



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 13, 2025

Declaration: We understand what plagiarism is and have ensured We, Group 6, did not plagiarise for this assignment. We declare that our submission for CA1 Synthesis Essay is our own work. This assignment is in partial fulfilment of the requirements for the module EE5311.

Declaration of AI-generated material: During the preparation of this work, the author used generative AI tools (Google Gemini) to assist in LaTeX code generation (specifically for TikZ diagrams) and linguistic polishing. The author has reviewed and edited the content as needed and takes full responsibility for the content of the publication.

Abstract

This work presents a lightweight, ground-up implementation of a reverse-mode automatic differentiation (AD) engine in Julia. Centered on a `Tensor` abstraction, the framework leverages operator overloading and closure-based pullbacks to compute Vector-Jacobian Products (VJPs) across dynamic computational graphs, with DFS-based topological sorting ensuring robust gradient scheduling. Beyond core AD, we implement Integrated Gradients (IG) to provide equitable feature attribution for both deep learning and differentiable physical simulations. The resulting pipeline demonstrates a unified approach to sensitivity and attribution analysis, effectively bridging the gap between low-level differentiable primitives and high-level interpretability in scientific computing. Our source code is open-source and available at: https://github.com/nanjinyao/EE5311_CA1_AD_IG

1 Introduction

Interpretability is critical not only for deep learning but also for physical systems governed by differentiable laws. To bridge these domains, this work presents a lightweight reverse-mode automatic differentiation (AD) engine implemented from first principles in Julia [4].

The framework is centered on a `Tensor` abstraction that dynamically constructs computational graphs via operator overloading. By utilizing closure-based *pullbacks* and Depth-First Search (DFS) topological scheduling, the engine efficiently computes Vector-Jacobian Products (VJPs) without the overhead of explicit Jacobian matrices.

Building on this foundation, we integrate *Integrated Gradients* (IG) as a unified attribution mechanism. Experiments on differentiable physical simulations (e.g., projectile motion) demonstrate that this pipeline effectively quantifies parameter sensitivity. This approach highlights how gradients serve as a universal language, connecting low-level differentiable primitives to high-level physical reasoning.

2 Core Principle: Chain Rule and VJP

The mathematical foundation of reverse-mode AD is the **Chain Rule** for multi-variable composite functions. In a computational graph, if a scalar output L (usually the loss) depends on a node u through several downstream nodes v_i , the gradient of L with respect to u is given by the sum of contributions from all paths:

$$\frac{\partial L}{\partial u} = \sum_i \frac{\partial L}{\partial v_i} \frac{\partial v_i}{\partial u} \quad (1)$$

In our implementation, we do not explicitly construct large Jacobian matrices. Instead, we compute the **Vector-Jacobian Product (VJP)**.

- **Forward Pass:** Compute and store the output $v = f(u)$.
- **Backward Pass:** Receive the downstream gradient $\nabla_v L$, multiply it by the local derivative $\frac{\partial v}{\partial u}$, and **accumulate** it into the gradient of u .

The accumulation (`.+=`) is crucial: if a variable is used in multiple operations (fan-out > 1), its total gradient must be the sum of gradients from all its consumers.

3 Core Abstraction: The Tensor Struct

The engine is built around a mutable `Tensor` object. Each `Tensor` acts as a node in a dynamic computational graph, aware of its data, its gradient, and its ancestors.

Listing 1: The actual Tensor implementation from ad_v7.ipynb

```

1  mutable struct Tensor
2      data::Array{Float64}           # Numerical values
3      grad::Array{Float64}          # Accumulated gradients
4      _backward::Function          # Pullback closure
5      _prev::Set{Tensor}           # Parent nodes for graph traversal
6      op::String                  # Debugging label
7      requires_grad::Bool         # Gradient tracking flag
8
9      function Tensor(data; _children=(), _op="", requires_grad=true)
10         if data isa Number
11             d = reshape([Float64(data)], 1, 1)
12         else
13             arr = convert(Array{Float64}, data)
14             d = ndims(arr) == 1 ? reshape(arr, :, 1) : arr
15         end
16         g = zeros(size(d))
17         # Structural pruning: only track parents if grad is required
18         prev = requires_grad ? Set{Tensor}(_children) :
19             Set{Tensor}()
20         new(d, g, () -> nothing, prev, _op, requires_grad)
21     end
21 end

```

The most critical design choice is the `_backward` field. It is a **closure** that captures the context of the forward operation (like the values of the operands), allowing the local backpropagation logic to be encapsulated within the node itself.

4 Algorithm Flow: Construction and Backtracking

The execution of the AD engine consists of two distinct phases:

4.1 Phase 1: Dynamic Graph Construction via Operator Overloading

To visualize the complete workflow, Figure 1 illustrates the transition from the forward construction to the backward accumulation.

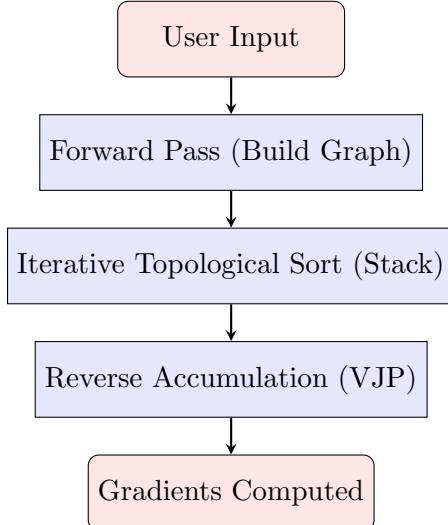


Figure 1: Flowchart of the Reverse-Mode Automatic Differentiation Engine.

In our engine, the forward pass does more than just compute values; it builds a directed acyclic graph (DAG) in real-time. When an operation like `a * b` is invoked, the engine intercepts it via operator overloading to "plant the seeds" for the backward pass. For matrix-based multiplication, the process follows three core steps:

1. **Forward Pass:** Compute the product $\mathbf{C} = \mathbf{AB}$. To ensure a uniform interface, our implementation treats even scalars as 1×1 matrices and vectors as $n \times 1$ column matrices.
2. **Structural Pruning:** A new `Tensor` node is created. To optimize memory and computation, its `_prev` set is populated only by parent nodes that have `requires_grad = true`. If neither parent requires a gradient, the child node's tracking is disabled as well.
3. **VJP (Pullback) Definition:** The engine attaches a closure that implements the Vector-Jacobian Product for matrix multiplication. Given the gradient of the loss with respect to the output $\mathbf{G} = \frac{\partial L}{\partial \mathbf{C}}$, the gradients for the inputs are computed using the following transpose rules:

$$\frac{\partial L}{\partial \mathbf{A}} = \mathbf{GB}^\top, \quad \frac{\partial L}{\partial \mathbf{B}} = \mathbf{A}^\top \mathbf{G} \quad (2)$$

Listing 2: Matrix Multiplication implementation with VJP logic from `ad_v7.ipynb`

```

1  function *(a::Tensor, b::Tensor)
2      # Determine if the output needs to track gradients
3      out_rg = a.requires_grad || b.requires_grad
4      # Structural Pruning: Only track parents that participate in
5      # backprop
6      kids = filter(x -> x.requires_grad, (a, b))
7
8      # Perform forward matrix multiplication
9      out = Tensor(a.data * b.data, _children=kids, _op="MatMul",
10                  requires_grad=out_rg)
11
12     if out_rg
13         function _backward()
14             # Matrix VJP:  $dL/dA = (dL/dC) * B'$  and  $dL/dB = A' * (dL/dC)$ 
15             # .+= ensures gradients from different paths are
16             # accumulated
17             if a.requires_grad
18                 a.grad .+= out.grad * transpose(b.data)
19             end
20             if b.requires_grad
21                 b.grad .+= transpose(a.data) * out.grad
22             end
23         end
24         out._backward = _backward
25     end
26     return out
27 end

```

4.2 Phase 2: Reverse Scheduling via Iterative Topological Sort

To ensure gradients are propagated correctly, a node must only be processed after all its downstream consumers have finished accumulating their gradients. While a recursive DFS is intuitive, it can lead to **stack overflow** in deep computational graphs. Our implementation utilizes an **iterative DFS** with an explicit stack to generate the topological order, ensuring both memory stability and execution correctness.

The Iterative DFS Mechanism: The algorithm uses a stack to manage the traversal, where each entry is a tuple `(node, processed_flag)`:

- **Discovery Phase:** When a node is first encountered (`processed_flag = false`), it is pushed back onto the stack with the flag set to `true`, followed by all its unvisited parents.
- **Post-order Completion:** When a node is popped and its flag is `true`, it means all its dependencies have been explored. The node is then added to the `topo` list. This effectively simulates post-order traversal without recursion.
- **Reversal:** Reversing this list yields a valid reverse-topological order, starting from the output and ending at the inputs.

Listing 3: Iterative topological sort and backward execution from ad_v7.ipynb

```

1  function backward_iterative(root::Tensor; init_grad=nothing)
2      # Early exit if root does not participate in gradients
3      if !root.requires_grad return nothing end
4
5      # --- Step 1: Iterative Topological Sort ---
6      topo = Tensor[]
7      visited = Set{Tensor}()
8      stack = [(root, false)] # (node, processed_flag)
9
10     while !isempty(stack)
11         curr, processed = pop!(stack)
12         if curr in visited continue end
13
14         if processed
15             push!(visited, curr)
16             push!(topo, curr) # Post-order: added after all
17                 ancestors
18         else
19             # Re-push current node with processed=true
20             push!(stack, (curr, true))
21             # Push all unvisited parents to the stack
22             for parent in curr._prev
23                 if parentnotin visited
24                     push!(stack, (parent, false))
25                 end
26             end
27         end
28
29     # --- Step 2: Gradient Seeding ---
30     if init_grad === nothing
31         root.grad .= 1.0
32     else
33         # Stability check: ensure dimensions match
34         root.grad .= (init_grad isa Number) ? Float64(init_grad) :
35             init_grad
36     end
37
38     # --- Step 3: Reverse-Topological Replay ---
39     # Replay pullbacks in reverse order (from output to input)
40     for node in reverse(topo)
41         node._backward()
42     end
43 end
```

By using a stack-based iterative approach, the engine can handle arbitrarily deep models (like very deep ResNets or long sequences in RNNs) that would otherwise crash a recursive implementation. The `reverse(topo)` loop ensures that when `node._backward()` is executed, the `node.grad` field is "fully baked"—meaning it has already received all partial gradient contributions from every downstream operation (see Algorithm 1).

5 Integrated Gradients for Interpretability

While raw gradients provide local sensitivity, they often suffer from the *saturation problem* in deep networks or complex physical models, where the gradient becomes near-zero even for important features. To address this, we implement **Integrated Gradients (IG)**, which computes attribution by integrating the gradients along a straight-line path from a baseline x_0 to the input x . The continuous definition for the i -th feature is:

$$\text{IG}_i(x) := (x_i - x_{0,i}) \int_{\alpha=0}^1 \frac{\partial F(x_0 + \alpha(x - x_0))}{\partial x_i} d\alpha \quad (3)$$

As analytical integration is typically intractable for complex differentiable models F , we approximate this integral using a **Riemann sum**. By sampling gradients at m discrete points along the path and averaging them, we obtain the computable discrete form used in our implementation (Algorithm 2):

$$\text{IG}_i(x) \approx (x_i - x_{0,i}) \times \frac{1}{m} \sum_{k=1}^m \nabla_x F \left(x_0 + \frac{k}{m}(x - x_0) \right) \quad (4)$$

5.1 Key Implementation Details

Our implementation in Julia is designed to be versatile, supporting both neural networks (via `Chain`) and arbitrary physical functions. Key features include:

- **Dual Mode Support:** The method can compute gradients using our `:ad` engine or via `:fd` (Finite Differences) for non-differentiable black-box functions.
- **Gradient Resetting:** To avoid accumulation of gradients from previous interpolation steps, we explicitly reset `xt.grad`, `out.grad`, and `y.grad` to zero in each iteration.
- **Dimensional Alignment:** Inputs and baselines are normalized to 2D arrays to ensure matrix operations remain consistent.

Listing 4: Integrated Gradients implementation from `ad_v7.ipynb`

```

1  function integrated_gradients(model ,
2      input ,
3      baseline ;
4      steps::Int=50 ,
5      target::Union{Nothing , Int}=nothing ,
6      method::Symbol=:ad ,
7      fd_eps::Float64=1e-5)
8
9      # Normalize input/baseline to 2D arrays (n x 1)
10     x = input isa Number ? reshape([Float64(input)], 1, 1) :
11         convert(Array{Float64}, input)
12     x0 = baseline isa Number ? reshape([Float64(baseline)], 1, 1) :
13         convert(Array{Float64}, baseline)
14
15     if target == nothing
16         return x0 .+ (x - x0) * (1 ./ steps)
17     else
18         target .+ (x - target) * (1 ./ steps)
19     end
20
21     if method == :ad
22         grad_fn = function (xt)
23             out = zeros(x)
24             y = model(out, xt)
25             out
26         end
27         grad_fn(x)
28         grad_fn(x0)
29         grad_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
30     else
31         grad_fn = function (xt)
32             out = zeros(x)
33             y = model(out, xt)
34             out
35         end
36         grad_fn(x)
37         grad_fn(x0)
38         grad_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
39     end
40
41     fd_fn = function (xt)
42         out = zeros(x)
43         y = model(out, xt)
44         out
45     end
46     fd_fn(x)
47     fd_fn(x0)
48     fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
49
50     fd_fn = function (xt)
51         out = zeros(x)
52         y = model(out, xt)
53         out
54     end
55     fd_fn(x)
56     fd_fn(x0)
57     fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
58
59     fd_fn = function (xt)
60         out = zeros(x)
61         y = model(out, xt)
62         out
63     end
64     fd_fn(x)
65     fd_fn(x0)
66     fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
67
68     fd_fn = function (xt)
69         out = zeros(x)
70         y = model(out, xt)
71         out
72     end
73     fd_fn(x)
74     fd_fn(x0)
75     fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
76
77     fd_fn = function (xt)
78         out = zeros(x)
79         y = model(out, xt)
80         out
81     end
82     fd_fn(x)
83     fd_fn(x0)
84     fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
85
86     fd_fn = function (xt)
87         out = zeros(x)
88         y = model(out, xt)
89         out
90     end
91     fd_fn(x)
92     fd_fn(x0)
93     fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
94
95     fd_fn = function (xt)
96         out = zeros(x)
97         y = model(out, xt)
98         out
99     end
100    fd_fn(x)
101    fd_fn(x0)
102    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
103
104    fd_fn = function (xt)
105        out = zeros(x)
106        y = model(out, xt)
107        out
108    end
109    fd_fn(x)
110    fd_fn(x0)
111    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
112
113    fd_fn = function (xt)
114        out = zeros(x)
115        y = model(out, xt)
116        out
117    end
118    fd_fn(x)
119    fd_fn(x0)
120    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
121
122    fd_fn = function (xt)
123        out = zeros(x)
124        y = model(out, xt)
125        out
126    end
127    fd_fn(x)
128    fd_fn(x0)
129    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
130
131    fd_fn = function (xt)
132        out = zeros(x)
133        y = model(out, xt)
134        out
135    end
136    fd_fn(x)
137    fd_fn(x0)
138    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
139
140    fd_fn = function (xt)
141        out = zeros(x)
142        y = model(out, xt)
143        out
144    end
145    fd_fn(x)
146    fd_fn(x0)
147    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
148
149    fd_fn = function (xt)
150        out = zeros(x)
151        y = model(out, xt)
152        out
153    end
154    fd_fn(x)
155    fd_fn(x0)
156    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
157
158    fd_fn = function (xt)
159        out = zeros(x)
160        y = model(out, xt)
161        out
162    end
163    fd_fn(x)
164    fd_fn(x0)
165    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
166
167    fd_fn = function (xt)
168        out = zeros(x)
169        y = model(out, xt)
170        out
171    end
172    fd_fn(x)
173    fd_fn(x0)
174    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
175
176    fd_fn = function (xt)
177        out = zeros(x)
178        y = model(out, xt)
179        out
180    end
181    fd_fn(x)
182    fd_fn(x0)
183    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
184
185    fd_fn = function (xt)
186        out = zeros(x)
187        y = model(out, xt)
188        out
189    end
190    fd_fn(x)
191    fd_fn(x0)
192    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
193
194    fd_fn = function (xt)
195        out = zeros(x)
196        y = model(out, xt)
197        out
198    end
199    fd_fn(x)
200    fd_fn(x0)
201    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
202
203    fd_fn = function (xt)
204        out = zeros(x)
205        y = model(out, xt)
206        out
207    end
208    fd_fn(x)
209    fd_fn(x0)
210    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
211
212    fd_fn = function (xt)
213        out = zeros(x)
214        y = model(out, xt)
215        out
216    end
217    fd_fn(x)
218    fd_fn(x0)
219    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
220
221    fd_fn = function (xt)
222        out = zeros(x)
223        y = model(out, xt)
224        out
225    end
226    fd_fn(x)
227    fd_fn(x0)
228    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
229
230    fd_fn = function (xt)
231        out = zeros(x)
232        y = model(out, xt)
233        out
234    end
235    fd_fn(x)
236    fd_fn(x0)
237    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
238
239    fd_fn = function (xt)
240        out = zeros(x)
241        y = model(out, xt)
242        out
243    end
244    fd_fn(x)
245    fd_fn(x0)
246    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
247
248    fd_fn = function (xt)
249        out = zeros(x)
250        y = model(out, xt)
251        out
252    end
253    fd_fn(x)
254    fd_fn(x0)
255    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
256
257    fd_fn = function (xt)
258        out = zeros(x)
259        y = model(out, xt)
260        out
261    end
262    fd_fn(x)
263    fd_fn(x0)
264    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
265
266    fd_fn = function (xt)
267        out = zeros(x)
268        y = model(out, xt)
269        out
270    end
271    fd_fn(x)
272    fd_fn(x0)
273    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
274
275    fd_fn = function (xt)
276        out = zeros(x)
277        y = model(out, xt)
278        out
279    end
280    fd_fn(x)
281    fd_fn(x0)
282    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
283
284    fd_fn = function (xt)
285        out = zeros(x)
286        y = model(out, xt)
287        out
288    end
289    fd_fn(x)
290    fd_fn(x0)
291    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
292
293    fd_fn = function (xt)
294        out = zeros(x)
295        y = model(out, xt)
296        out
297    end
298    fd_fn(x)
299    fd_fn(x0)
300    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
301
302    fd_fn = function (xt)
303        out = zeros(x)
304        y = model(out, xt)
305        out
306    end
307    fd_fn(x)
308    fd_fn(x0)
309    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
310
311    fd_fn = function (xt)
312        out = zeros(x)
313        y = model(out, xt)
314        out
315    end
316    fd_fn(x)
317    fd_fn(x0)
318    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
319
320    fd_fn = function (xt)
321        out = zeros(x)
322        y = model(out, xt)
323        out
324    end
325    fd_fn(x)
326    fd_fn(x0)
327    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
328
329    fd_fn = function (xt)
330        out = zeros(x)
331        y = model(out, xt)
332        out
333    end
334    fd_fn(x)
335    fd_fn(x0)
336    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
337
338    fd_fn = function (xt)
339        out = zeros(x)
340        y = model(out, xt)
341        out
342    end
343    fd_fn(x)
344    fd_fn(x0)
345    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
346
347    fd_fn = function (xt)
348        out = zeros(x)
349        y = model(out, xt)
350        out
351    end
352    fd_fn(x)
353    fd_fn(x0)
354    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
355
356    fd_fn = function (xt)
357        out = zeros(x)
358        y = model(out, xt)
359        out
360    end
361    fd_fn(x)
362    fd_fn(x0)
363    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
364
365    fd_fn = function (xt)
366        out = zeros(x)
367        y = model(out, xt)
368        out
369    end
370    fd_fn(x)
371    fd_fn(x0)
372    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
373
374    fd_fn = function (xt)
375        out = zeros(x)
376        y = model(out, xt)
377        out
378    end
379    fd_fn(x)
380    fd_fn(x0)
381    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
382
383    fd_fn = function (xt)
384        out = zeros(x)
385        y = model(out, xt)
386        out
387    end
388    fd_fn(x)
389    fd_fn(x0)
390    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
391
392    fd_fn = function (xt)
393        out = zeros(x)
394        y = model(out, xt)
395        out
396    end
397    fd_fn(x)
398    fd_fn(x0)
399    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
400
401    fd_fn = function (xt)
402        out = zeros(x)
403        y = model(out, xt)
404        out
405    end
406    fd_fn(x)
407    fd_fn(x0)
408    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
409
410    fd_fn = function (xt)
411        out = zeros(x)
412        y = model(out, xt)
413        out
414    end
415    fd_fn(x)
416    fd_fn(x0)
417    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
418
419    fd_fn = function (xt)
420        out = zeros(x)
421        y = model(out, xt)
422        out
423    end
424    fd_fn(x)
425    fd_fn(x0)
426    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
427
428    fd_fn = function (xt)
429        out = zeros(x)
430        y = model(out, xt)
431        out
432    end
433    fd_fn(x)
434    fd_fn(x0)
435    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
436
437    fd_fn = function (xt)
438        out = zeros(x)
439        y = model(out, xt)
440        out
441    end
442    fd_fn(x)
443    fd_fn(x0)
444    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
445
446    fd_fn = function (xt)
447        out = zeros(x)
448        y = model(out, xt)
449        out
450    end
451    fd_fn(x)
452    fd_fn(x0)
453    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
454
455    fd_fn = function (xt)
456        out = zeros(x)
457        y = model(out, xt)
458        out
459    end
460    fd_fn(x)
461    fd_fn(x0)
462    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
463
464    fd_fn = function (xt)
465        out = zeros(x)
466        y = model(out, xt)
467        out
468    end
469    fd_fn(x)
470    fd_fn(x0)
471    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
472
473    fd_fn = function (xt)
474        out = zeros(x)
475        y = model(out, xt)
476        out
477    end
478    fd_fn(x)
479    fd_fn(x0)
480    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
481
482    fd_fn = function (xt)
483        out = zeros(x)
484        y = model(out, xt)
485        out
486    end
487    fd_fn(x)
488    fd_fn(x0)
489    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
490
491    fd_fn = function (xt)
492        out = zeros(x)
493        y = model(out, xt)
494        out
495    end
496    fd_fn(x)
497    fd_fn(x0)
498    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
499
500    fd_fn = function (xt)
501        out = zeros(x)
502        y = model(out, xt)
503        out
504    end
505    fd_fn(x)
506    fd_fn(x0)
507    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
508
509    fd_fn = function (xt)
510        out = zeros(x)
511        y = model(out, xt)
512        out
513    end
514    fd_fn(x)
515    fd_fn(x0)
516    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
517
518    fd_fn = function (xt)
519        out = zeros(x)
520        y = model(out, xt)
521        out
522    end
523    fd_fn(x)
524    fd_fn(x0)
525    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
526
527    fd_fn = function (xt)
528        out = zeros(x)
529        y = model(out, xt)
530        out
531    end
532    fd_fn(x)
533    fd_fn(x0)
534    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
535
536    fd_fn = function (xt)
537        out = zeros(x)
538        y = model(out, xt)
539        out
540    end
541    fd_fn(x)
542    fd_fn(x0)
543    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
544
545    fd_fn = function (xt)
546        out = zeros(x)
547        y = model(out, xt)
548        out
549    end
550    fd_fn(x)
551    fd_fn(x0)
552    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
553
554    fd_fn = function (xt)
555        out = zeros(x)
556        y = model(out, xt)
557        out
558    end
559    fd_fn(x)
560    fd_fn(x0)
561    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
562
563    fd_fn = function (xt)
564        out = zeros(x)
565        y = model(out, xt)
566        out
567    end
568    fd_fn(x)
569    fd_fn(x0)
570    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
571
572    fd_fn = function (xt)
573        out = zeros(x)
574        y = model(out, xt)
575        out
576    end
577    fd_fn(x)
578    fd_fn(x0)
579    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
580
581    fd_fn = function (xt)
582        out = zeros(x)
583        y = model(out, xt)
584        out
585    end
586    fd_fn(x)
587    fd_fn(x0)
588    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
589
590    fd_fn = function (xt)
591        out = zeros(x)
592        y = model(out, xt)
593        out
594    end
595    fd_fn(x)
596    fd_fn(x0)
597    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
598
599    fd_fn = function (xt)
600        out = zeros(x)
601        y = model(out, xt)
602        out
603    end
604    fd_fn(x)
605    fd_fn(x0)
606    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
607
608    fd_fn = function (xt)
609        out = zeros(x)
610        y = model(out, xt)
611        out
612    end
613    fd_fn(x)
614    fd_fn(x0)
615    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
616
617    fd_fn = function (xt)
618        out = zeros(x)
619        y = model(out, xt)
620        out
621    end
622    fd_fn(x)
623    fd_fn(x0)
624    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
625
626    fd_fn = function (xt)
627        out = zeros(x)
628        y = model(out, xt)
629        out
630    end
631    fd_fn(x)
632    fd_fn(x0)
633    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
634
635    fd_fn = function (xt)
636        out = zeros(x)
637        y = model(out, xt)
638        out
639    end
640    fd_fn(x)
641    fd_fn(x0)
642    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
643
644    fd_fn = function (xt)
645        out = zeros(x)
646        y = model(out, xt)
647        out
648    end
649    fd_fn(x)
650    fd_fn(x0)
651    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
652
653    fd_fn = function (xt)
654        out = zeros(x)
655        y = model(out, xt)
656        out
657    end
658    fd_fn(x)
659    fd_fn(x0)
660    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
661
662    fd_fn = function (xt)
663        out = zeros(x)
664        y = model(out, xt)
665        out
666    end
667    fd_fn(x)
668    fd_fn(x0)
669    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
670
671    fd_fn = function (xt)
672        out = zeros(x)
673        y = model(out, xt)
674        out
675    end
676    fd_fn(x)
677    fd_fn(x0)
678    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
679
680    fd_fn = function (xt)
681        out = zeros(x)
682        y = model(out, xt)
683        out
684    end
685    fd_fn(x)
686    fd_fn(x0)
687    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
688
689    fd_fn = function (xt)
690        out = zeros(x)
691        y = model(out, xt)
692        out
693    end
694    fd_fn(x)
695    fd_fn(x0)
696    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
697
698    fd_fn = function (xt)
699        out = zeros(x)
700        y = model(out, xt)
701        out
702    end
703    fd_fn(x)
704    fd_fn(x0)
705    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
706
707    fd_fn = function (xt)
708        out = zeros(x)
709        y = model(out, xt)
710        out
711    end
712    fd_fn(x)
713    fd_fn(x0)
714    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
715
716    fd_fn = function (xt)
717        out = zeros(x)
718        y = model(out, xt)
719        out
720    end
721    fd_fn(x)
722    fd_fn(x0)
723    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
724
725    fd_fn = function (xt)
726        out = zeros(x)
727        y = model(out, xt)
728        out
729    end
730    fd_fn(x)
731    fd_fn(x0)
732    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
733
734    fd_fn = function (xt)
735        out = zeros(x)
736        y = model(out, xt)
737        out
738    end
739    fd_fn(x)
740    fd_fn(x0)
741    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
742
743    fd_fn = function (xt)
744        out = zeros(x)
745        y = model(out, xt)
746        out
747    end
748    fd_fn(x)
749    fd_fn(x0)
750    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
751
752    fd_fn = function (xt)
753        out = zeros(x)
754        y = model(out, xt)
755        out
756    end
757    fd_fn(x)
758    fd_fn(x0)
759    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
760
761    fd_fn = function (xt)
762        out = zeros(x)
763        y = model(out, xt)
764        out
765    end
766    fd_fn(x)
767    fd_fn(x0)
768    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
769
770    fd_fn = function (xt)
771        out = zeros(x)
772        y = model(out, xt)
773        out
774    end
775    fd_fn(x)
776    fd_fn(x0)
777    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
778
779    fd_fn = function (xt)
780        out = zeros(x)
781        y = model(out, xt)
782        out
783    end
784    fd_fn(x)
785    fd_fn(x0)
786    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
787
788    fd_fn = function (xt)
789        out = zeros(x)
790        y = model(out, xt)
791        out
792    end
793    fd_fn(x)
794    fd_fn(x0)
795    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
796
797    fd_fn = function (xt)
798        out = zeros(x)
799        y = model(out, xt)
800        out
801    end
802    fd_fn(x)
803    fd_fn(x0)
804    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
805
806    fd_fn = function (xt)
807        out = zeros(x)
808        y = model(out, xt)
809        out
810    end
811    fd_fn(x)
812    fd_fn(x0)
813    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
814
815    fd_fn = function (xt)
816        out = zeros(x)
817        y = model(out, xt)
818        out
819    end
820    fd_fn(x)
821    fd_fn(x0)
822    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
823
824    fd_fn = function (xt)
825        out = zeros(x)
826        y = model(out, xt)
827        out
828    end
829    fd_fn(x)
830    fd_fn(x0)
831    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
832
833    fd_fn = function (xt)
834        out = zeros(x)
835        y = model(out, xt)
836        out
837    end
838    fd_fn(x)
839    fd_fn(x0)
840    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
841
842    fd_fn = function (xt)
843        out = zeros(x)
844        y = model(out, xt)
845        out
846    end
847    fd_fn(x)
848    fd_fn(x0)
849    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
850
851    fd_fn = function (xt)
852        out = zeros(x)
853        y = model(out, xt)
854        out
855    end
856    fd_fn(x)
857    fd_fn(x0)
858    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
859
860    fd_fn = function (xt)
861        out = zeros(x)
862        y = model(out, xt)
863        out
864    end
865    fd_fn(x)
866    fd_fn(x0)
867    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
868
869    fd_fn = function (xt)
870        out = zeros(x)
871        y = model(out, xt)
872        out
873    end
874    fd_fn(x)
875    fd_fn(x0)
876    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
877
878    fd_fn = function (xt)
879        out = zeros(x)
880        y = model(out, xt)
881        out
882    end
883    fd_fn(x)
884    fd_fn(x0)
885    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
886
887    fd_fn = function (xt)
888        out = zeros(x)
889        y = model(out, xt)
890        out
891    end
892    fd_fn(x)
893    fd_fn(x0)
894    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
895
896    fd_fn = function (xt)
897        out = zeros(x)
898        y = model(out, xt)
899        out
900    end
901    fd_fn(x)
902    fd_fn(x0)
903    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
904
905    fd_fn = function (xt)
906        out = zeros(x)
907        y = model(out, xt)
908        out
909    end
910    fd_fn(x)
911    fd_fn(x0)
912    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
913
914    fd_fn = function (xt)
915        out = zeros(x)
916        y = model(out, xt)
917        out
918    end
919    fd_fn(x)
920    fd_fn(x0)
921    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
922
923    fd_fn = function (xt)
924        out = zeros(x)
925        y = model(out, xt)
926        out
927    end
928    fd_fn(x)
929    fd_fn(x0)
930    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
931
932    fd_fn = function (xt)
933        out = zeros(x)
934        y = model(out, xt)
935        out
936    end
937    fd_fn(x)
938    fd_fn(x0)
939    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
940
941    fd_fn = function (xt)
942        out = zeros(x)
943        y = model(out, xt)
944        out
945    end
946    fd_fn(x)
947    fd_fn(x0)
948    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
949
950    fd_fn = function (xt)
951        out = zeros(x)
952        y = model(out, xt)
953        out
954    end
955    fd_fn(x)
956    fd_fn(x0)
957    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
958
959    fd_fn = function (xt)
960        out = zeros(x)
961        y = model(out, xt)
962        out
963    end
964    fd_fn(x)
965    fd_fn(x0)
966    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
967
968    fd_fn = function (xt)
969        out = zeros(x)
970        y = model(out, xt)
971        out
972    end
973    fd_fn(x)
974    fd_fn(x0)
975    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
976
977    fd_fn = function (xt)
978        out = zeros(x)
979        y = model(out, xt)
980        out
981    end
982    fd_fn(x)
983    fd_fn(x0)
984    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
985
986    fd_fn = function (xt)
987        out = zeros(x)
988        y = model(out, xt)
989        out
990    end
991    fd_fn(x)
992    fd_fn(x0)
993    fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
994
995    fd_fn = function (xt)
996        out = zeros(x)
997        y = model(out, xt)
998        out
999    end
1000   fd_fn(x)
1001   fd_fn(x0)
1002   fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
1003
1004   fd_fn = function (xt)
1005       out = zeros(x)
1006       y = model(out, xt)
1007       out
1008   end
1009   fd_fn(x)
1010   fd_fn(x0)
1011   fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
1012
1013   fd_fn = function (xt)
1014       out = zeros(x)
1015       y = model(out, xt)
1016       out
1017   end
1018   fd_fn(x)
1019   fd_fn(x0)
1020   fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
1021
1022   fd_fn = function (xt)
1023       out = zeros(x)
1024       y = model(out, xt)
1025       out
1026   end
1027   fd_fn(x)
1028   fd_fn(x0)
1029   fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
1030
1031   fd_fn = function (xt)
1032       out = zeros(x)
1033       y = model(out, xt)
1034       out
1035   end
1036   fd_fn(x)
1037   fd_fn(x0)
1038   fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
1039
1040   fd_fn = function (xt)
1041       out = zeros(x)
1042       y = model(out, xt)
1043       out
1044   end
1045   fd_fn(x)
1046   fd_fn(x0)
1047   fd_fn.(x .+ (0:(steps-1)/steps)*(x-x0))
1048
1049   fd_fn = function (xt)
1050       out = zeros(x)
1051       y = model(out, xt)
1052       out
1053   end
1054   fd_fn(x)
1055   fd_fn(x0)
1056   fd_fn.(x
```

```

13     @assert size(x) == size(x0) "input and baseline must have same
14         shape"
15
16     diff = x .- x0 # Direction vector from baseline to input
17     total_grads = zeros(size(x))
18
19     # Riemann approximation of path integral
20     for s in 1:steps
21         alpha = s / steps # Interpolation factor
22         z = x0 .+ alpha .* diff # Interpolated point on the path
23
24         if method == :ad
25             # Automatic differentiation approach
26             xt = Tensor(z, _op="Input")
27             out = model(xt)
28             # Use helper to ensure a scalar output for
29             # backpropagation
30             y = to_scalar(out; target=target)
31
32             # Reset gradients before backward pass to avoid step
33             # pollution
34             if model isa Chain
35                 zero_grad!(model)
36             end
37             xt.grad .= 0.0
38             out.grad .= 0.0
39             y.grad .= 0.0
40
41             backward(y)
42             total_grads .+= xt.grad
43
44         elseif method == :fd
45             # Finite difference approach for black-box models
46             g, _ = grad_fd(model, copy(z); eps=fd_eps)
47             total_grads .+= g
48
49         else
50             error("Unknown method: $method. Use :ad or :fd.")
51         end
52     end
53
54     # Compute final attributions: (input - baseline) *
55     # average_gradient
56     avg_grads = total_grads ./ steps
57     attrs = diff .* avg_grads
58     return attrs
59
60 end

```

5.2 Completeness and Attribution

One major advantage of this implementation is that it satisfies the **Completeness Axiom**: the sum of the resulting attribution vector `attrs` will approximately equal the difference in the model's output, $F(x) - F(x_0)$. This provides a physically grounded interpretation of how much each parameter (like velocity or damping in a physical system) contributes to the final outcome.

6 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.

6.1 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).

6.1.1 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.

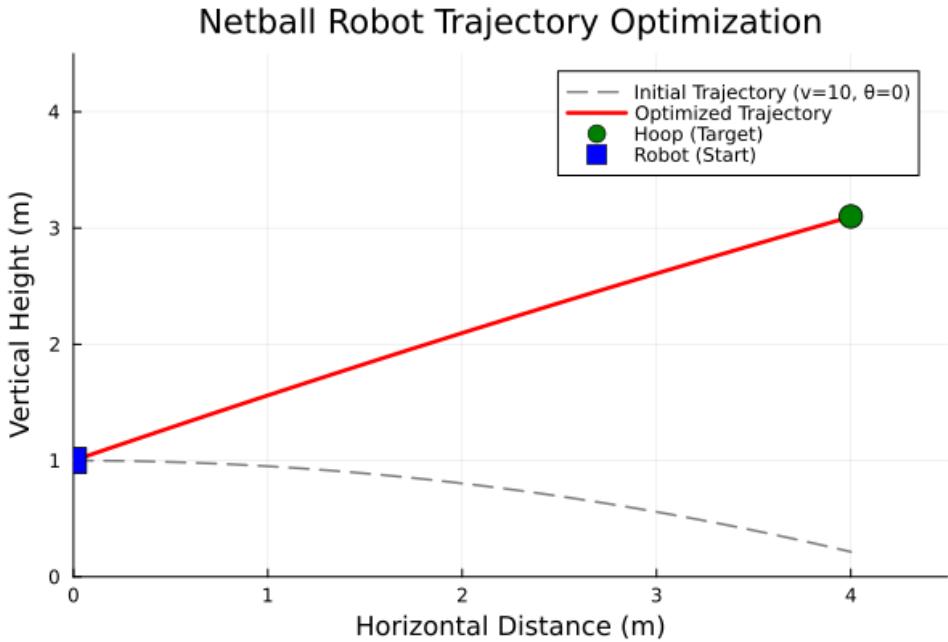


Figure 2: Trajectory optimization using reverse-mode AD. The red line shows the optimized path hitting the target hoop.

6.1.2 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.

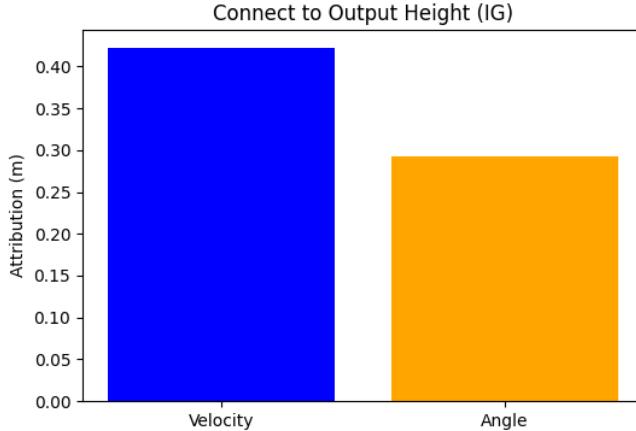


Figure 3: Integrated Gradients attribution for the Netball model, showing the contribution of velocity and angle to the result.

6.2 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.

6.2.1 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.

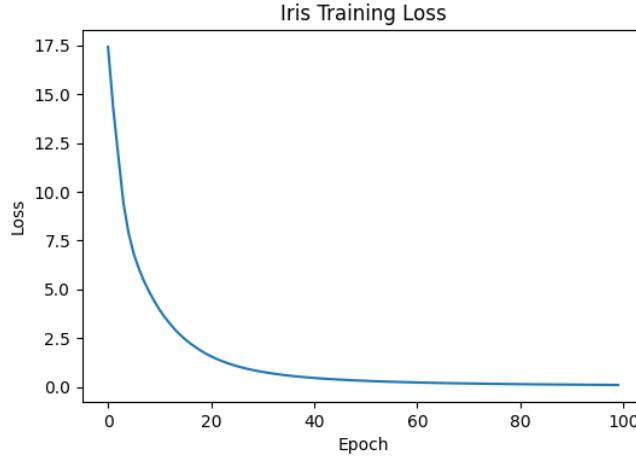


Figure 4: Training loss for the MLP on the Iris dataset, validating the AD engine’s stability.

6.2.2 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.

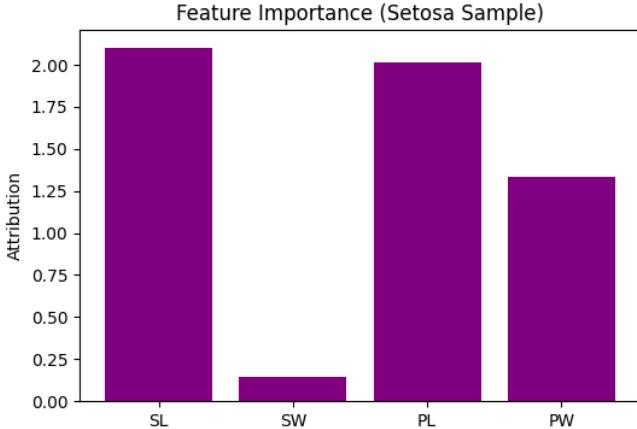


Figure 5: Feature importance attribution for an Iris Setosa prediction.

7 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.

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  $\leftarrow$   $[(y, \text{false})]$                                  $\triangleright$  Tuple: (node, processed_flag)
5: while stack is not empty do
6:    $(v, \text{processed}) \leftarrow \text{POP}(\text{stack})$ 
7:   if  $v \notin \text{visited}$  then
8:     if  $\text{processed}$  then
9:       PUSH(visited,  $v$ )
10:      PUSH(topo,  $v$ )                                 $\triangleright$  Post-order insertion
11:    else
12:      PUSH(stack,  $(v, \text{true})$ )
13:      for all  $parent \in v.\_prev$  do
14:        if  $parent \notin \text{visited}$  then
15:          PUSH(stack,  $(parent, \text{false})$ )
16:        end if
17:      end for
18:    end if
19:  end if
20: end while
21: Phase 2: Reverse Accumulation
22:  $y.\text{grad} \leftarrow 1$ 
23: for all  $v \in \text{REVERSE}(\text{topo})$  do
24:    $v.\_backward()$                                  $\triangleright$  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 $\text{IG}(x)$

```

1:  $\Delta \leftarrow x - x'$ ,  $G \leftarrow 0$ 
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 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.