Kinone: A Deep Learning Framework from First Principles

Name: Kshitish Kumar Ratha
Registration Number: MS22174

August 19, 2025

Abstract

This report details the design, implementation, and verification of Kinone[1], a deep learning framework developed from first principles using the NumPy scientific computing library. Motivated by the desire to demystify the internal mechanics of production-grade machine learning systems, this project constructs a transparent and auditable alternative. The framework's foundation is a reverse-mode automatic differentiation engine that operates on a dynamically generated computational graph, enabling the precise calculation of gradients for arbitrary scalar objectives through backpropagation. Built upon this core is a modular library of standard neural network primitives, including convolution, pooling, and normalization layers. The expressive power of this library is demonstrated through complete implementations of canonical deep learning architectures, specifically ResNet and EfficientNet. To support the full lifecycle of a machine learning project, Kinone is furnished with a custom data handling pipeline, a suite of common optimization algorithms, and tooling for interoperability, highlighted by an exporter to the ONNX format. A cornerstone of the development process was a rigorous verification strategy, where the analytical gradient of every operation was validated against numerical approximations from finite-difference methods. The result is a fully functional framework that serves as a powerful educational tool and an experimental platform for research requiring complete algorithmic control.

[1] The complete source code for this project is available at: https://github.com/pseudofractal/kinone.

Contents

1 Introduction

The field of deep learning has been fundamentally transformed by the availability of high-performance, open-source frameworks such as PyTorch [1], TensorFlow, and JAX. These libraries provide a high level of abstraction, enabling researchers and practitioners to design, train, and deploy complex neural network architectures with remarkable efficiency. This abstraction, however, while a catalyst for productivity, often obscures the fundamental mechanics that underpin gradient-based learning. The intricate processes of gradient computation, parameter updates, and memory management are handled by highly optimized, often opaque, backend engines. For many users, these core components remain a ”black box.”

1.1 Motivation and Problem Statement

While the productivity gains offered by modern frameworks are undeniable, a reliance on them without a foundational understanding of their inner workings can be a significant impediment to deep learning research and advanced application. When a model fails to train, or when a novel architecture requires a non-standard operation, a first-principles understanding of the underlying system becomes indispensable. The core motivation for the Kinone project stems from this educational and experimental imperative: to deconstruct the ”black box” by building a deep learning framework from scratch.

The central problem this project addresses is not a deficiency in existing tools, but rather a need for a system that prioritizes transparency, auditability, and pedagogical value over raw computational performance. The goal is to create a framework where every mathematical transformation and algorithmic step is explicit and accessible. Such a system serves two primary purposes. First, it is an unparalleled educational tool, forcing a concrete understanding of abstract concepts like backpropagation and the chain rule by requiring their direct implementation. Second, it provides a scaffold for research experiments that demand complete algorithmic control and deterministic behavior, which can sometimes be challenging to guarantee in frameworks that rely on device-specific kernels and complex optimizations.

Kinone is the proposed solution to this problem. It is a self-contained deep learning framework implemented entirely in Python with NumPy as its sole numerical dependency. By deliberately avoiding external machine learning libraries, it ensures that the complete chain of logic—from the forward pass through a network to the backward pass of gradients—is contained and auditable within the project’s own codebase.

1.2 Project Goals and Scope

To address the problem statement, the project was defined by a series of clear, concrete objectives. The successful completion of these goals resulted in the creation of a functional, end-to-end framework capable of handling a non-trivial machine learning task. The primary goals were as follows:

- **To design and implement a reverse-mode automatic differentiation engine.** This core component, often called an ”autograd” engine, needed to dynamically construct a computational graph and correctly propagate gradients from a scalar loss function back to every parameter in the model.
- **To build a comprehensive library of essential neural network layers and operations.** This involved creating robust and numerically stable implementations of standard primitives, including element-wise operations, matrix multiplication, 2D convolutions, pooling, and normalization layers.

- **To implement canonical deep learning architectures.** To validate the framework's expressive power and modularity, well-known models such as ResNet and EfficientNet were implemented using the developed library of layers.
- **To develop a complete data handling and training pipeline.** This encompassed creating a dataset-specific loader for a real-world dataset (NIH Chest X-ray), implementing standard optimization algorithms like Adam and SGD, and structuring a training script to manage the learning process.
- **To ensure correctness through a rigorous and systematic testing strategy.** A cornerstone of the project was to guarantee the correctness of the autograd engine. This was achieved by creating a full suite of unit tests where the analytically derived gradient for every operation was verified against a numerical approximation calculated using the finite-difference method.
- **To create supporting tools for analysis and interoperability.** To make the framework a more complete ecosystem, an ONNX exporter was developed to allow models to be used in other environments, and a Streamlit-based dashboard was created for monitoring the training process.
- **To run inference on edge-devices.** The ONNX export needs to be quantized by and then run inference on an AMD-Xilinx Kria KV260 Vision AI Starter Kit to showcase the efficiency of the architecture and the framework's ability to deploy models on resource-constrained devices.

The scope of this project was intentionally focused on correctness, clarity, and architectural completeness rather than computational performance. No GPU acceleration or low-level performance optimization was attempted. Kinone is therefore positioned as an educational and experimental tool, not as a performance-competitive replacement for production frameworks.

1.3 Report Structure

The remainder of this report is organized as follows. Section 2 reviews the theoretical foundations of automatic differentiation and gradient-based optimization that form the basis of the framework. Section 3 provides a detailed architectural overview and implementation details of the Kinone framework, from the core autograd engine to the high-level model definitions. Section 4 describes the verification and testing strategy used to ensure the correctness of the gradient calculations. Finally, Section 5 concludes the report with a summary of the project's achievements, key learnings, and potential directions for future work.

2 Theoretical Foundations

The implementation of the Kinone framework is predicated on a set of foundational mathematical and algorithmic principles that together enable modern gradient-based learning. This section provides a review of these core concepts, from the abstract representation of computation to the specific algorithms used for network optimization. Understanding these principles is essential for appreciating the design decisions made throughout the framework's architecture.

2.1 The Computational Graph

At its core, any differentiable program, from a simple arithmetic expression to a deep neural network, can be decomposed into a sequence of elementary operations. This decomposition can be formally represented as a Directed Acyclic Graph (dag), commonly known as a computational graph. In this graph, nodes represent either variables (e.g., tensors containing data, parameters, or intermediate values) or operations (e.g., matrix multiplication, addition, activation functions). The directed edges represent the flow of data, connecting the variables that serve as inputs to an operation, and connecting the operation to its resulting output variable.

Kinone adheres to the *define-by-run* paradigm, a concept popularized by frameworks like Chainer and PyTorch [1]. This means the computational graph is not statically declared beforehand but is built dynamically as the forward pass is executed. Each time an operation is applied to one or more tensors, a new output tensor is created, and a new node representing the operation is implicitly added to the graph. This operation node retains pointers to its input tensors (its ”parents”) and the output tensor (its ”child”).

This dynamic graph structure is not merely a conceptual model; it is the central data structure upon which the entire process of automatic differentiation rests. By recording the precise sequence of operations and their data dependencies, the graph provides a complete roadmap for the backward pass, ensuring that gradients can be correctly propagated back through the network from the final loss to every contributing parameter. Figure 1 illustrates a simple computational graph for a single neuron.

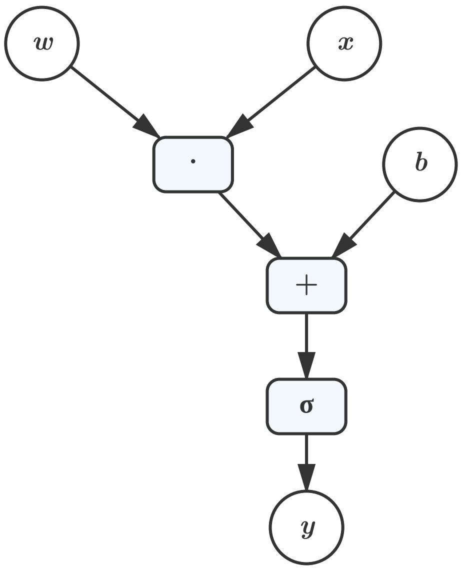


Figure 1: Computational Graph for a Single Neuron: $y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$

2.2 Reverse-Mode Automatic Differentiation

Automatic Differentiation (ad) is a family of techniques for numerically evaluating the derivative of a function specified by a computer program. For training deep neural networks, the preferred technique is reverse-mode ad, commonly known as backpropagation [2]. Its goal is to efficiently compute the gradient of a single scalar output (the loss, $\mathcal{L}$) with respect to a large number of input variables (the model parameters, θ).

The mathematical foundation of backpropagation is the multivariable chain rule. Consider a simple composition of vector functions, where a final scalar loss $\mathcal{L}$ is computed via an intermediate variable $\mathbf{u}$:

$$\mathbf{u} = g(\mathbf{v}) \quad \text{and} \quad \mathcal{L} = f(\mathbf{u}) \tag{1}$$

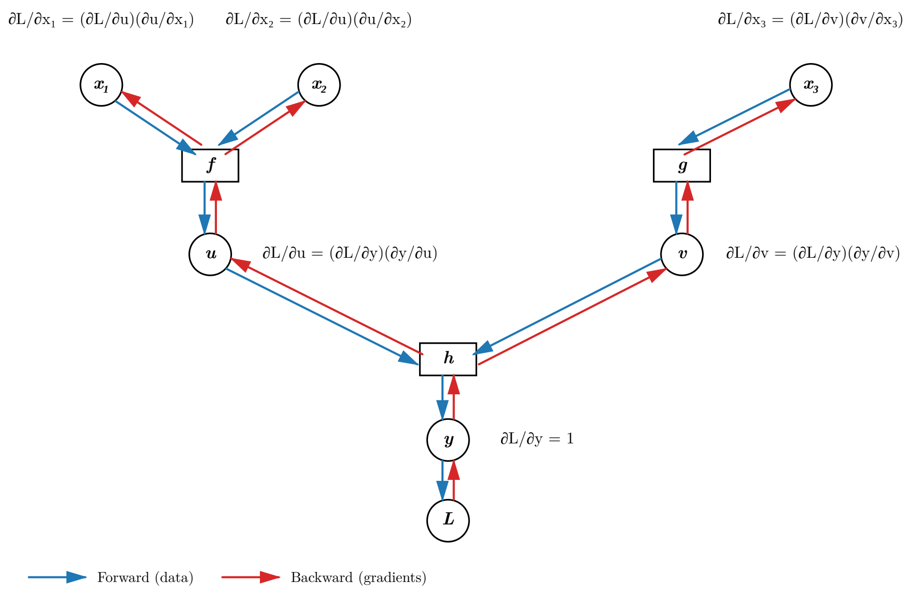


Figure 2: Reverse-mode AD on a small computational graph. Solid black arrows show the forward data flow; dashed red arrows show the backward gradient sweep, applying vector–Jacobian products $\partial\mathcal{L}/\partial u = (\partial\mathcal{L}/\partial y)(\partial y/\partial u)$, etc.

The chain rule states that the derivative of the loss $\mathcal{L}$ with respect to the input $\mathbf{v}$ can be expressed using the Jacobian matrices of the functions f and g:

$$\frac{\partial\mathcal{L}}{\partial\mathbf{v}} = \frac{\partial\mathcal{L}}{\partial\mathbf{u}}\frac{\partial\mathbf{u}}{\partial\mathbf{v}} = \nabla_{\mathbf{u}}\mathcal{L}\cdot J_g(\mathbf{v}) \tag{2}$$

where $J_g(\mathbf{v})$ is the Jacobian of g evaluated at $\mathbf{v}$, and $\nabla_{\mathbf{u}}\mathcal{L}$ is the gradient of $\mathcal{L}$ with respect to $\mathbf{u}$.

Reverse-mode ad applies this rule by traversing the computational graph backward, from the final output node to the input nodes. At each operation node, it computes the gradient with respect to that node's inputs based on the gradient it has already received from its output (the "upstream gradient"). If we denote the gradient of the final loss $\mathcal{L}$ with respect to a tensor $\mathbf{x}$ as $\bar{\mathbf{x}} = \nabla_{\mathbf{x}}\mathcal{L}$, the core update rule for an operation $\mathbf{u} = g(\mathbf{v})$ is:

$$\bar{\mathbf{v}} = J_g(\mathbf{v})^\top\bar{\mathbf{u}} \tag{3}$$

This process begins by initializing the gradient of the loss with respect to itself to one ($\bar{\mathcal{L}} = 1$). The algorithm then proceeds in reverse topological order of the graph, applying this update rule at each node. A key detail is that if a variable is used in multiple subsequent operations, its total gradient is the sum of the gradients propagated back from each of those branches.

This method is exceptionally efficient for deep learning because it calculates the gradients for all parameters in a single backward pass through the graph, with a computational cost that is a small constant multiple of the forward pass. Kinone's ‘Tensor.backward()‘ method is a direct implementation of this algorithm, performing a depth-first search to establish the topological order before accumulating gradients.

2.3 Neural Network Primitives

The power of a deep learning framework comes from its library of pre-defined, differentiable operations that can be composed to form complex models. The following are key primitives implemented in Kinone.

2.3.1 2D Convolution

The 2D convolution is the cornerstone of modern computer vision models, famously used in pioneering architectures such as LeNet-5 [3]. It operates by sliding a small, learnable kernel (or filter) over the spatial dimensions of a multi-channel input tensor. At each position, it computes the dot product between the kernel's weights and the corresponding input patch. For an input tensor **X** and a kernel **W**, the output **Y** is given by:

$$Y_{n,c_{out},i,j} = b_{c_{out}} + \sum_{c_{in}=0}^{C_{in}-1} \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} X_{n,c_{in},i\cdot s+k,j\cdot s+l} \cdot W_{c_{out},c_{in},k,l} \tag{4}$$

where s is the stride and b is an optional bias term. This operation allows the model to learn hierarchical spatial features, such as edges, textures, and eventually complex objects.

2.3.2 Batch Normalization

Batch Normalization is a critical technique for stabilizing and accelerating the training of deep networks [4]. It normalizes the output of a previous layer by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. The normalized output is then scaled and shifted by two learnable parameters, γ (gamma) and β (beta). For a mini-batch of activations $\mathcal{B} = \{x_1, \ldots, x_m\}$, the operation for each activation x_i is:

$$\begin{aligned}
\mu_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{(mini-batch mean)} \\
\sigma_{\mathcal{B}}^2 &= \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad \text{(mini-batch variance)} \\
\hat{x}_i &= \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \text{(normalize)} \\
y_i &= \gamma \hat{x}_i + \beta \quad \text{(scale and shift)}
\end{aligned}$$

During inference, the mean and variance are not computed per-batch but are instead replaced by running averages of these statistics collected during training. The backward pass for this operation is non-trivial, as the gradient for each sample depends on all other samples in the mini-batch.

2.4 Gradient-Based Optimization

Once the gradients of the loss function with respect to all model parameters ($\nabla_\theta \mathcal{L}$) have been computed, an optimizer uses these gradients to update the parameters in a direction that is expected to minimize the loss. All optimizers follow a general update rule:

$$\theta_{t+1} \leftarrow \theta_t - \eta \cdot \Delta\theta_t \tag{5}$$

where θ_t are the parameters at timestep t, η is the learning rate, and $\Delta\theta_t$ is the update step derived from the gradients. Kinone implements several standard algorithms.

2.4.1 Stochastic Gradient Descent (SGD)

The simplest optimizer, SGD, uses the raw gradient as the update direction: $\Delta\theta_t = \nabla_{\theta_t}\mathcal{L}(\theta_t)$. A common extension is SGD with momentum, which introduces a velocity term $\mathbf{v}_t$ that accumulates an exponentially decaying moving average of past gradients. This helps to dampen oscillations and accelerate convergence along persistent directions. The update rules are:

$$\mathbf{v}_{t+1} \leftarrow \mu \mathbf{v}_t + \nabla_{\theta_t} \mathcal{L}(\theta_t)$$
$$\theta_{t+1} \leftarrow \theta_t - \eta \mathbf{v}_{t+1}$$

where μ is the momentum coefficient.

2.4.2 Adaptive Optimizers: RMSProp and Adam

Adaptive methods adjust the learning rate for each parameter individually. **RMSProp** [5] maintains a moving average of the squared gradients for each parameter and divides the learning rate by the square root of this average. This effectively decreases the learning rate for parameters with large, consistent gradients and increases it for parameters with small or infrequent gradients.

Adam (Adaptive Moment Estimation) is arguably the most common optimizer in modern deep learning [6]. It combines the ideas of momentum (the first moment of the gradients) and per-parameter adaptive learning rates (the second moment of the gradients). It maintains two moving averages for each parameter:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{(first moment estimate)}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad \text{(second moment estimate)}$$

where $g_t = \nabla_{\theta_t} \mathcal{L}$. Adam also includes a bias-correction step to account for the fact that these estimates are initialized at zero. The final parameter update is then:

$$\theta_{t+1} \leftarrow \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{6}$$

where $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected moment estimates. This combination provides a robust and efficient optimization algorithm that works well across a wide range of problems.

3 Framework Architecture and Implementation

The Kinone framework is designed as a layered system, with each layer of abstraction building upon the one below it. This modular design promotes code reuse, simplifies debugging, and maintains a clear separation of concerns. This section provides a comprehensive overview of the framework's architecture, starting from the core automatic differentiation engine and progressing to the high-level model implementations and supporting tools.

3.1 The Core Autograd Engine

The engine, located in src/core/, is the cornerstone of the framework. It is responsible for constructing the computational graph and performing backpropagation. It consists of two primary classes: Tensor and Function.

3.1.1 The Tensor Class

The Tensor class, defined in src/core/tensor.py, is the fundamental data structure in Kinone. It serves as a wrapper around a NumPy ndarray and acts as a node in the computational graph. Its primary responsibilities are to store the raw data, track whether gradients are required, and accumulate gradients during the backward pass. The key attributes of a Tensor are:

- data: The underlying np.ndarray that holds the numerical data.

- requires_grad: A boolean flag indicating whether this tensor is a parameter or an intermediate value that requires a gradient to be computed for it.
- grad: A Tensor object that stores the accumulated gradient. It is initialized to None and is populated during the backward pass.
- _ctx: A reference to a Function object that created this tensor. This is the link that forms the backward graph, pointing to the operation that generated this tensor. For user-created tensors (leaf nodes), this is None.

By overloading standard arithmetic operators (e.g., __add__, __mul__), the Tensor class provides a seamless and intuitive API for building the computational graph.

3.1.2 The Function Abstraction

Every operation in the graph is an instance of a class that inherits from the abstract base class Function, defined in src/core/ops.py. This class provides the machinery for both the forward and backward passes of an operation. Its design is centered around a single class method, apply, which orchestrates the graph construction.

```python
@classmethod
def apply(cls, *args, **kwargs) -> Tensor:
  context = cls()
  context.parents = [arg for arg in args if isinstance(arg, Tensor)]

  raw_args = [arg.data if isinstance(arg, Tensor) else arg for arg in args]
  output_data = cls.forward(context, *raw_args, **kwargs)

  requires_grad = any(p.requires_grad for p in context.parents)
  output_tensor = Tensor(output_data, requires_grad)
  output_tensor._ctx = context if requires_grad else None

  return output_tensor
```

Listing 1: The Function.apply method orchestrates graph construction.

When an operation like add(a, b) is called, it internally invokes Add.apply(a, b). This method performs several critical steps:

1. It instantiates the specific function class (e.g., Add), creating a context object.
2. It saves the input tensors (a and b) as context.parents.
3. It calls the static forward method with the raw NumPy data to compute the result.
4. It creates a new output Tensor to hold the result.
5. Crucially, it sets the _ctx attribute of the new output tensor to the context object, thereby linking the output to the operation and inputs that created it.

Each concrete Function subclass must implement two methods: forward, which defines the mathematical computation, and backward, which defines the analytical derivative (the vector-Jacobian product) for that operation.

3.1.3 The Backpropagation Algorithm

The backpropagation process is initiated by calling the backward() method on the final scalar tensor (typically the loss). This method, implemented in src/core/tensor.py, executes the reverse-mode automatic differentiation algorithm described in Section 2.2. The implementation proceeds in two main phases:

1. **Topological Sort:** First, a correct processing order for the backward pass must be established. The graph is traversed from the final node backwards using a depth-first search (dfs), and nodes are added to a list in post-order. This effectively creates a reverse topological sort of the graph, ensuring that when we process a node, we have already processed all of its successors.

2. **Gradient Accumulation:** The algorithm then iterates through the sorted list of nodes in reverse. For each tensor in the list, it accesses its _ctx (the Function that created it). It then calls the backward method of that Function, passing the upstream gradient that has been accumulated in the current tensor's .grad attribute. The backward method returns the local gradients with respect to each of its inputs. These local gradients are then accumulated into the .grad attributes of the parent tensors.

This process continues until all nodes in the graph that contributed to the final output have been visited, at which point the .grad attribute of every parameter tensor contains the complete gradient of the loss with respect to that parameter.

3.2 Library of Operations

Building on the Function abstraction, Kinone implements a wide array of differentiable operations in src/core/ops.py. These primitives are the fundamental building blocks for all layers and models.

Table 1: A Selection of Implemented Primitive Operations

Category	**Operators**
Element-wise	Add, Sub, Mul, Div, Neg
Activations	ReLU, Sigmoid, Tanh, Swish, HardSigmoid, HardSwish
Reductions	Sum, Mean
Linear Algebra	MatMul
Convolutions	Conv2d, ConvNd, ConvTransposeNd

A notable implementation detail is the handling of convolution. To maintain efficiency without relying on external libraries, the Conv2d operation uses the im2col (image-to-column) technique. This method transforms the local patches of the input image into columns of a large matrix. The convolution operation can then be reformulated as a single, highly optimized matrix multiplication between this matrix and the reshaped convolution kernels. The backward pass is implemented via the corresponding col2im operation. Furthermore, a utility function, _unbroadcast, was implemented to correctly handle the backward pass for operations involving NumPy broadcasting, ensuring that gradients are correctly summed over the broadcasted dimensions.

3.3 Neural Network Modules

To facilitate the construction of complex models, Kinone provides a higher level of abstraction through its Module system, located in src/core/modules.py, pool.py, and batchnorm.py. This system is heavily inspired by the design of PyTorch.

3.3.1 The Module Base Class

The Module class is the base class for all neural network layers. It provides essential functionality for managing a layer's state and parameters:

- **Parameter Tracking:** It automatically registers any attributes that are instances of Tensor or other Modules as part of its state. The parameters() and named_parameters() methods provide convenient iterators for accessing all parameters within a module and its sub-modules, which is essential for passing them to an optimizer.
- **State Management:** It maintains a training attribute to switch between training and evaluation modes. This is critical for layers like BatchNorm2d and Dropout that behave differently in each mode.
- **Serialization:** The load_state_dict method allows for loading pre-trained weights from a dictionary, enabling model checkpointing and inference.

3.3.2 Layer Implementations

Specific layers are implemented as subclasses of Module. They encapsulate the parameters and the forward pass logic for a common neural network operation.

- Linear: Implements a fully connected layer. It initializes weight and bias tensors and its forward pass is defined as a composition of the primitive matmul and add operations.
- Conv2D: Encapsulates a 2D convolution. It holds the kernel and bias tensors as parameters and calls the conv2d primitive function in its forward pass.
- BatchNorm2d: Implements 2D batch normalization. It manages the learnable gamma and beta parameters as well as the non-learnable running_mean and running_var buffers.
- Sequential: A container module that chains a sequence of other modules together, allowing for the easy construction of feed-forward network blocks.

3.4 Model Architectures

The modular design of the framework culminates in the ability to construct complex, state-of-the-art model architectures, which are defined in src/core/models/.

3.4.1 ResNet Implementation

The implementation of the ResNet family of models (resnet.py) demonstrates the framework's ability to handle one of the most influential architectures in computer vision [7]. It includes implementations of the BasicBlock and Bottleneck residual blocks. The critical skip connection is implemented simply by adding the block's input to its output using the framework's add operation before the final activation function, showcasing the power of the underlying computational graph.

3.4.2 EfficientNet Implementation

To further test the framework's capabilities, a version of the EfficientNet-B0 architecture was implemented (efficientnet.py) [8]. This required building more complex components, such as the Squeeze-and-Excite block [9], which performs channel-wise attention, and the Inverted Residual Block (MBConv), which uses depthwise separable convolutions. The successful implementation of this modern architecture validates the flexibility and expressiveness of the Kinone module system.

3.5 Data Handling Pipeline

A complete machine learning workflow requires a robust pipeline for ingesting and preparing data. This functionality is encapsulated within the src/data/ directory.

3.5.1 The DataModule Abstraction

To organize data-related logic in a clean and reusable way, the abstract base class DataModule (datamodule.py) was created. It defines a standard interface for handling data, with methods for setup and for creating training, validation, and test data loaders.

3.5.2 NIH Chest X-ray Dataset and DataLoader

The framework includes a specific implementation for the NIH Chest X-ray dataset (nih_cxr.py). The NIHChestDataset class handles parsing the metadata CSV file, mapping the multi-label disease findings to a one-hot encoded vector, and loading the corresponding images. A custom DataLoader class was also implemented to handle batching and shuffling of the data, maintaining the ”from scratch” philosophy of the project.

3.5.3 Stratified Splitting

Given the significant class imbalance inherent in many medical imaging datasets, a custom stratified splitting function was implemented in nih_datamodule.py. This function ensures that the train, validation, and test splits maintain approximately the same class distribution as the original dataset, which is critical for reliable model evaluation.

3.6 Tooling and Interoperability

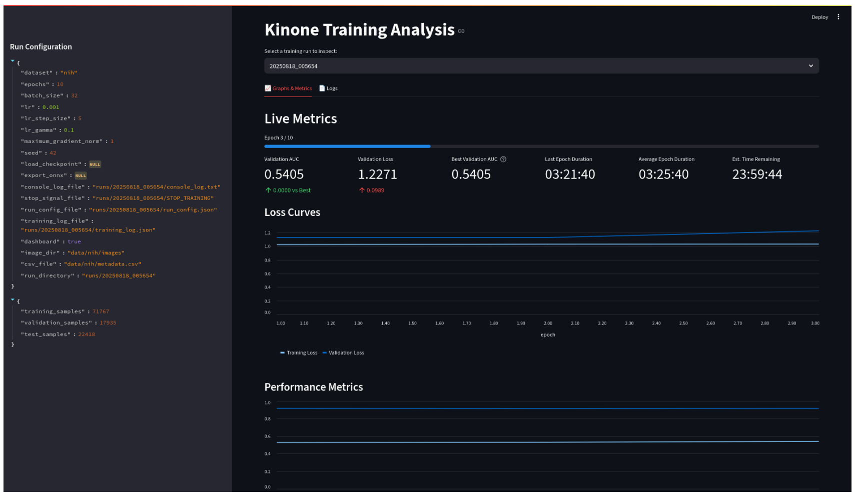
Beyond the core training components, a suite of tools was developed to support a more complete and user-friendly workflow.

3.6.1 ONNX Export

A key feature of Kinone is its ability to export trained models to the Open Neural Network Exchange (onnx) format. The exporter, located in src/core/onnx.py, works by performing another traversal of the model’s computational graph. For each Function context encountered, it generates a corresponding ONNX node protobuf with the correct attributes (e.g., strides for a convolution). The model’s parameters (weights and biases) are serialized as initializers in the ONNX file. This powerful feature allows models trained in Kinone to be deployed and run in high-performance inference engines like ONNX Runtime.

3.6.2 Training and Evaluation Scripts

The train.py script provides a complete training loop. It integrates all the components of the framework: it instantiates the data module, builds the model, initializes the optimizer and a learning rate scheduler, and runs the training process for a specified number of epochs. The evaluate.py



Run Configuration

```
{
  "dataset" : "nih"
  "epochs" : 10
  "batch_size" : 32
  "lr" : 0.001
  "lr_step_size" : 5
  "lr_gamma" : 0.1
  "maximum_gradient_norm" : 1
  "seed" : 42
  "load_checkpoint" : NULL
  "export_onnx" : NULL
  "console_log_file" : "runs/20250818_005654/console_log.txt"
  "stop_signal_file" : "runs/20250818_005654/STOP_TRAINING"
  "run_config_file" : "runs/20250818_005654/run_config.json"
  "training_log_file" :
  "runs/20250818_005654/training_log.json"
  "dashboard" : true
  "image_dir" : "data/nih/images"
  "csv_file" : "data/nih/metadata.csv"
  "run_directory" : "runs/20250818_005654"
}
{
  "training_samples" : 71767
  "validation_samples" : 17935
  "test_samples" : 22418
}
```

Kinone Training Analysis

Select a training run to inspect: 20250818_005654

Live Metrics

Epoch 3 / 10

Validation AUC	Validation Loss	Best Validation AUC	Last Epoch Duration	Average Epoch Duration	Est. Time Remaining
0.5405	1.2271	0.5405	03:21:40	03:25:40	23:59:44
↑ 0.0000 vs Best	↑ 0.0989				

Figure 3: Kinone Dashboard

script allows for the evaluation of a saved model checkpoint on the test set, computing final performance metrics.

3.6.3 Monitoring Dashboard

To provide real-time insight into the training process, an interactive dashboard was created using Streamlit (dashboard.py). This dashboard runs as a separate process and monitors the training run's output files. It displays live plots of training and validation loss, performance metrics like AUC, and provides controls to gracefully stop the training process. Communication between the training script and the dashboard is handled via a simple but effective file-based messaging system, where the training script writes to log files that the dashboard reads periodically.

4 Verification and Testing Strategy

In a system as complex as an automatic differentiation framework, correctness is not an incidental feature but the primary requirement. A subtle error in the gradient calculation for a single primitive operation can lead to catastrophic failures in model training that are exceedingly difficult to debug. Consequently, a rigorous and systematic verification strategy was a cornerstone of the Kinone project. The entire framework is underpinned by a comprehensive test suite that ensures the fidelity and reliability of every component, with a particular focus on the autograd engine.

4.1 Unit Testing with Pytest

The project employs a standard unit testing methodology using the Pytest framework. The test suite, located in the tests/ directory, is organized to mirror the structure of the main source code. This modular approach ensures that every component of the framework, from data loaders to optimizers, is tested in isolation. Key areas covered by the unit tests include:

- **Data Pipeline:** Verifying correct batching, shuffling, and data shapes from the DataLoader.
- **Loss Functions:** Ensuring that loss calculations are numerically correct and produce the expected gradients.

- **Optimizers:** Testing that parameter updates are applied correctly according to the specific algorithm's rules.
- **Module and Layer Functionality:** Confirming that model layers are constructed correctly and that state management (e.g., switching between training and evaluation modes) functions as expected.

While these tests are essential for verifying the high-level logic of the framework, the most critical part of the testing strategy is the specific method used to validate the correctness of the autograd engine itself.

4.2 Gradient Verification via Finite Differences

The correctness of the entire gradient-based learning process relies on the analytical gradients implemented in the backward method of each Function subclass. To verify these analytical implementations, a numerical approximation of the gradient is used as an independent ground truth. The chosen method for this numerical approximation is the finite-difference method.

Specifically, the project uses the second-order central-difference formula to estimate the partial derivative of a scalar function $f(\mathbf{x})$ with respect to a single component x_i of its input vector $\mathbf{x}$:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h} \tag{7}$$

where $\mathbf{e}_i$ is a standard basis vector with a one at the i-th position and zeros elsewhere, and h is a small, finite step size (chosen as 10^{-3} for float32 precision). By iterating this calculation over all elements of the input tensor $\mathbf{x}$, we can construct a full numerical gradient, $\hat{\nabla} f(\mathbf{x})$.

A helper function, finite_difference_gradients, was implemented in tests/utils.py to automate this process for any given function and set of input tensors. This function became the core of the verification strategy for the autograd engine. For every single primitive operation implemented in Kinone, a dedicated test was written that follows this procedure:

1. Instantiate one or more input Tensors with requires_grad=True.
2. Apply the operation to these tensors to produce an output tensor.
3. To obtain a scalar loss, a reduction (e.g., mean() or sum()) is applied to the output.
4. The backward() method is called on the final scalar loss, which triggers the autograd engine to compute and populate the analytical gradients in the .grad attribute of the input tensors.
5. Separately, the finite_difference_gradients utility is used to compute the numerical gradients for the same composite function.
6. Finally, the analytical gradient computed by Kinone is compared to the numerical gradient using numpy.allclose, which checks for element-wise equality within a specified tolerance.

This procedure was applied exhaustively to every operation, including element-wise functions, matrix multiplication, convolutions with various parameters (stride, padding, bias), pooling, and batch normalization. An example of a typical gradient test for a binary operation is shown in Listing 2.

```python
import numpy as np
from src.core.ops import mul
from src.core.tensor import Tensor
from tests.utils import assertion, finite_difference_gradients

```

```python
def test_mul_grad():
  shape = (4, 4)
  a = Tensor(np.random.randn(*shape), requires_grad=True)
  b = Tensor(np.random.randn(*shape), requires_grad=True)

  # 1. Compute analytical gradients via Kinone's autograd
  mul(a, b).mean().backward()

  # 2. Compute numerical gradients via finite differences
  num_grad_a, num_grad_b = finite_difference_gradients(
    lambda x, y: (x * y).mean(), [a.data.copy(), b.data.copy()]
  )

  # 3. Assert that the two are close
  assertion(a.grad.data, num_grad_a)
  assertion(b.grad.data, num_grad_b)
```

Listing 2: Example of a gradient verification test for the mul operation.

This comprehensive, gradient-centric testing strategy provides a very high degree of confidence in the correctness of the framework's core engine. It ensures that any model built with Kinone will have mathematically correct gradients, forming a reliable foundation for all training and experimentation.

5 Conclusion and Future Work

This project set out with the ambitious goal of building a modern deep learning framework from first principles, driven by the motivation to create a system that is transparent, educational, and algorithmically explicit. By adhering to a philosophy of implementing every core component from scratch using only NumPy, the Kinone framework has successfully met these objectives. This final section summarizes the key achievements of the project, reflects on the valuable insights gained during its development, and outlines promising directions for future work.

5.1 Summary of Project

Over the course of this project, a complete, end-to-end deep learning ecosystem was successfully engineered. The key accomplishments can be summarized as follows:

- **A Functional Autograd Engine:** A robust reverse-mode automatic differentiation engine was implemented. It dynamically builds a computational graph and correctly computes gradients for complex, composite functions, forming the reliable core of the entire framework.
- **A Modular Library of Neural Network Primitives:** A comprehensive set of standard neural network operations, layers, and activation functions was developed. This modular library enables the flexible construction of a wide variety of network architectures.
- **Implementation of State-of-the-Art Architectures:** The framework's capabilities were validated through the successful implementation of complex, canonical models, including the ResNet and EfficientNet families. This demonstrated that the abstractions were powerful enough to support modern deep learning research.
- **An End-to-End Workflow:** Kinone provides a complete workflow for a machine learning project, from data ingestion and preparation with a custom data pipeline to model training with multiple optimization algorithms and finally to evaluation.

- **Verified Correctness:** The correctness of the framework, particularly the critical gradient calculations, was systematically verified through a rigorous testing suite that benchmarked all analytical gradients against numerical finite-difference approximations.
- **Advanced Tooling for Interoperability and Analysis:** The framework is augmented with powerful tools, including an ONNX exporter for model deployment and a live Streamlit dashboard for real-time training monitoring, elevating it from a simple library to a more complete development environment.

5.2 Key Learnings

The process of building Kinone from the ground up provided numerous invaluable insights that are often abstracted away by high-level libraries. A primary lesson was a deeper, more intuitive understanding of the backpropagation algorithm; implementing the backward pass for diverse operations like broadcasting, convolution, and batch normalization solidified the theoretical concepts in a practical context. The project also highlighted the nuances of numerical stability, particularly in the implementation of activation and loss functions. Furthermore, it fostered an appreciation for the software engineering patterns that make deep learning frameworks both flexible and scalable, such as the modular design, the separation of operations from layers, and the importance of a clean and consistent API.

5.3 Future Directions

While Kinone has achieved its initial goals, it also serves as a strong foundation for many potential extensions and improvements. Future work could proceed in several promising directions:

- **Performance Optimization:** The most significant limitation of the current framework is its reliance on a pure Python and NumPy backend, which makes it computationally slow. A major avenue for future work would be to integrate a high-performance backend. This could involve using a Just-In-Time (jit) compiler like Numba to accelerate critical loops, or more ambitiously, creating an alternative backend that translates the computational graph to a system like JAX or CuPy to enable seamless GPU acceleration.
- **Expanding the Operator Set:** The current library of operations could be expanded to support a wider range of model architectures. Key additions would include operations required for Transformer models, such as multi-head self-attention, and operations for recurrent neural networks (rnns), like lstm or gru cells.
- **Advanced Framework Features:** The framework could be enhanced with more sophisticated features found in production libraries. This might include implementing higher-order differentiation (the ability to compute gradients of gradients), a more diverse set of learning rate schedulers, and built-in support for model checkpointing and automated experiment tracking.
- **Embedded Deployment and Model Quantization:** A particularly exciting direction would be to bridge the gap between training and deployment on edge devices. The existing ONNX export capability is the first step in this process. A future project could build a pipeline that takes the exported ONNX model, applies post-training quantization to reduce its size and computational requirements (e.g., converting float32 weights to

int8), and then deploys it to an embedded ai platform. Using a toolchain like Vitis ai, the quantized model could be compiled for deployment on System-on-Chip (soc) boards with programmable logic, such as the AMD-Xilinx Kria KV260 Vision AI Starter Kit, enabling efficient, low-latency inference directly on hardware. This would complete the full lifecycle from theoretical implementation to real-world application.

In its current state, Kinone stands as a testament to the power of building from first principles. It is a robust, well-tested, and transparent framework that successfully achieves its goal of demystifying the core mechanics of deep learning.

6 Code Availability

The complete source code for the Kinone framework, including all experiments and testing scripts detailed in this report, is publicly available on GitHub: https://github.com/pseudofractal/Kinone/.

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*, pp. 8026–8037, 2019.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [4, 11, 15, 17]

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. [26, 41]

[4] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International Conference on Machine Learning*, pp. 448–456, PMLR, 2015. [1, 2, 5, 9, 18]

[5] G. Hinton, “Neural networks for machine learning, lecture 6e.” Coursera, 2012. [22, 23, 24]

[6] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014. [6, 20, 25, 29]

[7] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016. [30, 32, 40]

[8] M. Tan and Q. V. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International Conference on Machine Learning*, pp. 6105–6114, PMLR, 2019. [3, 10, 13, 14]

[9] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7132–7141, 2018. [19, 21, 31, 38]