IsaGrad: Verified Automatic Differentiation over Computational Graphs in Imperative HOL

Alex Raymond Hyman, Filip Smola, Mark Chevallier and Jacques D. Fleuriot

School of Informatics, University of Edinburgh, Scotland

Abstract

We present IsaGrad, a formalisation of scalar reverse automatic differentiation (RAD) over mutable reference-based computational graphs using the Imperative HOL library of Isabelle, and verify its functional correctness by an interactive separation-logic proof. We use IsaGrad as the differentiation engine for two machine learning case studies: training a multi-layer perceptron and a recurrent neural network. Leveraging Isabelle's code generation, we extract imperative-style Haskell programs where the training process is backed by our verified algorithm. To our knowledge, our implementation is the first of its kind to prove the functional correctness of an imperative, heap-based RAD algorithm. Beyond showcasing Imperative HOL's expressiveness, this work demonstrates how formal verification increases confidence in computational graphs used for gradient-based optimisation, including neural network training.

Keywords

Automatic Differentiation, Neural Networks, Isabelle, Interactive Theorem Proving, Separation Logic, Computational Graphs

1. Introduction

Automatic differentiation (AD) is a fundamental technique in machine learning (ML) that enables efficient gradient computations during optimisation tasks. Neural networks [1], which are ubiquitous within ML, rely on gradient descent [2] for training and thus rely on AD [3].

A standard imperative implementation of reverse AD is to record the forward computation as a computational graph and then traverse it backwards to accumulate gradients [3]. In dynamic frameworks such as PyTorch [4, 5], graph nodes are represented and connected by mutable references, and the reverse sweep mutates node state in order to accumulate gradients.

AI models are becoming increasingly integrated into safety-critical environments, such as autonomous vehicles [6] and medical diagnostics [7, 8]. However, they are often treated as black box models which are difficult to fully trust. While much research focuses on verifying the behaviour of trained models, trust in the latter fundamentally relies on the mathematical soundness of the training process i.e., that the computed gradient values are truly derivatives. Such guarantees are useful for more than just offline training. For instance, neural adaptive controllers of cyberphysical systems may compute or adapt from gradients online [9, 10]. By formally verifying RAD in Isabelle/HOL, we provide a rigorous guarantee that the differentiation operates as specified, thereby ruling out a potential source of incorrect gradient calculations, which is relevant given empirical evidence that popular deep-learning frameworks suffer from differentiation-related bugs and beyond [11].

Since Isabelle/HOL is a functional language that by design does not have built-in references or mutability, verifying this imperative algorithm is challenging — we use Isabelle's Imperative HOL [12] and separation logic libraries [13, 14] to model the underlying heap semantics and pointers. In IsaGrad, graph edges are represented by heap references between nodes. Hence, our imperative RAD correctness proof goes beyond just calculus: the algorithm must preserve the graph structure, handle sharing between subgraphs, and update the intended gradients correctly. Separation logic is used to

LOGICNN 2026: Workshop on Logical Methods for Neural Network Analysis, July 25, 2026, Lisbon, Portugal

A.R.Hyman@sms.ed.ac.uk (A. R. Hyman); f.smola@ed.ac.uk (F. Smola); Mark.Chevallier@ed.ac.uk (M. Chevallier); jdf@ed.ac.uk (J. D. Fleuriot)

0009-0000-2270-885X (A. R. Hyman); 0009-0003-2045-3971 (F. Smola); 0000-0001-5307-7018 (M. Chevallier); 0000-0002-6867-9836 (J. D. Fleuriot)

reason about how the stateful RAD algorithm manipulates the heap by mutating potentially shared pointers, whilst leaving others unaffected. All the lemmas shown in this paper are fully proven and mechanised in Isabelle.

In this work, we deliberately restrict our initial formalisation of computational graphs to scalar values. This allows us to isolate the complexities of imperative graph traversal, reference sharing, and in-place gradient accumulation from the additional algebraic challenges posed by matrices/tensors. Additionally, we use idealised real and rational numbers rather than machine floating point representations. As part of generating executable code we map these directly to the `Double` type in Haskell (see Section 6). This increases the trusted code base, but in the future could be eliminated by using a formalisation of IEEE-754 floating point numbers [15]. Verifying this reference-based, mutable architecture lays significant groundwork required for future work where we extend IsaGrad with matrix/tensor computations, in order to bring the framework closer to ML practice.

The primary contribution of this paper is IsaGrad, an imperative RAD algorithm over reference-based computational graphs, formally proven to correctly compute derivatives. Building upon this verified foundation, we introduce a minimalist neural network library that leverages Isabelle's code generation to produce executable, imperative-style Haskell code. We demonstrate the flexibility of this approach through two case studies, using our algorithm to train a multi-layer perceptron on a non-linear decision boundary problem, and a recurrent neural network on a next-character prediction task.

Related Work. While ML libraries and AD engines have been formalised in other interactive theorem provers, none, to our knowledge, takes an imperative-style approach over mutable computational graphs. In Rocq, De Vilhena and Pottier [16] formalise a define-by-run scalar-based RAD algorithm that calculates the derivative of a single variable using effect handlers, rather than RAD over a reference-based CG which calculates the gradient of each node all in one pass. In Lean 4, TorchLean [17] compiles PyTorch-style programs into an intermediate representation, but restricts this to a pure Static Single Assignment format. Certigrad [18] focuses on stochastic backpropagation correctness but abstracts away the referencing mechanics. Beyond AD, projects like MLCERT [19], work by Brucker and Stell [20, 21] and Bentkamp [22] verify the mathematical properties, generalisation and expressiveness of networks after training, whereas IsaGrad verifies the underlying differentiation algorithm used during the training phase itself.

Paper Organisation In Section 2, we introduce background concepts. In Section 3 we describe our formalisation of reference-based computational graphs and imperative RAD in Isabelle/HOL. Then, in Section 4, we give an overview of a specification expressed functionally, followed by Section 5 which covers our approach to proving that the algorithm over the CG satisfies it. In Section 6, we utilise the RAD algorithm to train neural networks. Finally, in Section 7 we provide our conclusion and directions for future work.

2. Background

2.1. Computational Graphs

A *computational graph* (CG) is a directed acyclic graph (DAG) [23] that captures the structure of mathematical expressions. For example, the following sequence of assignments:

$$a := 2, \quad b := 3, \quad c := a + b, \quad d := ab, \quad e := c - d, \tag{1}$$

can be represented as the CG shown in Figure 1.

The directed edges in Figure 1 convey operation dependencies, pointing from a result back to its operands. This means that nodes only point to the subgraphs relevant to their data, which is a core component of our, and PyTorch's [4, 5], approach to representing CGs. We represent pointers between nodes as references within our program, as detailed in Section 3.

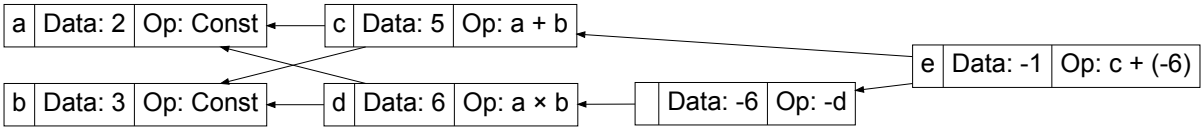

Figure 1: Computational graph for the sequence of assignments Equation 1. Each node is of the form: variable name, underlying value, and, operation. The `Op` field includes the operands as well for clarity.

CGs typically capture a set of *primitive* operations from which all other operations are derived. In IsaGrad, we select our primitive operations as addition, multiplication, division, negation and maximum. Additional operations like subtraction are defined in terms of primitives, hence, the unnamed, intermediate node of $-d$ in Figure 1. We introduce some terminology. If a node u points to a node v, we say that v is a *child* of u and that u is a *parent* of v. For example, in Figure 1, node d points to its children nodes a and b since $d := ab$. Moreover, we say that a node r is a *root* node of the CG if it has no parents i.e., no nodes depend on r, as is the case for node e in our example. Conversely, we call a node a *leaf* if it has no children.

As a CG is a DAG, it contains no loops, meaning that no variable depends on itself within the system of assignments it represents. For the purposes of this verification we assume that CGs have exactly one root node, as is the case for neural networks with a scalar loss. Additionally, we assume that every node is reachable from the root node, since unreachable nodes have no effect on the root and can thus be pruned from the CG.

2.2. Reverse Automatic Differentiation

The *gradient* of a node x in a CG with root node r is defined as the partial derivative of r with respect to x, denoted by $\frac{\partial r}{\partial x}$. Automatic differentiation [3] refers to a family of methods for computing these partial derivatives in terms of primitive operations. We focus on RAD, which performs better than forward-mode AD when the number of input nodes is much greater than the number of output/root nodes, as is the case for neural networks. The fundamental theorem which underpins the RAD algorithm is the Generalised Chain Rule [24]:

Theorem 1 (Generalised Chain Rule)**.** *Let $w = f(x_1, x_2, \dots, x_m)$ be a differentiable function of m intermediate variables, and for each $i \in \{1, \dots m\}$, let $x_i = x_i(t_1, t_2, \dots, t_n)$ be a differentiable function of n variables. Then*

$$\frac{\partial w}{\partial t_j} = \frac{\partial w}{\partial x_1}\frac{\partial x_1}{\partial t_j} + \frac{\partial w}{\partial x_2}\frac{\partial x_2}{\partial t_j} + \cdots + \frac{\partial w}{\partial x_m}\frac{\partial x_m}{\partial t_j} \tag{2}$$

for any $j \in \{1, 2, \dots, n\}$.

If we consider every input/leaf node in the CG to be constant except x, then the root r is a function of x's parents $(p_1, \dots, p_n)$. Then, via the chain rule (Equation 2), we have that

$$\frac{\partial r}{\partial x} = \frac{\partial r}{\partial p_1}\frac{\partial p_1}{\partial x} + \cdots + \frac{\partial r}{\partial p_n}\frac{\partial p_n}{\partial x} \tag{3}$$

Thus, a natural way to calculate node gradients is by operating in a topological order of the CG, where parents appear before children, incrementing gradients during the algorithm's stateful execution. The algorithm outline is:

1: Reset all the gradients of each node in the CG to 0
2: Set the root node's gradient to 1
3: `topo` ← topological ordering of the CG (parents before children)
4: **for** each node `n` ∈ `topo` **do**
5: **for** each `child` of `n` **do**
6: $\texttt{grad(child)} \leftarrow \texttt{grad(child)} + \texttt{grad(n)} \cdot \frac{\partial \texttt{n}}{\partial \texttt{child}}$

After the execution of RAD, the CG's gradients can be used to optimise the output of the root node by gradient descent, as discussed in Section 6.

2.3. Imperative HOL and Separation Logic

Although there are functional implementations of RAD done through the lens of category theory [25, 26, 27] and focused on source transformations [28], CGs are usually implemented in imperative languages, where references, mutability and effects are standard features. RAD takes advantage of references to enable sharing of subgraphs and in-place updates, caching partially computed answers to make the procedure more efficient.

The Imperative HOL framework [12] embeds constructs such as the heap and references into the functional language of Isabelle/HOL by wrapping all heap-related operations in the *Heap Monad*. This allows us to emulate a stateful, sequential program by encapsulating stateful computations within a wrapper type, implicitly carrying the mutable heap state through a sequence of operations. Imperative HOL implements Haskell-like `do` syntax [29] to abstract away monadic bind operations, allowing us to write code in a more intuitive, imperative style.

The Refinement Framework for Imperative HOL [13] formalises separation logic [14, 30], which allows us to reason about specific memory regions rather than the entire heap. This is vital to formally verifying RAD on our computational graphs, as we need to be able to alter the gradients of particular nodes whilst ensuring that other nodes remain unchanged. Relevant separation logic concepts such as Hoare triples [31] and separating conjunctions will be covered as we describe our formalisation in Subsection 4.3.

3. Implementing CGs and RAD in Isabelle

In this section, we provide an overview of our reference-based implementation of computational graphs in Isabelle.

3.1. Representation of Computational Graphs

We implement CGs using a polymorphic datatype, `'v node`, that encapsulates a node's data, gradient, and arithmetic operation. Moreover, the operation stores references to the node's children, thereby representing the directed edges of the CG. Nodes also have a boolean flag called `visited`, which is used to enable efficient topological sorting (see Subsection 3.3) and does not affect anything else. We use a mutually recursive datatype to define `node` and `operation`, enabling them to refer to each other:

```
datatype 'v node = Node (data: 'v) (grad: 'v) (oper: "'v operation") (visited: "bool")
     and 'v operation = Const | Mult "'v node ref" "'v node ref"
     | Plus "'v node ref" "'v node ref" | Div "'v node ref" "'v node ref"
     | Uminus "'v node ref" | Max "'v node ref" "'v node ref"
```

The `operation` datatype's constructors take *references* to the child nodes as arguments (rather than actual nodes), hence the argument type of `'v node ref`, where `'v` denotes a polymorphic type. However, since Imperative HOL requires `'v` to be countable, we use rational numbers as the concrete type for our CG i.e., we have `rat node` as the vertices of our graphs. For input nodes, where no operation is present, we use a nullary constructor `Const`. Each node contains an operation field which in turn allows the node to point to its children.

Using references means that multiple different nodes can share a child. The concept of a node being a child to many nodes, known as *variable sharing*, is common in CGs. Without it, the CG would effectively unroll into a tree, resulting in redundant, repeated computations. Thus, sharing children is both memory and computationally efficient.

While the `Node` constructor produces syntactically valid objects, it does not enforce semantic consistency between the data and the operation. In Subsection 3.2, we define functions that guarantee this consistency, which we verify in Subsection 4.3.

In order for the heap to be polymorphic, Imperative HOL only allows the heap to store countable datatypes [12]. This is because when a value v of type α is stored on the heap, the heap internally stores a natural number encoding of v. Note that this encoding is only used as a means for formalisation

within Isabelle, and host language features are used instead for generated code. Due to this countability constraint, we select rationals as the underlying type. In Section 6, we map rationals to Haskell's `Double` type. We could also map to Haskell's `Rational` type for a more faithful code generation at the cost of evaluation speed.

3.2. Construction of Computational Graphs

To create an initial node with some data x on the heap for our computational graph, we define the function `const_node`. It produces a reference to a node whose data is x, gradient is initialised to 0, operation is `Const` and `visited` field is initialised to `False`:

```
partial_function (heap) const_node :: "('a :: {zero, heap}) ⇒ 'a node ref Heap"
    where [code]: "const_node x = ref (Node x 0 Const False)"
```

The output of `const_node` is not of type `'a node ref` as one might expect, but instead is of `'a node ref Heap` i.e., the output is wrapped in Imperative HOL's `Heap` monad. The `Heap` monad handles potential failures of heap operations in a functional manner. Under the hood, `'a Heap` has the type `heap ⇒ ('a × heap) option`, which can be understood as an action which takes a heap as input, and returns a value of type `'a` and a (potentially altered) `heap`, wrapped in the `option` monad, in case a failure has occurred. It is similar to Haskell's `ST` [32] monad.

Reference-based `Heap` monad functions may not terminate on certain inputs e.g., if a function recurses on cyclic references. The `partial_function (heap)` keyword is used to define reference-based functions. The technical details involving their termination are covered by Krauss [33].

We create nodes by taking existing `node` references and performing arithmetic operations on them. See, for instance, our `mult_nodes` function:

```
partial_function (heap) mult_nodes::"('a :: {zero,times,heap}) node ref ⇒ 'a node ref ⇒ 'a node ref Heap"
  where [code]: "mult_nodes rx ry =
  do {x ← !rx;
      y ← !ry;
      ref (Node (data x * data y) 0 (Mult rx ry) False)}"
```

This function accepts two node references `rx` and `ry` and creates a new one with them as children bound together via a `Mult` operation. It dereferences both, via the `!` operator, to set the new reference's data as the product of the children's data; initialises the gradient to 0; records the operation on children; and finally, initialises `visited` to `False`. The other primitive functions are defined in the same way, with data fields and operations tailored to the respective primitive.

3.3. Topological Sorting of References

As mentioned in Subsection 2.2, performing RAD requires a topological sort of the CG nodes i.e., an ordering where parents always precede their children. A topological ordering is guaranteed to exist for any DAG; Section 4 explains how we ensure our data structures maintain this DAG property.

We implement a standard, reference-based imperative topological sort algorithm via a depth-first search [23]. Given a node reference `r`, the algorithm recursively goes through unvisited children, marking them via a mutable `visited` flag, and prepending them to a list upon exiting the recursion. This ensures parents always appear before their children. An overview of the Isabelle implementation, `topo_sort_aux`, is provided in Appendix A. We define a function `topo_sort_refs r` that calls this auxiliary function and subsequently resets all `visited` flags to `False`, leaving the CG's state unchanged.

3.4. Reverse Automatic Differentiation in Isabelle

In Subsection 2.2, Equation 3 shows that to calculate $\frac{\partial r}{\partial x}$ we simply need to know the gradients of x's parents, allowing us to increment x's gradient field by $\frac{\partial r}{\partial p_i}\frac{\partial p_i}{\partial x}$ for each parent p_i.

In the listing below, to perform RAD, `heap_cg_grad` first obtains `topo`, a topological ordering of the CG (line 3). All the nodes' gradients are then reset to zero (line 4) and the root's gradient is set

to one (line 5). Finally we iterate over `topo` (line 6), where at each step we push the current node's gradient to each of its children, multiplied by the local partial derivative rule. The pseudocode given in Subsection 2.2 closely corresponds to our Imperative HOL implementation:

```
1 definition heap_cg_grad :: "('a :: {heap,linorder,field}) node ref ⇒ unit Heap"
2   where "heap_cg_grad root = do {
3     topo ← topo_sort_refs root;
4     map_unit_monad (set_grad 0) topo;
5     set_grad 1 root;
6     map_unit_monad backprop_step topo}"
```

The return type is `unit Heap` because there is no meaningful return value, although, the CG's state on the heap has been altered (its gradients have been changed). We have a custom function `map_unit_monad` that iterates a stateful Heap monad operation over a list — on line 4 it zeroes the gradient of every node in `topo`. On line 6, it is used again to apply `backprop_step` over each node in `topo`. The function `backprop_step` propagates the current node's gradient to its children, adding to each child the product of the current gradient and the corresponding local partial derivative. We show the definition for `Const` and `Mult`, with other operations only differing in number of children and the particular derivative rule used.

```
fun backprop_step :: "('a::{heap,field,linorder}) node ref ⇒ unit Heap"
  where "backprop_step r = do {
      v ← !r;
      case oper v of
      Const ⇒ return ()
    | Mult c1 c2 ⇒ do {
        v1 ← !c1;
        v2 ← !c2;
        add_grad c1 (data v2 * (grad v));
        add_grad c2 (data v1 * (grad v)) }
    | ... (*other cases analogous*)}"
```

This function dereferences `r` to get the underlying `node` object `v`. There are no children to update in the case of a constant node. In the case of multiplication, child `c1`'s gradient is incremented by the data of the other child, `data v2`, multiplied by `grad v`. The factor of `data v2` comes from the multiplication derivative rule $\frac{\partial}{\partial c_1}(r) = \frac{\partial}{\partial c_1}(c_1 c_2) = c_2$. A symmetric action is performed on child `c2`.

Note that when `data v2` $= 0$, the derivative of `Div c1 c2` is not defined. However, `backprop_step` will not throw an error, since division by zero is defined to be zero in Isabelle. Moreover, the derivative $\frac{\partial \max(x,y)}{\partial x} = 1$ if $y < x$ and equal to 0 if $x < y$, but is undefined when $x = y$. In such a case, we have defined it to be 0, although various machine learning libraries choose different subgradients [34], such as $1/2$, and there is no general consensus [35, 36]. During verification of `heap_cg_grad`, in Section 4, we assume that no divisions by zero or maxima of equal arguments occur in the CG.

4. Specification of Computational Graphs

We verify our imperative CG data structure and RAD algorithm by creating a functional specification of their expected behaviour and then showing: (A) the imperative algorithm and data structure match the specification and (B) the specification is correct with respect to the relevant Isabelle/HOL analysis library definitions. This is similar to the "Simple Refinement" methodology for Imperative HOL [13].

4.1. Stack CGs

We introduce a functional specification called a *stack CG* that is guaranteed to be a DAG by construction. By refining this specification, we ensure that we only permit imperative data structures which are DAGs, and thus CGs. We first define operations for the stack CG that mirror those of the heap CG:

```
datatype 'a stack_op = StConst 'a | StMult nat nat | StPlus nat nat |
                       StDiv nat nat | StUminus nat | StMax nat nat
```

The constructor `StConst` takes a value of type `'a` whilst the other operations take indices, which act as pointers to the children, as arguments. A stack CG is a list of operations that represents all the assignments defining a CG. For example, the sequence of assignments (1) in Subsection 2.1 can be represented as the following stack CG:

$$[\underbrace{\texttt{StConst 2}}_{a:=2}, \underbrace{\texttt{StConst 3}}_{b:=3}, \underbrace{\texttt{StPlus 0 1}}_{c:=a+b}, \underbrace{\texttt{StMult 0 1}}_{d:=ab}, \underbrace{\texttt{StUminus 3}}_{-d}, \underbrace{\texttt{StPlus 2 4}}_{e:=c-d\equiv c+(-d)}] \tag{4}$$

Crucially, we specify that lists of stack operations are stack CGs if and only if they satisfy a correctness condition: the ith element in the `stack_op` list may only contain indices strictly less than i. This implies that a stack CG cannot contain cycles, hence, every stack CG represents a DAG. Moreover, any function which recurses on a stack operation's children/indices will necessarily terminate.

Each node in a *heap CG* stores its data and gradient values in mutable fields which are not necessarily always correct. Stack CGs, however, do not store the data or gradient of any node, but rather have functions mathematically specifying these. Efficiency does not matter here: this is just a specification which we aim to prove correct.

We define `stack_cg_data cg i` which recursively calculates the data at index `i` in the stack CG `cg` by applying each element's operation to the data of `i`'s children. This is relatively straightforward and will not be shown here due to space considerations. It can be illustrated using Equation 4 as an example: `stack_cg_data cg 5 = stack_cg_data cg 2 + stack_cg_data cg 4`, which then requires more recursive calls until the constant operations are reached.

Similarly, we define a function called `stack_cg_grad cg j i` which calculates the partial derivative of node j with respect to node i. Because it evaluates the derivative of an intermediate node with respect to a fixed independent variable, this function performs forward automatic differentiation (FAD). We show its definition for the constant and multiplication cases, with the other ones being analogous.

```
function stack_cg_grad :: "('a :: {field,linorder}) stack_cg ⇒ nat ⇒ nat ⇒ 'a"
  where "stack_cg_grad cg j i =
  ( if j < length_cg cg then
      if i = j then 1 else (
        case nth_cg cg j of
          StConst v ⇒ 0
        | StMult x y ⇒ stack_cg_grad cg x i * stack_cg_data cg y +
                       stack_cg_data cg x   * stack_cg_grad cg y i
        | ... (* other operations *) )
    else undefined)"
```

The function first checks that j is a valid index, i.e., less than the length of the stack CG. If $i = j$, then $\frac{\partial i}{\partial j} = 1$. Otherwise, it performs a case split on the operation of node j. If node j is a constant node, then it does not depend at all on node i and the derivative is 0. If the jth node is a multiplication node, the function applies the product rule. Analogous derivative rules are applied for each other primitive.

Note that the function implements an application of the chain rule from the inputs to the outputs, which is the essence of FAD, and is relatively straightforward to prove correct. The following Isabelle lemma formalises this correctness, proving that `stack_cg_grad` accurately calculates derivatives:

```
lemma stack_cg_grad_derivative:
 fixes i j :: nat and cg :: "real stack_cg" and x :: real
 assumes "j < length_cg cg" and "i ≤ j" and "safe_deriv_update i x cg"
 shows "((λx. stack_cg_data (update_cg i x cg) j)
        has_real_derivative
        stack_cg_grad (update_cg i x cg) j i) (at x)"
```

The function `update_cg i x cg` changes the `i`th element of `cg` to `StConst x`. The lemma views the data of node j as a function of x, and shows this function has a derivative which is equal to the output of `stack_cg_grad`. As discussed at the end of Subsection 3.4, our division and maximum operators are not fully differentiable. The assumption `safe_deriv_update i x cg` ensures that by making the ith node x, we are not causing a division by zero or a maximum of equal arguments in any node.

Due to space considerations we only provide a sketch of our proof for the above lemma. We wish to show that the statement holds for $i \leq j$ and proceed by strong induction on j. The statement is trivial for $i = j$. When $i < j$, we perform a case split on the operation of the jth node. For the case where node j is `StMult x1 x2`, using our inductive hypothesis, we know that `stack_cg_grad cg x1 i` and `stack_cg_grad cg x2 i` are the derivatives $\frac{\partial x1}{\partial i}$ and $\frac{\partial x2}{\partial i}$. Using that in conjunction with the product rule shows that `stack_cg_grad`'s returned value gives the derivative $\frac{\partial j}{\partial i}$. The other cases are similar.

4.2. Bridging the Gap between Implementation and Specification of AD

Proving that our implementation `heap_cg_grad` refines its specification, is effectively tantamount to proving that RAD and FAD both compute the same derivatives. The way to bridge this gap is to show that the sum $\sum_{j=1}^{n} \text{grad}(p_j)\frac{\partial p_j}{\partial i}$ (where p_j are the parents of i) is equivalent to the output of `stack_cg_grad cg (length_cg cg - 1) i`.

We prove this equivalence in two steps by using the concept of *path products* over a CG: (A) Show that `stack_cg_grad cg j i` is equal to the sum of all path products from j to i and; (B) Show that this sum of path products can be factored into the sum of parent gradients multiplied by local partial derivatives (LPDs).

A *path* from node j to node i in a stack CG is a list of indices beginning with j and ending with i, where each element is a parent of the subsequent element. The *path product*, denoted $\Pi_{\text{CG}}(xs)$, is the product of the LPDs encountered along that path:

$$\Pi_{\text{CG}}([j, v_1, v_2, \ldots, v_n, i]) = \frac{\partial j}{\partial v_1} \cdot \frac{\partial v_1}{\partial v_2} \cdots \frac{\partial v_{n-1}}{\partial v_n} \cdot \frac{\partial v_n}{\partial i}$$

where we have an Isabelle function `LPD cg u v` $= \frac{\partial u}{\partial v}$ if u is a parent of v, and 0 otherwise.

This lemma formalises step (A), demonstrating that FAD is equivalent to summing the path products:

```
lemma stack_cg_grad_sum_of_paths_equivalence:
  assumes "j < length_cg cg"
  shows "stack_cg_grad cg j i = (∑ path ∈ paths_set cg j i . path_product cg path)"
```

where `paths_set cg j i` denotes the set of all valid paths from j to i and `path_product cg` denotes Π_{CG} in Isabelle. This lemma is proven via strong induction on j, the recursive argument of `stack_cg_grad`. Essentially, the function's recursion down from j's children to i leads to an exploration of all paths from j to i, where each step along a path corresponds to a multiplication by a LPD.

To prove step (B), we observe that every path from j to i must necessarily pass through a parent of i. Therefore, the set of all paths from j to i is exactly the union of paths from j to i passing through each parent p:

$$\text{paths}(j, i) = \bigcup_{p \in \text{parents}(i)} \{xs \mathbin{@} [i] \mid xs \in \text{paths}(j, p)\}$$

where $xs \mathbin{@} [i]$ represents a path from j to p with i appended to the end. By substituting this union into our path product sum, we can factor out the final step to the parent:

$$\sum_{p \in \text{parents}(i)} \sum_{xs \in \text{paths}(j,p)} \Pi_{\text{CG}}(xs \mathbin{@} [i]) = \sum_{p \in \text{parents}(i)} \left(\sum_{xs \in \text{paths}(j,p)} \Pi_{\text{CG}}(xs) \right) \frac{\partial p}{\partial i}$$

Applying our result from (A) (`stack_cg_grad_sum_of_paths_equivalence`) to the inner right-hand side sum completes the proof of (B).

Combining (A) and (B) yields our final result, formalised as:

```
lemma stack_cg_grad_is_sum_of_parents':
  assumes "j < length_cg cg" and "j ≠ i"
  shows "stack_cg_grad cg j i = (∑ p ∈ parents_set cg i . stack_cg_grad cg j p * (LPD cg p i))"
```

This lemma successfully bridges the gap. It proves that the specification `stack_cg_grad` calculates i's gradient based exactly on i's parent gradients, mirroring the logic of the imperative `heap_cg_grad`. We use this equivalence in Section 5 to prove that our implementation refines its specification.

4.3. Heap CG to Stack CG Correspondence

We first develop a notion of correspondence between stack CGs and heap CGs. Stack CGs use indices to indicate children, whereas heap CGs use references, therefore, we use a mapping from stack indices to heap references as a basis for such a correspondence:

```
type_synonym 'a node_map = "nat ⇒ 'a node ref"
```

Now we create a function `valid_graph_upto` to check if the heap contains `node` references which match the arithmetic structure of a stack CG, thereby implying that the heap CG is a DAG.

```
1 fun valid_graph_upto :: "('a stack_cg ⇒ nat ⇒ 'a node  ⇒ bool) ⇒
2    ('a :: {field,linorder,heap}) node_map ⇒ 'a stack_cg ⇒ nat ⇒ assn"
3 where "valid_graph_upto P M cg 0 = emp"
4     | "valid_graph_upto P M cg (Suc n) =
5       ( ∃A  d g v. (M n ↦r Node d g (expected_oper M cg n) v) *
6         ↑(P cg n (Node d g (expected_oper M cg n) v))) *
7        valid_graph_upto P M cg n"
```

This function relies on separation logic [14, 30] syntax which is introduced to Isabelle/HOL via the refinement framework [13], and so we explain its meaning here. Firstly, `valid_graph_upto P M cg n` is a function that returns an *assertion* that the heap contains `node` references which match the sequence of operations of the first `n` elements of the stack CG, under a node mapping `M`, and satisfy a predicate `P`.

An assertion is just a predicate on *partial heaps*. A partial heap (h, A) is a heap h (a set of address-value pairs) whose addresses are limited to the set A. We say that a partial heap (h, A) models an assertion P, written $(h, A) \models P$, when (h, A) satisfies P using only the values stored in h at addresses in A. The separating conjunction, $*$, allows us to say that two assertions apply to distinct sections of the heap i.e., $(h, A) \models P * Q$ if and only if there exists **disjoint** address sets A_1 and A_2 such that $A = A_1 \cup A_2$ with $(h, A_1) \models P$ and $(h, A_2) \models Q$. Finally, `emp` is the assertion only satisfied by the empty heap.

The non-zero case of `valid_graph_upto` asserts that:

- Line 5: There exist some d, g and v such that the reference $M(n)$ points to a node on the heap with data d, gradient g, visited flag v, and whose operation and children match that of the stack CG. The term $\exists_A$ represents existential quantification over assertions.
- Line 6: The supplied predicate P is satisfied given the stack CG, the current index n, and $M(n)$'s underlying node. P is a predicate, and so the $\uparrow$ operator casts that to a *pure* assertion.
- Line 7: Recursively checks that the same properties hold for the lower indices

These three assertions are combined via separating conjunctions. Omitting the pure assertions for clarity, `valid_graph_upto P M cg (Suc n)` returns the following pointer assertions:

$$(M(n) \mapsto_r w_n) * (M(n-1) \mapsto_r w_{n-1}) * \cdots * (M(0) \mapsto_r w_0) * \text{emp}$$

which means each reference $M(i)$ on the heap points to a node object w_i, and due to the separating conjunctions, these pointers are distinct — $M(i) \neq M(j)$ for $i \neq j$. Note that the `emp` base case ensures that the partial heap of concern *only* contains the references corresponding to `cg` and nothing else.

We then define a wrapper function for the entire graph, `valid_graph P M cg`, by calling `valid_graph_upto P M cg (length_cg cg)`. We abbreviate commonly used predicates: `data_match cg` asserts that the heap CG's mutable data fields equal those returned by `stack_cg_data`, whilst `grad_match cg` asserts that the gradient and data fields match `stack_cg_grad cg (length_cg cg - 1)` and `stack_cg_data cg`, respectively. These predicates also ensure `visited v = False` so that the heap CG is ready for topological sorting (definitions in Appendix B.1).

We then show that the heap CG construction functions shown in Subsection 3.2 preserve correctness of the heap CG with respect to the stack CG. We show the lemma for the division operation:

```
1 lemma div_nodes_correct_short:
2   assumes "i < length_cg cg" "j < length_cg cg"
3   shows "< valid_graph data_match M cg>
4            div_nodes (M i) (M j)
5          < λr. valid_graph data_match (M(length_cg cg := r)) (StDiv i j ▷ cg) >"
```

The lemma statement is a *Hoare triple* [31] of the form $\langle P \rangle c \langle \lambda x \,.\, Q(x) \rangle$. It states that if we begin with a heap CG matching the structure and data of a stack CG (precondition, line 3), executing `div_nodes` on references `M(i)` and `M(j)` (program, line 4) will return a new reference `r`. Crucially, the resulting heap CG will correctly match the updated stack CG with the division operation appended, `StDiv i j ▷ cg`, with the node map `M` updated to point to `r` (postcondition, line 5).

With analogous lemmas proven for all operations, including constant nodes, we know that we can build a heap CG matching its functional specification before and after each primitive operation, such that the mutable data fields will agree with the `stack_cg_data` function.

5. Verification of Imperative RAD Algorithm

We give our main theorem: `heap_cg_grad` calculates gradient values matching those specified by `stack_cg_grad` (which we proved correct via `stack_cg_grad_derivative`):

```
theorem heap_cg_grad_eq_stack_cg_grad:
  assumes "0 < length_cg cg" and "fully_reachable_from_root cg"
  shows "< valid_graph data_match M cg >
           heap_cg_grad (M (length_cg cg - 1))
         < λ _ . valid_graph grad_match M cg > "
```

This states that if we have a heap CG with data correctly matching a stack CG, and run `heap_cg_grad` on its root node (located at `M(length_cg cg - 1)` in the heap), we will have a heap CG afterwards that has data *and* gradient fields matching `stack_cg_data cg` and `stack_cg_grad cg (length_cg cg - 1)`, respectively. The assumption that all nodes are reachable from the root ensures the topological sort captures the entire graph. As shown in Subsection 3.4, `heap_cg_grad` consists of four steps: topologically sort the nodes, zero the gradients, set the root gradient to 1, and iterate `backprop_step`. We must show each step works as intended.

5.1. Graph Initialisation and Topological Sorting

Before the gradients can be accumulated, `heap_cg_grad` must reset the graph's gradients and produce a topological ordering of the nodes. While fundamental to RAD's execution, the formal proofs for these steps are relatively straightforward and largely rely on standard separation logic lemmas and induction arguments over lists.

For the gradient reset, we define a `zeroing_invariant` and prove that iterating `set_grad 0` successfully zeroes all gradients while preserving topological order and data. For the topological sort, we formalise a pure functional equivalent over stack CGs and prove that our imperative `topo_sort_refs` function simulates it. This means the imperative algorithm returns a list of references, `rs`, containing every node exactly once, with parents strictly preceding children — a property encapsulated in the predicate `topo_and_full M cg rs`. The formal lemmas and proof details for these stages are provided in Appendix C.1 and Appendix C.2.

5.2. Iteration of `backprop_step`

The crux of the RAD verification lies in showing that iterating `backprop_step` over the topological order correctly models the stateful, in-place accumulation of gradients. The overarching correctness of this backpropagation loop is encapsulated here:

```
lemma iterate_backprop_step_correct:
assumes "length_cg cg > 0" and "topo_and_full M cg rs"
shows "< valid_graph data_zero_grad_match M cg >
         do { set_grad 1 (M (length_cg cg - 1));
              map_unit_monad backprop_step rs }
       < λ _ . valid_graph grad_match M cg > "
```

To prove this, we must track the intermediate value of a node's gradient as the iteration progresses. We construct a predicate `backprop_invariant` to capture the heap CG state during RAD, based on a

set processed of nodes that have had backprop_step already applied to them. It asserts that every non-root node's gradient equals the sum of processed parent gradients multiplied by LPDs, with their data unchanged and correct.

The primary verification challenge is proving that applying backprop_step to a single node maintains this invariant across the entire graph — in the proof we show that most nodes remain untouched, except for the children nodes whose gradients get incremented by exactly the correct value to maintain backprop_invariant. The specific gradient update rules depend on the node's primitive operation and whether its children are equal or distinct, therefore we require proofs for every scenario.

By proving these single-step updates for all operations (detailed in Appendix C.3), we establish a loop invariant. When the iteration completes, the processed set contains all valid indices. At the end, every node's gradient will equal the sum of its parent gradients multiplied by LPDs, which is the true derivative by the stack_cg_grad_is_sum_of_parents and stack_cg_grad_derivative lemmas. Thus, we conclude that iterating backprop_step over the topological order of the heap CG correctly computes the gradient for every node, completing the proof of our theorem heap_cg_grad_eq_stack_cg_grad.

5.3. Final Correctness Result for heap_cg_grad

The final consideration is that heap_cg_grad produces the same values as stack_cg_grad when the stack CG is rational. However, our correctness result for stack_cg_grad is specifically for real derivatives. Therefore, we formalised the below theorem, heap_cg_grad_cast_to_real_deriv which embeds the rationals into reals. It demonstrates that executing heap_cg_grad yields a heap CG whose root node data and evaluated gradients exactly match the function represented by the graph and all its partial derivatives.

```
theorem heap_cg_grad_cast_to_real_deriv:
  fixes cg :: "rat stack_cg" and x :: rat and i :: nat
  defines "L ≡ length_cg cg"
  assumes "0 < L" and "i < L" and "safe_deriv_update i x cg"
      and "fully_reachable_from_root (update_cg i x cg)"
  defines "f ≡ (λ(z::real). stack_cg_data (update_cg i z (map_cg of_rat cg)) (L - 1))"
  shows "< valid_graph data_match M (update_cg i x cg) >
             do {heap_cg_grad (M (L - 1));
             v_i ← !(M i);
             v_r ← !(M (L - 1));
             return (v_i, v_r)}
         <λ (v_i, v_r) .
           valid_graph grad_match M (update_cg i x cg) *
           ↑( of_rat (data v_r) = f (of_rat x)) *
           ↑( ( f has_real_derivative of_rat (grad v_i)) (at (of_rat x))) >"
```

6. Neural Networks and Code Generation

By performing RAD over a neural network's computational graph, we obtain the gradients of the network's *parameters* and can optimise them via gradient descent [1, 2]. In this section, we present a minimalist NN library powered by IsaGrad and outline two case studies: generating a non-linear decision boundary and next-character prediction via an RNN. The main building block is a *neuron*:

```
datatype 'a neuron = Neuron (weights: "'a node ref list") (bias: "'a node ref") (is_relu: bool)
```

A neuron contains weights $\vec{w} \in \mathbb{Q}^m$ and a bias $b \in \mathbb{Q}$, modelled as Const nodes. When evaluated, it computes $\vec{w} \cdot \vec{x} + b$, optionally passed through a $\mathrm{ReLU}(x) \equiv \max(x, 0)$ [37] activation. This is formalised using our CG building functions:

```
partial_function (heap) call_neuron ::
  "('a :: {heap,zero,plus,times,linorder}) neuron ⇒ 'a node ref list ⇒ 'a node ref Heap"
  where [code]: "call_neuron neuron rxs = do {
    weighted_sum ← map_sum_monad (λ (x, y) . mult_nodes x y) (zip (weights neuron) rxs);
    act ← add_nodes (bias neuron) weighted_sum;
```

```
    if is_relu neuron then relu_node act else return act}"
```

Here, `map_sum_monad` maps `mult_nodes` over the zipped list of weights and inputs, and then sums the output by folding the `add_nodes` function, implementing the dot product. The bias is then added on, and the output is potentially passed through ReLU depending on the `is_relu` flag.

Neurons are then collected into lists called *layers*, and a basic type of NN, a *multi-layer perceptron* (MLP) is simply a sequence of layers. The function `call_layer` evaluates every neuron in the layer on their respective input, and `call_mlp` iteratively applies `call_layer` to the previous layer's output, starting with the initial input data.

The final output of an MLP is evaluated by a *loss function*, which becomes the root of the CG. Executing RAD from this root calculates the derivative of the loss with respect to every parameter in the network, allowing the parameters to be iteratively updated via gradient descent ($p := p - \gamma \cdot \text{grad}(p)$) for a learning rate γ:

```
partial_function (heap) update_parameter :: "('a :: {heap,times,minus,plus}) ⇒ 'a node ref ⇒ unit Heap"
  where [code]: "update_parameter γ rp = do {
    v ← !rp;
    set_data (data v - γ * grad v) rp}"
```

6.1. Non-Linear Decision Boundary Demo

Isabelle's code generation allows us to translate our verified CG functions, RAD algorithm and NN library into executable Haskell code. As part of this translation we map rationals to Haskell's `Double` for improved evaluation speed. While this is standard practice, this mapping is not fully faithful because machine floating-point representations have limited precision. A closer mapping would be to Haskell's `Rational`, which would reduce evaluation speed and still not be faithful because of machines' finite range. To eliminate all doubt, one would need to verify the algorithm with a formalisation of the machine representation rather than idealised rationals. Since the CG and RAD algorithm are verified in Isabelle, this reduces the trusted computing base to Isabelle's code generator, the rational to `Double` mapping, Haskell's standard library and runtime, and the underlying hardware.

To demonstrate the functionality of this generated code, we replicate the same demo as Karpathy's Micrograd [38]. This involves training a neural network to compute a non-linear decision boundary for the `moons` dataset from scikit-learn [39] by using the Hinge Loss function [40]. Figure 2 shows the decision boundary calculated by a NN with two 16-node hidden layers. On a HP Elitebook i5 with 16 GB of RAM, the original Micrograd version takes over 100 seconds whereas ours takes only 7 seconds to reproduce the same result.

Micrograd also offers a basic neural network library which is powered by a RAD algorithm that operates over reference-based computational graphs in Python. In the demo, there are 50 red points and 50 blue points which a NN is being trained to separate with a non-linear decision boundary. Our Isabelle formalisation of the loss function (`loss_function_k`, detailed in Appendix D) mirrors Micrograd's Python implementation closely. It calculates the model's raw decision scores, computes the hinge loss for each prediction, and adds an L2 regularisation penalty [41] to prevent overfitting [42]. We present our core training function:

```
partial_function (heap) train::"rat ⇒ rat mlp ⇒ rat node ref list list ⇒ rat node ref list ⇒ rat Heap"
where [code, simp]:
"train γ nn inputs_list expected_list = do {
  loss ← loss_function nn inputs_list expected_list;
  heap_cg_grad loss;
  Heap_Monad.fold_map (update_parameter γ) (parameters_mlp nn);
  node_ref_to_data loss}"
```

The function simply calculates the loss of the NN based on the inputs and expected outputs, and makes it the root of a CG. It then calculates the gradients of the entire CG, and therefore all the NN parameters via `heap_cg_grad`. Afterwards, all parameters are updated via gradient descent and then finally the loss value is returned.

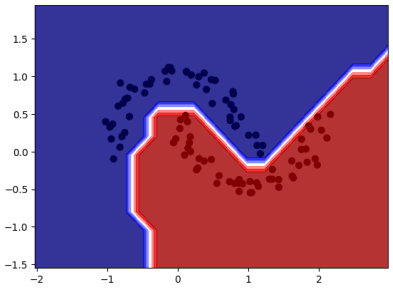

Figure 2: Non-Linear Decision Boundary of `moons` obtained by our NN Generated from Isabelle

This CG/NN library and demo provide strong evidence toward a proof-of-concept imperative ML library where key functionalities have been formally verified. Such a library would be useful for safety-critical AI, where we want a robust, correct, and auditable framework for building and training models.

6.2. Recurrent Neural Networks

To demonstrate a more architecturally complex and computationally expensive task, we implement a Recurrent Neural Network (RNN) demo, following Section 10.2 of Goodfellow et al. [1]. As they note early in the chapter, "To go from multilayer networks to recurrent networks, we need to take advantage of ... sharing parameters across different parts of a model". IsaGrad's reference-based CGs naturally handle this via reference sharing. When the recurrent calls of an RNN are unrolled, to produce a DAG, operations across all time steps have pointers to the same parameter nodes on the heap. As a result, our imperative RAD algorithm natively implements *Backpropagation Through Time* (BPTT): as `backprop_step` iterates over the course of `heap_cg_grad`, it inherently accumulates the gradients from multiple time steps into the shared parameter node's mutable gradient field, thus requiring no changes to our RAD algorithm.

We define a datatype for an RNN which makes use of the `layer` datatype of our basic NN library:

```
datatype 'a rnn = RNN (Wxh: "'a layer") (Whh: "'a layer") (Why: "'a layer") (h: "'a node ref list")
```

Our RNN has three layers: W_{xh}: input-to-hidden, W_{hh}: hidden-to-hidden and W_{hy}: hidden-to-output, alongside a hidden state vector h represented by a list of node references. Note that evaluating a layer on an input $\vec{x}$ via the `call_layer` function outputs a list of nodes whose data is, in effect, equivalent to a matrix multiplication of the layer's weights on $\vec{x}$, followed by a vector addition of the layer's biases.

We train our RNN to perform next-character prediction, learning the underlying statistical patterns of sequences from a text dataset. To achieve this, characters are mapped to $\mathbb{N}$. During training, we select a subsequence of length `seq_length` from our training dataset `D` and assign the `inputs` list as `D[s : s+seq_length]` and the targets list as `D[s+1 : s+seq_length+1]`, meaning that `targets[t]` is the character directly after `inputs[t]` in `D`. Beginning with a `loss` node of zero and an initial `hidden` state node list, for $t = 0, \ldots,$ `seq_length` $- 1$, we execute the following function to calculate the loss at each step in the sequence:

```
partial_function (heap) lossfun_step ::
  "rat rnn ⇒ nat list ⇒ nat list ⇒ nat ⇒ rat node ref ⇒ rat node ref list ⇒
    (rat node ref × rat node ref list) Heap" where [code, simp]:
  "lossfun_step model inputs targets t loss hidden = do {
    x ← onehot (inputs ! t) (get_input_size model);
    wx ← call_layer (Wxh model) x;
    wh ← call_layer (Whh model) hidden;
    h_new ← Heap_Monad.fold_map (λ (x, y) . add_nodes x y ⤜ elliot) (zip wx wh);
    ys ← call_layer (Why model) h_new;
    ps ← softmax_approx ys;
    neg_log_taylor_ref ← log_approx (ps ! (targets ! t)) ⤜ uminus_node;
    loss ← add_nodes loss neg_log_taylor_ref;
```

```
13    return (loss, h_new)}"
```

This function computes:

- Line 5: One-hot encode the input natural number as a vector $\vec{x}^{(t)}$.
- Lines 6–8: Update the hidden state via equation: $\vec{h}^{(t)} = \text{elliot}(U\vec{x}^{(t)} + V\vec{h}^{(t-1)} + \vec{b})$. Matrices U and V represent the weights of layers W_{xh} and W_{hh}, and our implementation handles the hidden bias $\vec{b}_h$ by summing the separate biases from layers W_{xh} and W_{hh}.
- Lines 9–10: Calculate model's probabilities for the next character via $\vec{p} = \text{softmax}(W\vec{h}^{(t)} + \vec{c})$
- Line 11: Compute the negative log-likelihood of the model giving the target response.
- Line 12: Increment the loss by the negative log-likelihood, $-\ln(\vec{p}_c)$, where $\vec{p}_c$ is the probability assigned to the target character.
- Line 13: Return the new loss and hidden state pair for the next step in the sequence.

The irrational functions ln and exp are approximated using piecewise-connected polynomials that closely match the curves over particular intervals (Appendix D.3). To compute the cross-entropy (CE) loss over the entire sequence we iterate `lossfun_step` from $t = 0, \ldots,$ `seq_length-1`, beginning with an initial loss of 0 and the model's current hidden state. Once the sequence is fully traversed, the total CE loss alongside the final hidden state are returned — this loop is wrapped in the `lossfun` function.

Finally, to train the RNN via BPTT, we execute the following:

```
partial_function (heap) train_rnn :: "rat rnn ⇒ rat ⇒ nat list ⇒ nat list ⇒ rat node ref Heap"
  where [code, simp]:
  "train_rnn model lr inputs targets = do {
    (total_loss, final_hidden) ← lossfun model inputs targets;
    heap_cg_grad total_loss;
    update_parameters_clipped model lr;
    update_hidden_state model final_hidden;
    return total_loss}"
```

This function gets the total loss and the final hidden state from the unrolled sequence of RNN calls. It calculates the gradients of all RNN parameters using `heap_cg_grad` and performs gradient descent, applying element-wise gradient clipping between $[-5, 5]$ to mitigate the exploding gradient problem common in RNNs [43]. Finally, it updates the model's hidden state references to the raw data values computed in the current training iteration. This resets the values as new leaf nodes, detaching the new hidden state from the previous computational graph (Truncated Backpropagation Through Time [44, 45]) which prevents unbounded CG growth over several training iterations.

To evaluate our implementation, we trained the RNN on the `dinos.txt` dataset [46], containing roughly 1,500 dinosaur names. Using an 80/20 training/test split, we trained an RNN with a hidden state size of 100 for 30,000 iterations. We periodically measured the Bits Per Character (BPC = CE / ln(2)) on the test set to monitor generalisation.

The results are shown in Figure 3 in Appendix D.2. The BPC metric exhibits a standard convergence curve, with rapid learning occurring during the first 5,000 iterations before settling into a stable plateau. The test BPC tracks the smoothed training BPC closely throughout the run, diverging slightly at the end to achieve a final test CE of 0.909 and a BPC of 1.311. Qualitatively, the model generated novel, plausible-sounding dinosaur names such as 'elkusaurus', 'rexulucoro', and 'douonyedus'.

This small-scale demo gives us evidence that our RNN implementation is learning as intended, and shows us that our imperative RAD algorithm natively supports unrolled recurrent architectures.

6.3. Toward Neural Network Verification

The main goal of this project was to produce a formally verified imperative RAD algorithm over reference-based computational graphs. The final theorem `heap_cg_grad_cast_to_real_deriv` confirms that we have succeeded in this aim. From there, a subsequent goal is to prove functional correctness and verify various properties of our NN implementation.

Our NN verification attempts are currently a work in progress. We have proven initial lemmas demonstrating that the library works as intended, such as, `call_neuron_correct_relu`.

```
lemma call_neuron_correct_relu:
  (*js and ks are the indices of the neuron weights and neuron inputs, respectively*)
  assumes "length js = length ks"
  and "⋀j. j ∈ set js ⟹ j < length_cg cg" and "⋀k. k ∈ set ks ⟹ k < length_cg cg"
  and "bias_idx < length_cg cg" and "weights neuron = map M js" and "bias neuron = M bias_idx"
  and "is_relu neuron = True"
  shows "< valid_graph data_match M cg >
           call_neuron neuron (map M ks)
         < λ act_ref . ∃_A M' cg' . valid_graph data_match M' cg'
             * ↑(act_ref = M' (length_cg cg' - 1))
             * ↑(stack_cg_data cg' (length_cg cg' - 1) =
                     ReLU (dot_product cg js ks + stack_cg_data cg bias_idx)) >"
```

This lemma proves that `call_neuron` outputs a valid heap CG whose root node represents $\text{ReLU}(ws \cdot xs + b)$. However, the proof of this lemma is rather laborious, requiring numerous sub-lemmas to manage the complexity associated with tracking individual scalar nodes on the heap.

This verification process raises a clear point of future work: reasoning about neural networks is fundamentally more tractable using matrix-based CGs. The vectorisation of operations collapses the size of the graph drastically. Switching to matrices would significantly compress the CG, thereby increasing computational efficiency and streamlining our verification approach, by allowing us to operate at a scale more suitable for neural networks.

7. Conclusion

This paper presents IsaGrad, the first formalisation of an imperative, reference-based reverse automatic differentiation algorithm within an interactive theorem prover. Using the Imperative HOL and separation logic libraries of Isabelle, we verified that this algorithm accurately computes derivatives over mutable computational graphs. We then utilised this algorithm to build a neural network library. By automatically generating executable Haskell code from our formalisation, we successfully demonstrated the framework's practical viability through a non-linear decision boundary classification task and a next-character prediction task using an RNN.

This project shows that it is possible to implement and verify an imperative, reference-based RAD algorithm which lies at the core of automatic differentiation engines in neural learning frameworks such as PyTorch. While the current implementation of IsaGrad uses scalar operations, this design choice successfully isolated and resolved the verification complexities associated with mutable data, references and separation logic. Consequently, this work lays much of the architectural and logical groundwork required to scale towards a fully verified, matrix-based machine learning library.

Work on adapting IsaGrad to use matrices/tensors is already underway, and will massively compress computational graphs and improve computational efficiency. Furthermore, higher dimensional data structures allow us to work at a layer of abstraction more appropriate for deep learning architectures, enabling us to implement more sophisticated models such as CNNs and transformers [47].

As AI systems are increasingly deployed into safety-critical domains, eliminating silent algorithmic errors within their underlying differentiation engines becomes of great importance. Having a robust, formally verified mathematical foundation minimises the risk of invisible gradient errors during training, and IsaGrad provides a rigorous blueprint for achieving end-to-end scalable formal guarantees in ML libraries.

Declaration on Generative AI

During the preparation of this paper, the first author used ChatGPT to perform grammar and spelling checks, as well as rewording suggestions. After using this tool, the authors reviewed and edited the

content as needed and take full responsibility for the publication's content. Generative AI was not used for writing the formalisation.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, The MIT Press, 2016. URL: https://www.deeplearningbook.org/.

[2] S. Ruder, An overview of gradient descent optimization algorithms, 2017. doi:10.48550/arXiv.1609.04747. arXiv:1609.04747.

[3] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, Journal of Machine Learning Research 18 (2018) 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An imperative style, high-performance deep learning library, in: Proceedings of the 33rd International Conference on Neural Information Processing Systems, 721, Curran Associates Inc., Red Hook, NY, USA, 2019, pp. 8026–8037.

[5] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ, accessed on 2026-02-17.

[6] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixão, F. Mutz, L. de Paula Veronese, T. Oliveira-Santos, A. F. De Souza, Self-driving cars: A survey, Expert Systems with Applications 165 (2021) 113816. URL: https://www.sciencedirect.com/science/article/pii/S095741742030628X. doi:https://doi.org/10.1016/j.eswa.2020.113816.

[7] F. Gou, J. Liu, C. Xiao, J. Wu, Research on artificial-intelligence-assisted medicine: A survey on medical artificial intelligence, Diagnostics 14 (2024). URL: https://www.mdpi.com/2075-4418/14/14/1472. doi:10.3390/diagnostics14141472.

[8] A. S. Panayides, A. Amini, N. D. Filipovic, A. Sharma, S. A. Tsaftaris, A. Young, D. Foran, N. Do, S. Golemati, T. Kurc, K. Huang, K. S. Nikita, B. P. Veasey, M. Zervakis, J. H. Saltz, C. S. Pattichis, Ai in medical imaging informatics: Current challenges and future directions, IEEE Journal of Biomedical and Health Informatics 24 (2020) 1837–1857. doi:10.1109/JBHI.2020.2991043.

[9] K. Narendra, K. Parthasarathy, Identification and control of dynamical systems using neural networks, IEEE Transactions on Neural Networks 1 (1990) 4–27. doi:10.1109/72.80202.

[10] H. Patino, D. Liu, Neural network-based model reference adaptive control system, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 30 (2000) 198–204. doi:10.1109/3477.826961.

[11] J. Chen, Y. Liang, Q. Shen, J. Jiang, S. Li, Toward Understanding Deep Learning Framework Bugs, ACM Transactions on Software Engineering and Methodology 32 (2023) 135:1–135:31. doi:10.1145/3587155.

[12] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, J. Matthews, Imperative Functional Programming with Isabelle/HOL, in: O. A. Mohamed, C. Muñoz, S. Tahar (Eds.), Theorem Proving in Higher Order Logics, volume 5170, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 134–149. doi:10.1007/978-3-540-71067-7_14.

[13] P. Lammich, Refinement to Imperative HOL, Journal of Automated Reasoning 62 (2019) 481–503. doi:10.1007/s10817-017-9437-1.

[14] J. Reynolds, Separation logic: A logic for shared mutable data structures, in: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Comput. Soc, Copenhagen, Denmark, 2002, pp. 55–74. doi:10.1109/LICS.2002.1029817.

[15] L. Yu, A formal model of ieee floating point arithmetic, Archive of Formal Proofs (2013). https://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.

[16] P. E. De Vilhena, F. Pottier, Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD

Library, Logical Methods in Computer Science Volume 19, Issue 4 (2023) 8851. doi:10.46298/lmcs-19(4:5)2023.
[17] R. J. George, J. Cruden, W. Adkisson, X. Zhong, H. Zhang, A. Anandkumar, TorchLean: Formalizing Neural Networks in Lean, 2026. URL: https://arxiv.org/abs/2602.22631. doi:10.48550/arXiv.2602.22631. arXiv:2602.22631, accessed on 2026-06-18.
[18] D. Selsam, P. Liang, D. L. Dill, Developing Bug-Free Machine Learning Systems With Formal Mathematics, 2017. doi:10.48550/arXiv.1706.08605. arXiv:1706.08605.
[19] A. Bagnall, G. Stewart, Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees, Proceedings of the AAAI Conference on Artificial Intelligence 33 (2019) 2662–2669. doi:10.1609/aaai.v33i01.33012662.
[20] A. D. Brucker, A. Stell, Verifying Feedforward Neural Networks for Classification in Isabelle/HOL, in: M. Chechik, J.-P. Katoen, M. Leucker (Eds.), Formal Methods, Springer International Publishing, Cham, 2023, pp. 427–444. doi:10.1007/978-3-031-27481-7_24.
[21] A. D. Brucker, A. Stell, Formalizing neural networks, Archive of Formal Proofs (2025). https://isa-afp.org/entries/Neural_Networks.html, Formal proof development.
[22] A. Bentkamp, J. C. Blanchette, D. Klakow, A Formal Proof of the Expressiveness of Deep Learning, Journal of Automated Reasoning 63 (2019) 347–368. doi:10.1007/s10817-018-9481-5.
[23] T. H. Cormen (Ed.), Introduction to Algorithms, 2nd. ed., 10th pr ed., MIT Press [u.a.], Cambridge, Mass., 2007.
[24] J. Stewart, Calculus: Early Transcendentals, 7th edition ed., Cengage Learning, 2016.
[25] C. Elliott, The simple essence of automatic differentiation, Proc. ACM Program. Lang. 2 (2018) 70:1–70:29. doi:10.1145/3236765.
[26] M. Huot, A. Shaikhha, Denotationally correct, purely functional, efficient reverse-mode automatic differentiation, 2023. URL: https://arxiv.org/abs/2212.09801. arXiv:2212.09801.
[27] M. Vákár, T. Smeding, CHAD: Combinatory Homomorphic Automatic Differentiation, ACM Transactions on Programming Languages and Systems (TOPLAS) 44 (2022) 20:1–20:49. doi:10.1145/3527634.
[28] B. A. Pearlmutter, J. M. Siskind, Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator, ACM Trans. Program. Lang. Syst. 30 (2008). URL: https://doi.org/10.1145/1330017.1330018. doi:10.1145/1330017.1330018.
[29] M. Lipovača, Learn you a Haskell for great good! : a beginner's guide, No Starch Press, 2011.
[30] P. O'Hearn, J. Reynolds, H. Yang, Local Reasoning about Programs that Alter Data Structures, in: L. Fribourg (Ed.), Computer Science Logic, Springer, Berlin, Heidelberg, 2001, pp. 1–19. doi:10.1007/3-540-44802-0_1.
[31] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, Foundations of Computing, 5. printing ed., MIT Press, Cambridge, Mass., 2001.
[32] Control.Monad.ST, 2026. URL: https://hackage-content.haskell.org/package/base-4.22.0.0/docs/Control-Monad-ST.html.
[33] A. Krauss, Recursive definitions of monadic functions, Electronic Proceedings in Theoretical Computer Science 43 (2010) 1–13. URL: http://dx.doi.org/10.4204/EPTCS.43.1. doi:10.4204/eptcs.43.1.
[34] R. T. Rockafellar, Convex Analysis, Princeton University Press, 2015.
[35] P. Forums, Gradient of ‘maximum‘ and ‘minimum‘ functions, 2022. URL: https://discuss.pytorch.org/t/gradient-of-maximum-and-minimum-functions/141529, accessed on 2026-02-11.
[36] PyTorch, Fix subgradient for element-wise max and min by varal7. pull request 59669. pytorch/pytorch, 2021. URL: https://github.com/pytorch/pytorch/pull/59669.
[37] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10, Omnipress, Madison, WI, USA, 2010, pp. 807–814.
[38] A. Karpathy, Github - karpathy/micrograd: A tiny scalar-valued autograd engine and a neural net library on top of it with pytorch-like api, 2020. URL: https://github.com/karpathy/micrograd.git.
[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer,

R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, coRR abs/1201.0490 (2012). URL: http://arxiv.org/abs/1201.0490. arXiv:1201.0490.

[40] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, A. Verri, Are loss functions all the same?, Neural Comput. 16 (2004) 1063–1076. URL: https://doi.org/10.1162/089976604773135104. doi:10.1162/089976604773135104.

[41] J. Kukačka, V. Golkov, D. Cremers, Regularization for deep learning: A taxonomy, CoRR abs/1710.10686 (2017). URL: http://arxiv.org/abs/1710.10686. arXiv:1710.10686.

[42] B. Ghojogh, M. Crowley, The theory behind overfitting, cross validation, regularization, bagging, and boosting: Tutorial, 2023. URL: https://arxiv.org/abs/1905.12787. arXiv:1905.12787.

[43] R. Pascanu, T. Mikolov, Y. Bengio, On the difficulty of training recurrent neural networks, in: Proceedings of the 30th International Conference on Machine Learning, PMLR, 2013, pp. 1310–1318.

[44] Y. Chauvin, D. E. Rumelhart, Back propagation : theory, architectures, and applications, Psychology Press, 2009.

[45] I. Sutskever, Training recurrent neural networks, 2013. URL: https://utoronto.scholaris.ca/items/99833e3b-ab4e-4155-a07f-44cbf6093667.

[46] jungsoh, rnn-character-level-language-model/dinos.txt at main · jungsoh/rnn-character-level-language-model, 2025. URL: https://github.com/jungsoh/rnn-character-level-language-model/blob/main/dinos.txt.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[48] S. Mac Lane, Categories for the Working Mathematician, volume 5 of *Graduate Texts in Mathematics*, Springer New York, New York, NY, 1978. doi:10.1007/978-1-4757-4721-8.

A. Imperative Topological Sort Implementation

```
partial_function (heap) topo_sort_aux ::
  "('a :: heap) node ref ⇒ 'a node ref list ⇒ 'a node ref list Heap"
  where [code]: "topo_sort_aux r ts = do {
    v ← !r;
    if visited v then
      return ts
    else do {
      mark_ref True r;
      children_topo ←
        (case oper v of
          Const ⇒ return ts
        | Mult c1 c2 ⇒ do {
            t1 ← topo_sort_aux c1 ts;
            topo_sort_aux c2 t1 }
        | ... (* other cases Plus, Div, Max, Uminus*)
        );
      return (r # children_topo)
    }
  }"
```

B. Separation Logic and Graph Validity

B.1. Predicate Abbreviations

The abbreviations for `valid_graph_upto P M cg (Suc n)` are

```
abbreviation "data_match cg n v ≡ data v = stack_cg_data cg n ∧ visited v = False"

abbreviation "grad_match cg n v ≡
  data v = stack_cg_data cg n ∧ grad v = stack_cg_grad cg (length_cg cg - 1) n ∧ visited  v = False"
```

C. Verification Details

C.1. Zeroing Gradients

We define a predicate to capture how the heap CG changes as we zero its gradients:

```
definition zeroing_invariant ::
  "nat set ⇒ 'a::{inverse,plus,times,uminus,zero,linorder} stack_cg ⇒ nat ⇒ 'a node ⇒ bool" where
  "zeroing_invariant processed cg i v = (
     (data v = stack_cg_data cg i ∧ visited v = False) ∧ (i ∈ processed ⟶ grad v = 0) )"
```

This predicate relies on a set, `processed`, which contains the indices of all nodes whose gradients have been reset. It dictates that a node's data must match the stack CG, its `visited` flag must remain false, and, if the node's index is in the `processed` set, then its gradient must be zero. Using this predicate, we establish how the heap CG's state evolves after a single step of gradient reset:

```
lemma zero_gradient_step:
  assumes "k < length_cg cg"
  shows "< valid_graph (zeroing_invariant processed) M cg  >
          set_grad 0 (M k)
         <λ _. valid_graph (zeroing_invariant (insert k processed)) M cg  >"
```

This lemma confirms that after setting a node `M(k)`'s gradient to 0, the `zeroing_invariant` is held when the index `k` is added to the processed set via `insert k processed`. To prove a statement like this, we developed a general rule providing sufficient conditions for how an update affects the heap CG:

```
lemma valid_graph_upto_update_rule:
  assumes "i < n"
  and "n ≤ length_cg cg" (*valid index*)
  and "⋀ old_v. oper old_v = oper (f old_v)" (*operation doesnt change -- CG structure static*)
  and "⋀ old_v. P cg i old_v ⟹ Q cg i (f old_v)" (*Q holds after change of M i*)
  and "⋀j v. j ≠ i ∧ j < n ⟹ P cg j v ⟹ Q cg j v" (*Q holds for every other element*)
shows "< valid_graph_upto P M cg n >
          do { v ← !(M i); (M i) := f v }
       <λ _. valid_graph_upto Q M cg n >"
```

Suppose we wish to prove that: if a valid heap CG satisfies a predicate P, and a reference $M(i)$ is updated from v to $f(v)$, then Q holds over the updated CG. The lemma `valid_graph_upto_update_rule` says that we must show two things in order to prove such a statement: that the new predicate Q holds for the updated reference, and that P implies Q for all unchanged references. If these conditions are met, Q holds over the heap CG after the update. Applying `valid_graph_upto_update_rule` makes the proof of `zero_gradient_step` trivial using Isabelle's simplifier.

Finally, we prove that iterating `set_grad 0` over a list of references, `map M idxs`, maintains this invariant, provided that we add `set idxs` alongside our `processed` set:

```
lemma zero_gradients_loop:
  assumes "set idxs ⊆ {0..<length_cg cg}" (*The indices must be valid for the cg*)
  shows "< valid_graph (zeroing_invariant processed) M cg >
            map_unit_monad (set_grad 0) (map M idxs)
         <λ _. valid_graph (zeroing_invariant (processed ∪ set idxs)) M cg  >"
```

This lemma is proven using induction on the list `idxs`. The overarching lemma showing `set_grad 0`'s correctness is

```
lemma zeroing_gradients_correct:
  assumes "0 < length_cg cg"
  and "fully_reachable_from_root cg"
```

```
  and "set rs = {M i | i . i < length_cg cg}"
  shows "< valid_graph data_match M cg  >
           map_unit_monad (set_grad 0) rs
         <λ _ . valid_graph data_zero_grad_match M cg >"
```

To prove this, we apply the `zero_gradients_loop` lemma using the following steps:

1. Instantiate `processed` as the empty set `{}`. This makes our precondition `valid_graph (zeroing_invariant {})`, which is equivalent to the desired starting state `valid_graph data_match`.
2. After the loop finishes, the set of processed indices equals `set idxs`. Because `idxs` covers every node in the stack CG, this postcondition entails our target state, `valid_graph data_zero_grad_match`.
3. We apply precondition strengthening and postcondition weakening [31] to yield the result.

C.2. Topological Sort

We define a pure, functional equivalent of the toposort algorithm, `topo_pure :: "'a stack_cg ⇒ nat ⇒ nat list ⇒ (nat list × nat list)"`. An initial call `topo_pure cg (length_cg cg - 1) []` returns `(v_out, t_out)`, representing the list of visited indices and the final topologically sorted list of indices for the stack CG, respectively. Because it operates on the stack CG specification, we can prove in a relatively straightforward manner that it terminates and produces a topologically sorted list containing every node index exactly once.

We then formalise what it means for a list of heap references to be a complete, topologically sorted representation of the graph using two predicates:

```
definition is_topo_sorted_refs :: "'a node_map ⇒ 'a stack_cg ⇒ 'a node ref list ⇒ bool" where
"is_topo_sorted_refs M cg rs = (distinct rs ∧
  (∀ p j . p < length_cg cg ∧ j < length_cg cg ∧ M p ∈ set rs ∧ M j ∈ set rs ∧ is_parent cg p j ⟶
     index rs (M p) < index rs (M j)))"

definition topo_and_full :: "'a node_map ⇒ 'a stack_cg ⇒ 'a node ref list ⇒ bool" where
    "topo_and_full M cg rs = (is_topo_sorted_refs M cg rs ∧  set rs = {M i | i . i < length_cg cg})"
```

The predicate `is_topo_sorted_refs` ensures there are no duplicate references and that for any parent-child pair `(p,j)` in `rs`, the parent appears strictly before the child in `rs`. The predicate `topo_and_full` extends this to ensure the list contains exactly the set of all stack CG indices mapped by the node map M.

C.3. Iteration of `backprop_step`

The following definition returns the intermediate value of the ith node's gradient during the stateful execution of RAD, given a set of processed nodes:

```
definition intermediate_heap_grad :: "nat set ⇒ 'a::{field, linorder} stack_cg ⇒ nat ⇒ 'a" where
  "intermediate_heap_grad processed cg i =
     (if i = length_cg cg - 1  then 1
      else (∑ p_idx ∈ processed . stack_cg_grad cg (length_cg cg - 1) p_idx * LPD cg p_idx i))"
```

The set `processed` contains the indices of all nodes that have had `backprop_step` applied to them. The definition states that a non-root node's gradient is the sum of the gradients of these processed nodes multiplied by the local partial derivative (LPD). Because the LPD is formally defined as 0 when `p_idx` is not a parent of `i`, this summation effectively calculates the sum of the gradients from processed *parents* only. The definition also ensures that the root's gradient equals 1.

From this definition, we construct the `backprop_invariant` which captures the state of the heap CG during the iteration of `backprop_step`.

```
definition backprop_invariant ::
  "nat set ⇒ 'a::{field, linorder} stack_cg ⇒ nat ⇒ 'a node ⇒ bool" where
  "backprop_invariant processed cg i v =
     (data v = stack_cg_data cg i ∧ grad v = intermediate_heap_grad processed cg i ∧ visited v = False)"
```

The lemma for the multiplication case with distinct children serves as an example:

```
lemma backprop_step_mult_different_children:
  assumes "i < length_cg cg"
  and "parents_set cg i ⊆ processed" (*i's parents processed since done in topological order*)
  and "processed ⊆ {0..length_cg cg - 1}" (*processed indices are valid*)
  and "i ∉ processed" (* M(i) has not been processed, but is about to be *)
  and "nth_cg cg i = StMult x1 x2" (*case: cg[i] is a multiplication node*)
  and "x1 ≠ x2"  (*cg[i] has _distinct_ children*)
  shows "< valid_graph (backprop_invariant processed) M cg  >
           backprop_step (M i)
         < λ _ . valid_graph (backprop_invariant (insert i processed)) M cg >"
```

Note that the assumption `parents_set cg i ⊆ processed` holds in the wider algorithm because nodes are processed in topological order.

The proof of this lemma requires showing two things. First, for any node that is not a child of `M(i)`, its gradient remains unchanged, which is correctly reflected by `intermediate_heap_grad (insert i processed)`. Second, for the children of `M(i)`, `intermediate_heap_grad (insert i processed)` also confirms that their gradient fields have increased exactly by `grad (M i) * LPD i x`. We prove this by making use of our valid_graph_upto_update_rule.

The lemma covering the case where the children are identical is structurally similar, replacing the assumption `x1 ≠ x2` with `x1 = x2`. However, the argument is a tad more complex. Because both children point to the same node in memory, the first gradient update modifies the shared node. When the algorithm executes the second update, it operates on this *already* partially altered state, temporarily violating the `backprop_invariant`. To manage this in the proof, we define an intermediate gradient predicate to express the state of the heap mid-execution.

C.4. Casting to Reals

```
theorem heap_cg_grad_cast_to_real_deriv:
  fixes cg :: "rat stack_cg" and x :: rat and i :: nat
  defines "L ≡ length_cg cg"
  assumes "0 < L" and "i < L" and "safe_deriv_update i x cg"
      and "fully_reachable_from_root (update_cg i x cg)"
  defines "f ≡ (λ(z::real). stack_cg_data (update_cg i z (map_cg of_rat cg)) (L - 1))"
  shows "< valid_graph data_match M (update_cg i x cg) >
             do {heap_cg_grad (M (L - 1));
             v_i ← !(M i);
             v_r ← !(M (L - 1));
             return (v_i, v_r)}
         <λ (v_i, v_r) .
           valid_graph grad_match M (update_cg i x cg) *
           ↑( of_rat (data v_r) = f (of_rat x)) *
           ↑( ( f has_real_derivative of_rat (grad v_i)) (at (of_rat x))) >"
```

The above statement consists in essence of the following assumptions and conclusion:

Assumptions of theorem: Take a rational stack CG and alter its ith node to x. Suppose that this alteration does not cause divisions by zero, nor maxima of equal arguments to be evaluated (line 4, for differentiability). Further suppose that the stack CG is fully reachable from the root (line 5). Now define a function $f : \mathbb{R} \to \mathbb{R}$ such that $f(z)$ is the data of the root node of the stack CG, given that the ith node has been altered to z (line 6). Note the rational `cg` and value x need to be cast from $\mathbb{Q}$ to $\mathbb{R}$ for f's definition.

Hoare Triple: Precondition (Line 7): We begin with a heap CG that matches this altered rational stack CG in arithmetic structure and data. Program (Lines 8-11): We execute `heap_cg_grad` and obtain the ith node and the root node. Postcondition (Lines 12-15): Afterwards, we are guaranteed three things: (1) The heap CG matches the stack CG in structure, data, and *gradient*. (2) The data of the heap CG's root node, cast to $\mathbb{R}$, equals $f(x)$. (3) The function f's real derivative at x exactly equals the gradient value stored at the ith heap CG node, cast to $\mathbb{R}$.

D. Neural Network Auxiliary Functions

D.1. Non-Linear Decision Boundary Loss Function

```
partial_function (heap) loss_function_k ::
    "rat mlp ⇒ rat node ref list list ⇒ rat node ref list ⇒ rat node ref Heap"
where [code, simp]:
"loss_function_k nn inputs y = do {
  scores ← Heap_Monad.fold_map (call_mlp nn) inputs ⤜ return ∘ List.concat;
  one_val ← const_node 1;
  sum_data_loss ← map_sum_monad ((case_prod mult_nodes) >=> sub_nodes one_val >=> relu_node)
                                (zip y scores);
  len_scores ← const_node ((of_int ∘ int ∘ length) scores);
  data_loss ← div_nodes sum_data_loss len_scores;
  α ← const_node 0.0001;
  summed_sq_params ← map_sum_monad (λ x . mult_nodes x x) (parameters_mlp nn);
  reg_loss ← mult_nodes α summed_sq_params;
  total_loss ← add_nodes data_loss reg_loss;
  return total_loss
}"
```

With the data preprocessing handled externally, line 5 calculates the model's raw decision scores, where the sign indicates the predicted class and the magnitude indicates confidence. Lines 6–8 compute the hinge loss for each prediction using the formula $\text{ReLU}(1 - y \cdot s)$, for expected class y and prediction score s. If a point is correctly classified with a sufficient margin ($y \cdot s \geq 1$), then $\ell(s) = 0$. Uncertain or misclassified points incur a positive loss. Note that the results are summed using the `map_sum_monad` and `>=>` (Kleisli arrow [48]) operators to chain the monadic CG operations. Line 10 calculates the average loss per data point, and then lines 11-13 calculate the L2 regularisation penalty [41] preventing overfitting [42] by penalising excessively large weights, which can cause non-smooth decision boundaries. Finally, lines 14-15 combine the *data loss* and *regularisation loss* to obtain the total loss, which is the quantity minimised during training.

D.2. RNN Training Curve

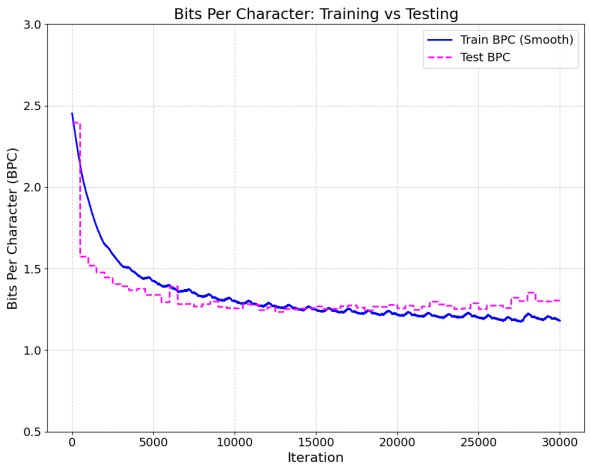

Figure 3: Bits Per Character during training and testing on `dinos.txt` [46] for RNN with hidden size 100

D.3. Logarithm and Softmax Rational Approximations

```
1  partial_function (heap) scaled_taylor_exp :: "rat node ref ⇒ rat node ref ⇒ rat node ref Heap" where [
       code, simp]:
```

```
  "scaled_taylor_exp m x = do {
    shifted ← sub_nodes x m;
    sixteen ← const_node (16);
    negsixteen ← uminus_node sixteen;
    clamped ← max_nodes shifted negsixteen;
    scaled ← div_nodes clamped sixteen;
    one ← const_node 1;
    two ← const_node 2;
    t ← add_nodes one scaled;
    sq ← mult_nodes scaled scaled;
    sq ← div_nodes sq two;
    t ← add_nodes t sq;

    t ← mult_nodes t t;
    t ← mult_nodes t t;
    t ← mult_nodes t t;
    mult_nodes t t
}"

partial_function (heap) softmax_approx :: "rat node ref list ⇒ rat node ref list Heap" where [code, simp]:
  "softmax_approx xs = do {
    m ← find_max xs;
    approx_exps ← Heap_Monad.fold_map (scaled_taylor_exp m) xs;
    s ← map_sum_monad return approx_exps;
    Heap_Monad.fold_map (λ x . div_nodes x s) approx_exps
  }"
```

Let $s(\vec{x})$ denote the softmax function for an input vector $\vec{x} \in \mathbb{Q}^n$. To ensure numerical stability, we exploit the shift-invariance property of softmax, $s(\vec{x}) = s(\vec{x} - c\vec{1})$, by setting $c = \max(\mathbf{x})$. We define the shifted input components as $x_i' = x_i - \max(\mathbf{x})$, meaning that $x_i' \leq 0$ for all i.

We approximate exp using the second-order Maclaurin polynomial $T_2(x) = 1 + x + \frac{x^2}{2}$. While this approximation is highly accurate near 0, it diverges for large negative values. To address this, we clamp the inputs at -16, defining $x_i'' = \max(x_i', -16)$. This lower bound introduces negligible error since $e^{x_i'} \approx 0$ for $x_i' \leq -16$, thus restricting our domain to $x_i'' \in [-16, 0]$.

To ensure the polynomial is evaluated where it is most accurate, we employ the scaling and squaring method. We scale down x_i'' by 16 to define $z_i = x_i''/16$, bounding the input to the relatively well-behaved interval $z_i \in [-1, 0]$. Over this region, $T_2(z_i) \approx e^{z_i}$.

Finally, we undo the scaling to approximate the original exponential. Using the fact that $e^x = (e^{x/16})^{16}$, we compute the result by squaring our Taylor approximation four consecutive times:

$$e^{x_i'} \approx e^{x_i''} \approx (T_2(z_i))^{16}$$

These rational approximations are then summed and used as the denominators to compute the final softmax distribution.

```
partial_function (heap) dropoff :: "rat node ref ⇒ rat node ref Heap" where [code, simp]:
  "dropoff x = do {
    one ← const_node 1;
    negtenth ← const_node (-1/10);
    d ← div_nodes negtenth x;
    add_nodes one d
}"

(* def log_taylor(x : Value) -> Value: return 2 * ((x - 1)/(x + 1)) *)
partial_function (heap) log_taylor :: "rat node ref ⇒ rat node ref Heap" where [code, simp]:
  "log_taylor x = do {
    one ← const_node 1;
    two ← const_node 2;
    numerator ← sub_nodes x one;
    denominator ← add_nodes x one;
    fraction ← div_nodes numerator denominator;
```

```
    mult_nodes two fraction
}"


partial_function (heap) log_approx :: "rat node ref ⇒ rat node ref Heap" where [code, simp]:
  "log_approx x = do {
    l ← log_taylor x;
    d ← dropoff x;
    m ←  min_nodes l d;
    neg_nine ← const_node (-9);
    max_nodes m neg_nine
}"
```

To approximate the natural logarithm $\ln(x)$ for probability inputs $x \in (0, 1]$, we employ a clamped, piecewise rational approximation that balances stability near $x = 1$ with asymptotic divergence as $x \to 0^+$.

First, we define a base approximation $l(x) = 2\frac{x-1}{x+1}$, which is the first term of the Taylor series expansion for the logarithm. However, because $\lim_{x\to 0^+} l(x) = -1$, we introduce an asymptotic "drop-off" function $d(x) = 1 - \frac{1}{10x}$ which rapidly approaches $-\infty$ as $x \to 0^+$.

We combine these two curves by taking their minimum: $\min(l(x), d(x))$. When x is near 1, $l(x) < d(x)$, therefore, $l(x)$ is returned. As the probability approaches 0, the curve for $d(x)$ steeply intersects and drops below $l(x)$, simulating an asymptote.

Finally, to prevent unbounded gradient explosions or numerical instability when the network predicts near-zero probabilities, we clamp the output from below at -9. The complete approximation is thus defined as:

$$\ln(x) \approx \max\left(\min\left(2\frac{x-1}{x+1}, 1 - \frac{1}{10x}\right), -9\right)$$

This yields a fully rational approximation of the natural logarithm.