From Scratch: Reverse-Mode Automatic Differentiation and Integrated Gradients for Interpretability in Neural and Physics-Based Models

Course code: EE5311 Differentiable and Probabilistic Computing
Electrical and Computer Engineering

Supervised By: Prof. CHITRE, Mandar Anil

Written By:
Liu Fei (A0275104M)
Cao Yuan (A0275177U)
Jin Xuan (A0328457U)
Gao Jiaxuan (A0332428H)
Nan Jinyao (A0319482X)

Date Last Edited: Feb 16, 2025

Abstract

We present a lightweight reverse-mode automatic differentiation (AD) engine implemented from scratch in Julia. The system builds dynamic computational graphs using a `Tensor` abstraction with operator overloading, and computes gradients via closure-based pullbacks (VJPs) scheduled by an iterative DFS topological traversal. Elementwise operations are supported through a broadcast-aware rule registry with shape-correct gradient reduction (`unbroadcast`). On top of this AD core, we implement Integrated Gradients (IG) for interpretability, supporting both AD-based gradients and finite differences for scalar black-box functions. Case studies on projectile-motion optimization and Iris MLP attribution demonstrate a unified pipeline for sensitivity analysis across physics-based and neural models. Our source code is available at: https://github.com/nanjinyao/5311_CA1.

“

1 Introduction

Automatic Differentiation (AD) is a foundational tool in modern scientific computing, enabling efficient and exact gradient computation for optimization, learning, and sensitivity analysis. While industrial frameworks provide highly optimized AD systems, understanding their internal mechanics is essential for both research transparency and algorithmic innovation. In this work, we implement a reverse-mode AD engine from scratch in Julia, emphasizing clarity, extensibility, and architectural rigor.

The system is built around a mutable `Tensor` abstraction that dynamically constructs computational graphs through operator overloading. Reverse-mode differentiation is implemented using closure-based Vector-Jacobian Products (VJPs) and executed via an iterative Depth-First Search (DFS) topological traversal, avoiding recursion limits and ensuring stable gradient scheduling.

A distinguishing feature of our implementation is a broadcast-aware differentiation mechanism. Instead of hard-coding elementwise operations, we integrate with Julia's broadcasting system through a custom `BroadcastStyle`. Backward rules for primitive functions (e.g., addition, multiplication, trigonometric and exponential functions) are managed via an extensible registry. To correctly propagate gradients through broadcasted operations, we introduce an `unbroadcast` operator that reduces gradients to their original operand shapes. This design separates forward semantics from backward logic and enables modular extension of differentiable primitives.

On top of the AD core, we implement Integrated Gradients (IG) as a unified attribution method for interpretability. The framework supports both AD-based gradients and finite-difference gradients for scalar-valued black-box functions, allowing consistent interpretability analysis across neural networks and differentiable physical models.

Through experiments on projectile-motion optimization and Iris classification, we demonstrate that gradients serve as a universal computational primitive bridging physics-based reasoning and deep learning interpretability.

2 Core Principle: Chain Rule and Vector–Jacobian Products (VJP)

Reverse-mode AD computes gradients efficiently for a scalar objective by repeatedly applying the chain rule along a computational graph. Let L be a scalar output and let an intermediate node u feed into one or more downstream nodes $v_1, \dots, v_k$. By the multivariate chain rule, the total derivative with respect to u is the sum of all downstream contributions:

$$\frac{\partial L}{\partial u} = \sum_{i=1}^{k} \frac{\partial L}{\partial v_i} \frac{\partial v_i}{\partial u}. \tag{1}$$

This “fan-out” case is ubiquitous in computational graphs: a value may be reused by multiple operators, so its gradient must accumulate contributions from multiple paths.

Our implementation follows the standard VJP (pullback) formulation used in reverse-mode AD. For each primitive operation producing $v = f(u)$, we do not form Jacobians explicitly. Instead, we store a local *pullback* that maps the downstream sensitivity $g_v \triangleq \partial L/\partial v$ to the upstream sensitivity $g_u \triangleq \partial L/\partial u$:

$$g_u \ += \ g_v \cdot \frac{\partial f(u)}{\partial u}, \tag{2}$$

where $+=$ denotes in-place accumulation. In code, this corresponds to updating `parent.grad .+= ...` inside each node’s `_backward` closure. The closure captures any forward-pass context required to evaluate the local derivative (e.g., operands for multiplication, masks for ReLU, or cached broadcast results for certain elementwise functions).

By repeatedly executing these pullbacks in a valid reverse-topological order (from outputs back to inputs), the engine ensures that when a node’s pullback is invoked, its `grad` field already contains the total accumulated $\partial L/\partial(\texttt{node})$ from all of its downstream consumers. This invariant enables correct gradient propagation without materializing large Jacobian matrices and forms the mathematical backbone of the reverse replay procedure used throughout the project.

3 Core Abstraction: The `Tensor` Struct

The entire AD engine is built around a mutable `Tensor` abstraction. Each `Tensor` instance represents a node in a dynamic computational graph and encapsulates both numerical data and the information required for reverse-mode differentiation.

Listing 1: Core Tensor abstraction from MiniAD.jl

```julia
mutable struct Tensor
    data::Array{Float64}
    grad::Array{Float64}
    _backward::Function
    _prev::Set{Tensor}
    op::String
    requires_grad::Bool

    function Tensor(data; _children=(), _op="", requires_grad=true)
        d = if data isa Number
            reshape([Float64(data)], 1, 1)
        else
            arr = convert(Array{Float64}, data)
            ndims(arr) == 1 ? reshape(arr, :, 1) : arr
        end

        g = zeros(size(d))
        prev = requires_grad ? Set{Tensor}(_children) :
            Set{Tensor}()
        new(d, g, () -> nothing, prev, _op, requires_grad)
    end
end
```

3.1 Design Principles

Uniform 2D Representation. All numerical inputs are internally normalized to two-dimensional arrays: scalars are reshaped to $(1, 1)$ matrices, and vectors are reshaped to $(n, 1)$

column matrices. This design ensures that all operations—particularly matrix multiplication—operate under consistent linear algebra semantics without requiring separate scalar or vector code paths.

Gradient Accumulator. Each tensor stores a `grad` field of identical shape to `data`. During backpropagation, gradients are accumulated in-place using `.+=`. This accumulation is essential for handling fan-out in computational graphs, where a single tensor contributes to multiple downstream operations.

Closure-Based Pullback. The field `_backward` stores a closure implementing the local Vector–Jacobian Product (VJP). Instead of constructing global Jacobians, each operation defines how downstream sensitivities are mapped back to its parents. The closure captures any forward-pass context required for computing derivatives, enabling modular differentiation logic per operation.

Graph Connectivity and Structural Pruning. The field `_prev` records parent tensors for graph traversal. However, parents are only tracked if `requires_grad=true`. This conditional tracking serves as structural pruning: nodes that do not participate in gradient computation are excluded from the graph, reducing memory overhead and traversal cost.

Together, these components define a minimal yet fully functional reverse-mode AD node abstraction. The computational graph emerges dynamically during forward execution, while the stored pullbacks enable efficient reverse replay without explicit symbolic differentiation.

3.2 Relation to Industrial Frameworks

Readers familiar with deep learning frameworks such as PyTorch [6] will recognize this as the **Define-by-Run** paradigm. Our `Tensor` struct plays a similar role to `torch.Tensor` (specifically with `requires_grad=True`), and the closure-based backward pass mimics the Autograd engine.

However, unlike PyTorch's highly optimized C++ backend, MiniAD is implemented entirely in high-level Julia. This offers a pedagogical advantage: the entire lifecycle of a gradient—from graph construction to VJP execution—is transparent and inspectable within a single language, demystifying the black-box nature of industrial AD systems.

4 Execution Model: Graph Construction and Reverse Replay

The MiniAD engine operates in two phases: dynamic graph construction during the forward pass and reverse replay during backpropagation. The overall execution pipeline is illustrated in Figure 1.

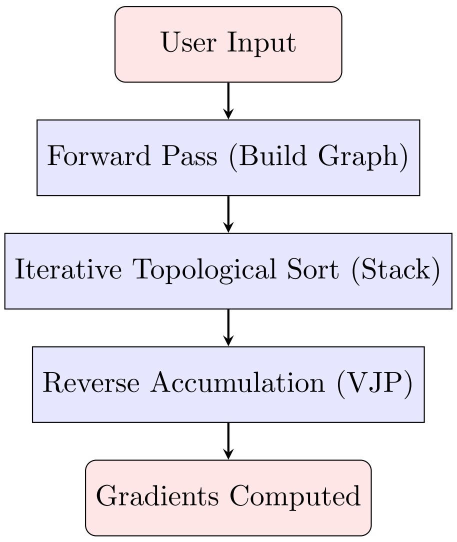
User Input
Forward Pass (Build Graph)
Iterative Topological Sort (Stack)
Reverse Accumulation (VJP)
Gradients Computed

Figure 1: Execution pipeline of the MiniAD reverse-mode automatic differentiation engine.

4.1 Phase 1: Dynamic Graph Construction

During forward execution, each overloaded operator constructs a new `Tensor` node. The node stores:

- The computed numerical result in `data`
- References to differentiable parents in `_prev`
- A closure implementing the local VJP in `_backward`

Because parent references are recorded only when `requires_grad=true`, non-differentiable branches are structurally pruned from the graph. The computational graph therefore emerges implicitly from standard Julia expressions, without requiring a separate graph-building interface.

4.2 Phase 2: Iterative Topological Scheduling

To ensure correct gradient propagation, a node must be processed only after all downstream consumers have contributed to its gradient. This is achieved through a post-order traversal implemented via an explicit stack.

The full implementation of the iterative topological sort is provided in Appendix A.2.

The traversal builds a post-order list `topo`. Reversing this list yields a valid reverse-topological order, ensuring that when `node._backward()` is executed, `node.grad` already equals the total derivative accumulated from all downstream paths.

4.3 Reverse Replay Invariant

The correctness of reverse-mode AD relies on the following invariant:

> When a node's pullback is executed, its gradient field already stores the complete $\partial L/\partial($`node`$)$ accumulated from all consumers.

This invariant is guaranteed by the post-order construction followed by reverse iteration. The output node is first seeded with $\partial L/\partial L = 1$, and successive pullbacks propagate sensitivities backward via in-place accumulation.

4.4 Stability Considerations

The stack-based implementation avoids recursion depth limits and supports arbitrarily deep computational graphs. Combined with structural pruning at the node level, this yields a memory-stable and execution-safe reverse-mode engine suitable for both neural networks and differentiable physical models.

5 Broadcast-Based Elementwise Differentiation

MiniAD integrates with Julia's broadcasting mechanism to support elementwise operations efficiently. Instead of hard-coding primitives, we use a custom `BroadcastStyle` and a backward rule registry. This allows operations like `sin.(x)` or `x .+ y` to automatically participate in the AD graph. Detailed implementation of the broadcast registry, custom style, and shape-correct gradient reduction (`unbroadcast`) is detailed in Appendix A.3.

6 Integrated Gradients for Interpretability

While reverse-mode AD provides local sensitivity information, raw gradients often fail to capture feature importance in nonlinear models due to gradient saturation. To obtain a more faithful attribution measure, we implement *Integrated Gradients* (IG), which computes feature attributions by integrating gradients along a straight-line path from a baseline input x_0 to the target input x.

For a scalar-valued model F, the i-th attribution is defined as:

$$\mathrm{IG}_i(x) = (x_i - x_{0,i}) \int_0^1 \frac{\partial F\big(x_0 + \alpha(x - x_0)\big)}{\partial x_i}\, d\alpha. \tag{3}$$

In practice, this integral is approximated using a Riemann sum with m steps:

$$\mathrm{IG}(x) \approx (x - x_0) \odot \frac{1}{m} \sum_{k=1}^{m} \nabla_x F\left(x_0 + \frac{k}{m}(x - x_0)\right), \tag{4}$$

where $\odot$ denotes elementwise multiplication.

Our implementation supports both AD-based gradients and finite differences, ensuring applicability to both differentiable tensors and black-box functions. The full code listing and design details (such as 2D alignment and gradient resetting) are discussed in Appendix A.4.

6.1 Interpretability Across Domains

Because the IG implementation operates purely on gradients, it can be applied uniformly to:

- Neural networks built from MiniAD layers
- Differentiable physical models expressed using Tensor operations
- Scalar black-box functions via finite differences

This unifies sensitivity analysis across machine learning and physics-based simulations within a single computational framework.

7 Numerical Experiments and Case Studies

We demonstrate the efficacy of our from-scratch AD and IG implementation through two case studies: optimizing a physics-based projectile model and interpreting a neural network classifier. These experiments validate that the engine correctly handles gradient propagation for both explicit physical equations and learned black-box functions.

7.1 Case Study I: Physics-Based Model — Robotic Projectile Motion

This case study utilizes the AD engine in two distinct ways: first for parameter optimization (Gradient Descent), and second for model interpretability (Integrated Gradients).

7.1.1 Optimization of Trajectory

The goal is to optimize the launch velocity v and angle θ of a projectile to hit a target hoop at coordinates $(4.0, 3.1)$. The loss function is defined as the squared Euclidean distance between the projectile's position at the target x-coordinate and the hoop's height:

$$L(v, \theta) = (y_{\text{pred}}(v, \theta) - h_{\text{hoop}})^2$$

Using our AD engine, we compute ∇L and perform gradient descent. Figure 2 shows the optimization progress from an initial failing trajectory to a successful shot.

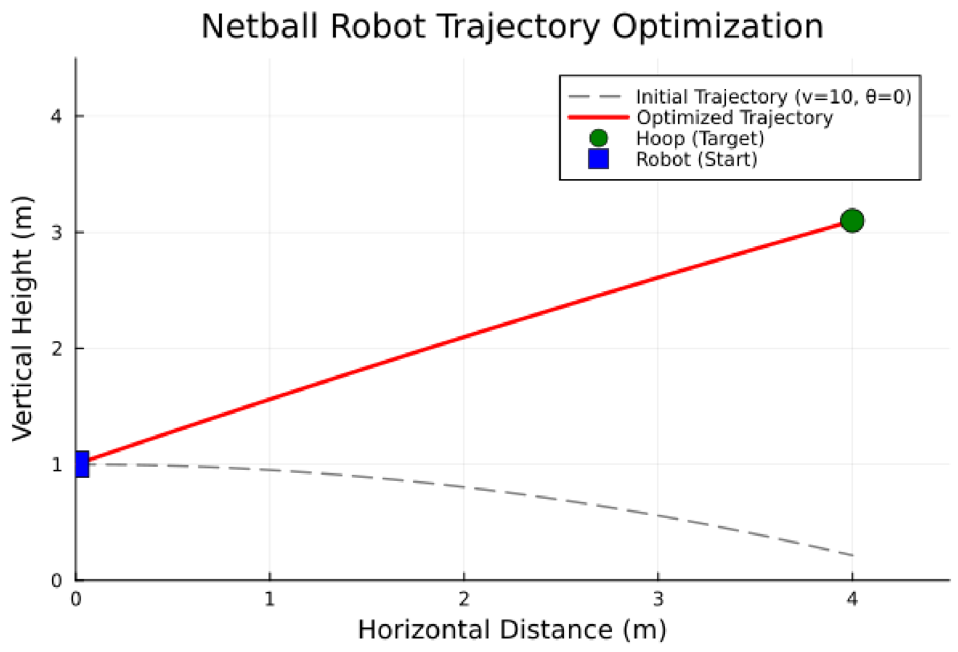

Figure 2: Trajectory optimization using reverse-mode AD. The red line shows the optimized path hitting the target hoop.

7.1.2 Explainability with IG

We apply Integrated Gradients to understand the sensitivity of the final trajectory height with respect to the launch parameters. Figure 3 illustrates the attribution scores, indicating which parameter (velocity or angle) had a more significant impact on the final outcome relative to the baseline.

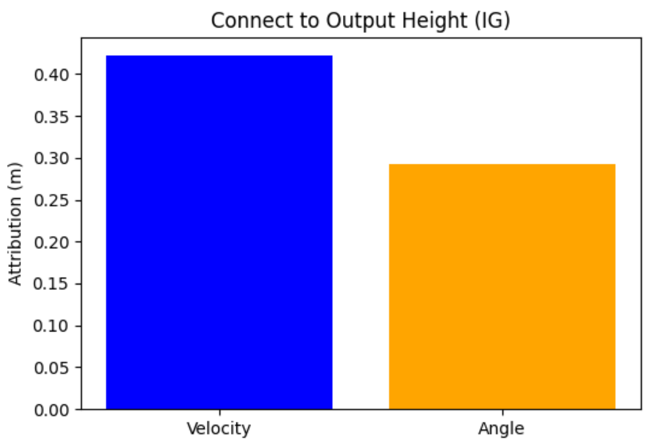

Figure 3: Integrated Gradients attribution for the Netball model, showing the contribution of velocity and angle to the result.

7.2 Case Study II: Neural Model — Feature Importance in Iris Classification

We train a Multi-Layer Perceptron (MLP) on the Iris dataset [5] to demonstrate the engine's capability in a standard deep learning context. The network consists of input, hidden, and output layers with ReLU activations, trained using Cross-Entropy loss.

7.2.1 Training Performance

The model is trained for 100 epochs using SGD. The loss curve in Figure 4 confirms the correct implementation of the backward pass through the composition of linear and non-linear layers.

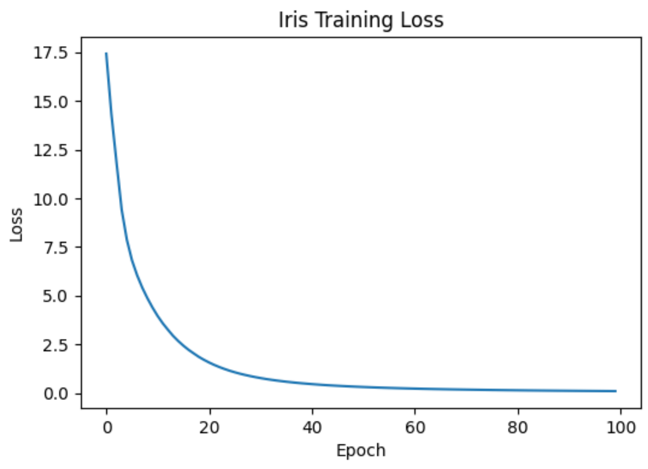

Figure 4: Training loss for the MLP on the Iris dataset, validating the AD engine's stability.

7.2.2 Feature Attribution

Using IG, we analyze the importance of the four input features (Sepal/Petal Length/Width) for a specific prediction (e.g., Setosa). Figure 5 highlights the most influential features driving the model's decision, providing transparency to the "black box" neural network.

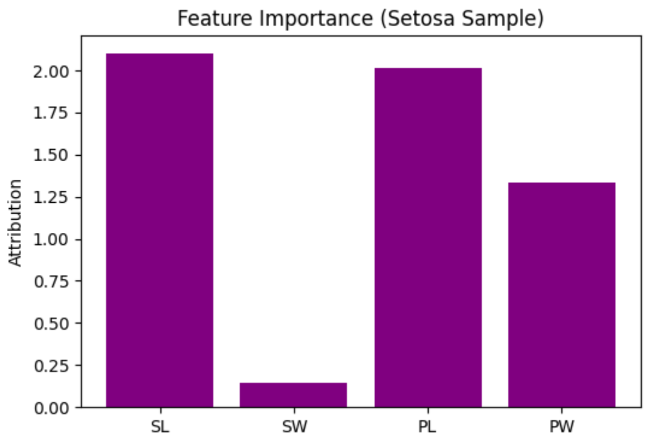

Figure 5: Feature importance attribution for an Iris Setosa prediction.

8 Conclusion

This work presented a minimalist yet functional implementation of Reverse-Mode Automatic Differentiation and Integrated Gradients in Julia. By decomposing programs into computational graphs of primitive operations, we demonstrated how exact gradients can be computed efficiently. The application of this engine to both a physics-based optimization problem and a neural network classification task highlights the universality of the approach. The project serves as a pedagogical bridge between the theoretical foundations of differentiable programming and its practical application in model interpretability.

References

[1] Griewank, A., & Walther, A. (2008). Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM.

[2] Sundararajan, M., Taly, A., & Yan, Q. (2017). Axiomatic attribution for deep networks. In International Conference on Machine Learning (pp. 3319-3328). PMLR.

[3] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2017). Automatic differentiation in machine learning: a survey. Journal of Machine Learning Research, 18(153), 1-43.

[4] Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. SIAM review, 59(1), 65-98.

[5] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. Annals of eugenics, 7(2), 179-188.

[6] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Advances in neural information processing systems (pp. 8026-8037).

A Appendix

A.1 Algorithms

Algorithm 1 Reverse-Mode AD via Iterative Topological Sort

```
Require: Output node $y$ (scalar Tensor)
Ensure: Gradients stored in grad fields of all reachable nodes
 1: Phase 1: Iterative Topological Sort
 2: topo $\leftarrow$ Empty List
 3: visited $\leftarrow$ Empty Set
 4: stack $\leftarrow [(y, \textbf{false})]$                ▷ Tuple: (node, processed_flag)
 5: while stack is not empty do
 6:     $(v, \text{processed}) \leftarrow$ Pop(stack)
 7:     if $v \notin$ visited then
 8:         if processed then
 9:             Push(visited, $v$)
10:             Push(topo, $v$)                         ▷ Post-order insertion
11:         else
12:             Push(stack, $(v, \textbf{true})$)
13:             for all $parent \in v.\_prev$ do
14:                 if $parent \notin$ visited then
15:                     Push(stack, $(parent, \textbf{false})$)
16:                 end if
17:             end for
18:         end if
19:     end if
20: end while
21: Phase 2: Reverse Accumulation
22: $y$.grad $\leftarrow 1$
23: for all $v \in$ Reverse(topo) do
24:     $v$._backward()                                 ▷ Executes VJP closure
25: end for
```

Algorithm 2 Integrated Gradients using Reverse-Mode AD

```
Require: Model $F(\cdot)$, input $x$, baseline $x'$, steps $m$, optional target index
Ensure: Attribution vector IG($x$)
 1: $\Delta \leftarrow x - x', \quad G \leftarrow 0$
 2: for $k = 1$ to $m$ do
 3:     $\alpha \leftarrow k/m$
 4:     $z \leftarrow x' + \alpha\Delta$
 5:     Wrap $z$ as input Tensor and compute scalar output $y$
 6:     Backward($y$) to obtain $\nabla_x F(z)$
 7:     $G \leftarrow G + \nabla_x F(z)$
 8: end for
 9: $\bar{G} \leftarrow G/m$
10: $\text{IG}(x) \leftarrow \Delta \odot \bar{G}$
```

A.2 Implementation of Iterative Topological Sort

The iterative topological sort ensures stack-safety for deep graphs.

Listing 2: Iterative topological traversal and reverse replay

```julia
function backward_iterative(root::Tensor; init_grad=nothing)
    if !root.requires_grad
        return nothing
    end

    topo = Tensor[]
    visited = Set{Tensor}()
    stack = [(root, false)]

    while !isempty(stack)
        curr, processed = pop!(stack)
        if curr in visited
            continue
        end

        if processed
            push!(visited, curr)
            push!(topo, curr)
        else
            push!(stack, (curr, true))
            for parent in curr._prev
                if parent ∉ visited
                    push!(stack, (parent, false))
                end
            end
        end
    end

    root.grad .= (init_grad === nothing ? 1.0 : init_grad)

    for node in reverse(topo)
        node._backward()
    end
end
```

A.3 Implementation of Broadcast Differentiation

While matrix multiplication and reduction operators are implemented explicitly, most elementwise operations in MiniAD are handled through Julia's broadcasting mechanism. Instead of defining separate forward and backward logic for each primitive (e.g., addition, multiplication, trigonometric functions), we integrate directly with Julia's broadcast infrastructure and implement a modular backward rule registry.

A.3.1 Custom Broadcast Style

MiniAD defines a custom `TensorStyle` to intercept broadcasted operations:

Listing 3: Custom broadcast style definition

```julia
struct TensorStyle <: BroadcastStyle end
Base.BroadcastStyle(::Type{Tensor}) = TensorStyle()
Base.BroadcastStyle(::TensorStyle, ::BroadcastStyle) = TensorStyle()
Base.BroadcastStyle(::BroadcastStyle, ::TensorStyle) = TensorStyle()

broadcastable(t::Tensor) = t
```

This allows expressions such as `a .+ b`, `sin.(x)`, or `exp.(x)` to construct differentiable graph nodes automatically.

A.3.2 Backward Rule Registry

Rather than embedding gradient logic inside the broadcast dispatcher, we maintain a registry mapping primitive functions to their backward rules:

Listing 4: Broadcast backward rule registry

```julia
const BROADCAST_RULES = IdDict{Any,Function}()

function register_broadcast_rule!(f, rule::Function)
    BROADCAST_RULES[f] = rule
end
```

When a broadcasted operation is executed, MiniAD:

1. Extracts raw data from `Tensor` arguments
2. Performs the forward broadcast computation
3. Constructs a new `Tensor` node
4. Looks up the corresponding backward rule from the registry
5. Attaches the rule as the node's `_backward` closure

If no rule is registered, an explicit error is raised, preventing silent gradient failures.

A.3.3 Shape-Correct Gradient Reduction: `unbroadcast`

Broadcasted operations may expand singleton dimensions. During backpropagation, gradients must therefore be reduced back to the original operand shape. This is handled by the `unbroadcast` operator:

Listing 5: Gradient shape reduction for broadcasted ops

```julia
function unbroadcast(out_grad::AbstractArray, orig_size::Tuple)
    g = out_grad

    while ndims(g) > length(orig_size)
        g = sum(g, dims=1)
    end

    for d in 1:length(orig_size)
        if orig_size[d] == 1 && size(g, d) != 1
            g = sum(g, dims=d)
        end
    end

    return reshape(g, orig_size)
end
```

This ensures that the gradient passed back to each operand matches its original dimensions, preserving consistency across scalar, vector, and matrix broadcasts.

A.3.4 Example: Elementwise Multiplication

For example, the broadcast rule for elementwise multiplication is registered as:

```
register_broadcast_rule!(*, (out, res, args...) -> begin
    x, y = args
    if x isa Tensor && x.requires_grad
        val_y = y isa Tensor ? y.data : y
        x.grad .+= unbroadcast(out.grad .* val_y, size(x.data))
    end
    if y isa Tensor && y.requires_grad
        val_x = x isa Tensor ? x.data : x
        y.grad .+= unbroadcast(out.grad .* val_x, size(y.data))
    end
end)
```

The backward rule directly implements the local derivative

$$\frac{\partial(x \cdot y)}{\partial x} = y, \quad \frac{\partial(x \cdot y)}{\partial y} = x,$$

while ensuring correct dimensional reduction.

A.3.5 Extensibility

This registry-based design cleanly separates forward semantics from backward logic. New differentiable primitives can be added without modifying the core AD engine—only a corresponding broadcast rule needs to be registered.

Compared to monolithic AD implementations, this approach provides a modular and extensible differentiation framework while remaining minimal in code complexity.

A.4 Details of Integrated Gradients

A.4.1 Implementation in MiniAD

The implementation follows the definition directly:

Listing 6: Integrated Gradients implementation in MiniAD

```
function integrated_gradients(model, input, baseline;
    steps::Int=50, target=nothing, method=:ad)

    x = _align2d(input)
    x0 = _align2d(baseline)

    diff = x .- x0
    total_grads = zeros(size(x))

    for s in 1:steps
        α = s / steps
        z = x0 .+ α .* diff

        if method == :ad
            xt = Tensor(z, _op="Input")
            out = model(xt)
            y = to_scalar(out; target=target)

            if model isa Chain
                zero_grad!(model)
            end

            xt.grad .= 0.0
            y.grad .= 0.0

            backward(y)
            total_grads .+= xt.grad

        elseif method == :fd
            g, _ = grad_fd(model, copy(z))
            total_grads .+= g
        end
    end

    avg_grads = total_grads ./ steps
    return diff .* avg_grads
end
```

A.4.2 Key Design Aspects

2D Alignment. Inputs and baselines are internally normalized to 2D arrays to maintain consistency with the Tensor abstraction. This avoids special-case handling for scalars and vectors.

Dual Gradient Modes. The implementation supports two gradient computation modes:

- `:ad` — Uses the reverse-mode AD engine.
- `:fd` — Uses finite differences for scalar-valued black-box functions.

This allows attribution analysis even when the model is not constructed using MiniAD tensors.

Gradient Resetting. Before each backward pass, gradients of the input tensor (and model parameters, if applicable) are reset. This prevents cross-step contamination during the Riemann approximation.

Completeness Property. The implementation satisfies the Completeness Axiom:

$$\sum_i \mathrm{IG}_i(x) \approx F(x) - F(x_0),$$

up to numerical approximation error from discretization. This provides a consistency check for attribution correctness.

A.5 Member Contributions

- **Cao Yuan**: Proposed the core idea and integrated the code.
- **Liu Fei**: Participated in the toy example development and case optimization.
- **Jin Xuan**: Implemented complex main operators, conducted case verification, and wrote the paper.
- **Gao Jiaxuan**: Responsible for operator and method optimization.
- **Nan Jinyao**: Responsible for application case development, code integration, and paper writing.