (2020). *MONAI: Medical Open Network for AI*. https://monai.io

[12] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., et al. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of EMNLP*, 38–45.

[13] Gugger, S., Debut, L., Wolf, T., Schmid, P., Mueller, Z., & Manber, M. (2022). *Accelerate*. https://huggingface.co/docs/accelerate

[14] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., et al. (2020). PyTorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12), 3005–3018.

[15] Woo, S., Park, J., Lee, J.Y., & Kweon, I.S. (2018). CBAM: Convolutional block attention module. *Proceedings of the European Conference on Computer Vision (ECCV)*, 3–19.

[16] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32.

[17] Biewald, L. (2020). *Experiment Tracking with Weights and Biases*. https://www.wandb.com

[18] Wu, Y., & He, K. (2018). Group normalization. *Proceedings of the European Conference on Computer Vision (ECCV)*, 3–19.

[19] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

[20] Tan, M., & Le, Q.V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. *International Conference on Machine Learning*, 6105–6114.

[21] Dosovitskiy, A., Beyer, L., Kolesnikov, A., et al. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations*.

A. Complete Command Line Reference

B. File Format Specification

B.1. Input Data (NPZ)

```python
'input_train': numpy.ndarray
    # Shape: (num_samples, height, width)
    # Dtype: float32 recommended
    # Content: 2D representations of guided wave signals
    #   - Dispersion curves (frequency-wavenumber)
    #   - Spectrograms (time-frequency)
    #   - B-scans (position-time)
    #   - Other 2D transforms

'output_train': numpy.ndarray
    # Shape: (num_samples, num_targets)
    # Dtype: float32 recommended
    # Content: Material properties to predict
    #   - Thickness (mm)
    #   - Wave velocity (m/s)
    #   - Elastic moduli (GPa)
    #   - Density (kg/m^3)
    #   - Custom parameters
```

Listing 7: Required keys in train_data.npz

B.2. Checkpoint Structure

```
best_checkpoint/
|-- model.safetensors     # Model weights (via Accelerate)
|-- optimizer.bin           # Optimizer state
|-- scheduler.bin           # LR scheduler state
|-- random_states.pkl     # RNG states for reproducibility
+-- training_meta.pkl       # Custom metadata:
                            #   - epoch: int
                            #   - best_val_loss: float
                            #   - patience_ctr: int
```

B.3. Output Files

```
output_directory/
|-- best_checkpoint/       # Best model checkpoint
|-- epoch_10_checkpoint/  # Periodic checkpoint (if --save_every=10)
|-- epoch_20_checkpoint/
|-- ...
|-- best_model_weights.pth # Standalone weights file
|-- training_history.csv   # Epoch-by-epoch metrics
|-- train_data_cache.dat   # Memory-mapped input cache
|-- scaler.pkl             # Fitted StandardScaler
+-- data_metadata.pkl      # Shape and dimension info
```

Table 7
Command line arguments

Argument	Type	Default	Description
`-model`	str	`ratenet`	Registered model architecture
`-list_models`	flag	–	Print available models and exit
`-batch_size`	int	128	Per-GPU batch size
`-lr`	float	0.001	Initial learning rate
`-epochs`	int	1000	Maximum epochs
`-patience`	int	20	Early stopping patience
`-weight_decay`	float	0.0001	AdamW weight decay
`-grad_clip`	float	1.0	Maximum gradient norm
`-data_path`	str	`train_data.npz`	Training data path
`-workers`	int	8	DataLoader workers
`-seed`	int	2025	Random seed
`-resume`	str	None	Checkpoint to resume
`-save_every`	int	10	Checkpoint frequency
`-output_dir`	str	`.`	Output directory
`-compile`	flag	–	Enable torch.compile
`-precision`	str	`bf16`	Mixed precision mode
`-wandb`	flag	–	Enable W&B tracking
`-project_name`	str	`DL-Training`	W&B project name
`-run_name`	str	None	W&B run name

Software Availability

The WaveDL framework is freely available under the MIT License. Source code, documentation, and usage examples can be obtained from the GitHub repository: `https://github.com/ductho-le/WaveDL`. The software has been tested on Linux systems (Ubuntu 20.04, CentOS 7) with Python 3.11+ and PyTorch 2.0+.