Make sure to always keep your promises: A model-agnostic attribution algorithm for Neural Networks

FIRST AUTHOR NAME[1]

[1]FIRST AFFILIATION
AUTHOR1@EMAIL.ADDRESS AUTHOR2@EMAIL.ADDRESS

Abstract

TBC

1 Introduction

The rapid advancements in deep learning over the past decade have revolutionized numerous domains, from computer vision and natural language processing to healthcare and scientific discovery. However, the increasing complexity and opacity of high-performing models have raised significant concerns about their interpretability and trustworthiness. As these models are deployed in critical applications, understanding their decision-making processes has become a pressing need.

To address this, the field of interpretability has seen a surge of interest, leading to the development of various tools and techniques for explaining the behavior of deep neural networks. Modern interpretability methods can be broadly categorized into gradient-based approaches, perturbation-based methods, and propagation-based techniques: - **Gradient-based methods** (e.g., Integrated Gradients [Sundararajan et al.(2017)Sundararajan, Taly, and Yan]) compute attributions by analyzing the gradients of the model's output with respect to its input features. - **Perturbation-based methods** (e.g., SHAP [Lundberg & Lee(2017)Lundberg and Lee]) estimate feature importance by systematically perturbing input features and observing the impact on the model's predictions. - **Propagation-based methods** (e.g., Layer-wise Relevance Propagation (LRP) [Bach et al.(2015)Bach, Binder, Montavon, Klauschen, distribute the model's output relevance backward through the network using custom propagation rules.

While these methods have been instrumental in advancing our understanding of neural networks, they often fall short when applied to modern architectures. For instance, gradient-based methods can suffer from saturation effects, perturbation-based methods are computationally expensive, and propagation-based methods are often tailored to specific architectures, limiting their generalizability. Moreover, the scalability of these methods remains a challenge as model sizes continue to grow.

Amidst these challenges, there has been notable progress in adapting classic interpretability methods to modern architectures. For example, recent implementations of Integrated Gradients and Deep SHAP have been extended to handle transformer-based models and other complex architectures. In particular, Layer-wise Relevance Propagation has been modified to be attention-aware, enabling it to explain the behavior of transformer models effectively. Achitbat et al. (2024) demonstrated that LRP could faithfully explain transformer models like Vision Transformers (ViT) and LLaMA 2 by introducing custom attribution rules for the components of the attention mechanism.

Building on these findings, we propose a novel algorithm that addresses the limitations of existing methods. Our approach scales efficiently with model size and provides faithful attributions without compromising computational efficiency. By leveraging operation-level granularity and a promise-based system, our method achieves broad compatibility with modern architectures while maintaining high attribution quality.

2 Background

Modern transformer architectures, powered by the attention mechanism [**?**], have revolutionized machine learning by enabling models to handle multimodal data and scale to massive datasets. However, their complexity introduces challenges in interpretability, particularly in understanding latent reasoning processes. Attention maps, while useful for visualizing token interactions [**?**, **?**], fail to provide a holistic view of model behavior [**?**]. Recent studies

[?, ?] reveal that critical information, such as factual knowledge, is often stored in feed-forward network neurons, separate from attention layers.

Existing attribution methods can be broadly categorized into perturbation-based, gradient-based, and rule-based approaches. Perturbation-based methods, such as SHAP [Lundberg & Lee(2017)Lundberg and Lee], require extensive computational resources, making them impractical for large models. Gradient-based methods, like Input × Gradient [?], are efficient but suffer from noisy gradients and low faithfulness. Rule-based methods, including Layer-wise Relevance Propagation (LRP) [Bach et al.(2015)Bach, Binder, Montavon, Klauschen, Müller, and Samek], offer a promising alternative by allowing customization of propagation rules. However, applying LRP to transformers has been challenging due to unique operations like softmax and layer normalization [?, ?].

Background: Classic LRP via Taylor Approximation Classic LRP redistributes the prediction score (relevance) backwards through the layers of a neural network. The key assumption is that the relevance at output neuron j is proportional to the function value at j: $R_j \propto f_j(x)$. To propagate relevance to the inputs, we use the Taylor expansion and the conservation rule.

For a function $f_j(x)$, the first-order Taylor expansion around a root point x_0 is:

$$f_j(x) \approx f_j(x_0) + \sum_i \frac{\partial f_j}{\partial x_i}\Big|_{x_0} (x_i - x_{0,i}) \tag{1}$$

The conservation rule requires that the sum of input relevances equals the output relevance:

$$\sum_i R_{i\leftarrow j} = R_j \tag{2}$$

Assuming $R_{i\leftarrow j}$ is proportional to the Taylor term for input i:

$$R_{i\leftarrow j} = \frac{\frac{\partial f_j}{\partial x_i}(x_i - x_{0,i})}{f_j(x) - f_j(x_0) + \epsilon} R_j \tag{3}$$

For a linear layer $y_j = \sum_i x_i w_{ji} + b_j$, and choosing $x_0 = 0$, this simplifies to:

$$R_{i\leftarrow j} = \frac{x_i w_{ji}}{y_j + \epsilon} R_j \tag{4}$$

where ϵ is a small stabilizer to avoid division by zero. The total relevance assigned to input i is then $R_i = \sum_j R_{i\leftarrow j}$. This derivation shows how the epsilon rule arises from the proportionality assumption, Taylor expansion, and conservation principle, linking the mathematical definition of LRP to its practical implementation.

Our work builds on these insights to address the limitations of existing methods, providing a scalable and efficient solution for interpreting transformer models.

2.1 Computation Graphs in Deep Learning Frameworks

Deep learning frameworks rely on computation graphs to represent the sequence of operations performed during the forward and backward passes of a neural network. Formally, a computation graph is a directed acyclic graph (DAG) $G = (V, E)$, where V is the set of nodes representing operations (e.g., matrix multiplication, activation functions), and E is the set of directed edges representing data flow between operations.

Each node $v \in V$ in the graph corresponds to a function f_v that maps its inputs to outputs. For example, a linear layer can be represented as $f_v(x) = Wx + b$, where W and b are the weights and biases, respectively. The edges $e \in E$ capture the dependencies between these operations, ensuring that the graph is acyclic and can be traversed in topological order.

The backward pass in deep learning leverages the chain rule of calculus to compute gradients efficiently. For a scalar loss function L and a parameter θ, the gradient $\frac{\partial L}{\partial \theta}$ is computed by traversing the graph in reverse order:

$$\frac{\partial L}{\partial \theta} = \sum_{v \in \text{children}(\theta)} \frac{\partial L}{\partial f_v} \cdot \frac{\partial f_v}{\partial \theta}, \tag{5}$$

where children(θ) are the nodes that depend on θ.

Algorithm 1 Computation Graph Construction

Require: Model output $hidden\_states$
Ensure: In-adjacency list in_adj, Out-adjacency list out_adj, Topologically sorted nodes $topo\_stack$

Initialize $in\_adj \leftarrow \emptyset$, $out\_adj \leftarrow \emptyset$
Initialize $visited \leftarrow \emptyset$, $topo\_stack \leftarrow \emptyset$
$root \leftarrow hidden\_states.grad\_fn$
DFS($root$, in_adj, out_adj, $visited$, $topo\_stack$) **return** in_adj, out_adj, $topo\_stack$
function DFS(fcn, in_adj, out_adj, $visited$, $topo\_stack$)
    if fcn = None or $fcn \in visited$ **then**
        return
    end if
    $visited \leftarrow visited \cup \{fcn\}$
    for each $child \in fcn.next\_functions$ **do**
        $out\_adj[fcn].append(child)$
        $in\_adj[child].append(fcn)$
        DFS($child$, in_adj, out_adj, $visited$, $topo\_stack$)
    end for
    $topo\_stack.push(fcn)$
end function

The promise-based attribution algorithm extends this concept by associating each node in the graph with a custom propagation rule for relevance. This modular approach allows for operation-specific attributions, enabling the algorithm to handle complex architectures like transformers. By leveraging the DAG structure, the algorithm ensures that relevance is propagated efficiently and faithfully, adhering to the conservation principle:

$$\sum_{i \in \text{inputs}} R_i = \sum_{j \in \text{outputs}} R_j, \tag{6}$$

where R_i and R_j are the relevance scores at the input and output nodes, respectively.

This theoretical foundation underscores the flexibility and scalability of the promise-based approach, making it well-suited for modern deep learning models.

3 Methods

3.1 Computation Graph Processing

The computation graph is a directed acyclic graph (DAG) representing the sequence of operations in a neural network. However, the autograd graph only has one entrypoint at the Node representing the model output, and tracks only the backward edges from output to input. Algorithm 1 outlines the process of constructing an auxiliary graph from the autograd graph of a PyTorch model. This graph will allow access to any Node's in- and out-adjacencies, and will be used in the traversal algorithm.

3.2 Operation-Level Relevance Propagation

We implement propagation rules for each type of autograd Node (a Node in this case maps to an operation call), then traverse the auxiliary graph in a style similar to Kahn's Algorithm (link to Appendix), but using a Depth-First heuristic instead of Breadth-First. The operation-level approach modularizes relevance propagation, reducing any model architecture to a fixed set of fundamental tensor operations. Algorithm 2 describes the propagation process.

Algorithm 2 Operation-Level Relevance Propagation

Require: Model output *hidden_states*, Relevance interpretation target *target_node*, Auxiliary graph *in_adj*, *out_adj*, Output relevance R_{out}
Ensure: Input relevance R_{in}

Initialize $stack \leftarrow [hidden\_states.grad\_fn]$
Initialize $nodes\_pending \leftarrow \{node : len(parents)$ for $node, parents \in in\_adj.items()\}$
Initialize $node\_inputs \leftarrow \{node : [null$ for $parent$ in $parents\ ]$ for $node, parents \in in\_adj.items()\}$
Initialize fcn_map which maps $type(node)$ to its corresponding propagation function.
Set $node\_inputs[hidden\_states.grad\_fn] = [R\_out]$
while *stack* **do**
    $curnode \leftarrow stack.pop()$
    $curnode\_in\_rel = node\_inputs[curnode]$
    if $curnode == target\_node$ **then return** $R_{in} := curnode\_in\_rel$
    end if
    $prop\_fcn \leftarrow fcn\_map[type(curnode)]$
    $curnode\_outputs \leftarrow prop\_fcn(curnode\_in\_rel)$
    for each $child, output \in zip(out\_adj[curnode], curnode\_outputs)$ **do**
        $node\_inputs[child].add(output)$ [1]
        $nodes\_pending[child] \leftarrow nodes\_pending[child] - 1$
        if $nodes\_pending[child] == 0$ **then**
            $stack.push(child)$
        end if
    end for
end while

3.3 A Caveat to Autograd

While the above solution seems sound, a subtle issue lies within defining the propagation functions for each Node type.

Consider a simple approach for backpropagating R_c to R_a and R_b for $c = a + b$ as follows:

$$R_a = R_c \times \frac{a^2}{a^2 + b^2} \quad \text{and} \quad R_b = R_c \times \frac{b^2}{a^2 + b^2} \tag{7}$$

or alternatively,

$$R_a = R_c \times \frac{a.abs()}{a.abs() + b.abs()} \quad \text{and} \quad R_b = R_c \times \frac{b.abs()}{a.abs() + b.abs()} \tag{8}$$

We suggest this because we want to attribute relevance to each tensor component proportionally to the magnitude of its contribution to the operation result.

Since autograd is a differentiation library, it only caches within its Nodes information necessary for computing the gradients w.r.t. its arguments.

But for $c = a + b$, we see that $\frac{\partial a}{\partial c} = \frac{\partial b}{\partial c} = 1$. This implies that a and b need not be stored in any autograd 'AddBackward0' Node, which indeed is the case.

This poses a critical problem for us, since without a and b, we are then unable to compute R_a and R_b faithfully when we traverse the 'AddBackward0' Node.

3.4 The Promise System

The idea of the Promise System is to allow traversal to continue when a missing input case is encountered, but **we switch the objective of the traversal** from relevance propagation to a local DFS starting at the Node in question. While we search a way to recover these activations, we still store the information required to perform

[1] Extra implementation details hidden, assume we can access the number of total inputs landed and later aggregate the inputs according to the propagation function signature.

the propagations for each intermittent Node, making use of the traversal to the fullest. Once the activations are found, we can then compute the propagations for each Node that was covered in the search, starting at the search root. We will now formally define this system.

Definition 1 (Promise)**.** A Promise is a mutable metadata object that is attached to a Node which requires some uncached forward pass input to compute its relevance propagation. The core structure of a Promise object is defined below:

```
{
  "rout": R_out_curnode,
  "args": [ None ] * num_missing_inputs,
  "rins": [ None ] * num_missing_inputs
}
```

Definition 2 (Promise Origin Node)**.** A Promise's Origin Node refers to the Node for which the Promise was created.

Definition 3 (Promise Branch)**.** A Promise branch is a sub-object to a Promise, corresponding to exactly 1 of the Origin Node's missing forward call inputs. Sibling Promise Branches all share full access to their corresponding Promise object.

The objective of a Promise is to act as a placeholder while we continue traversing the graph in search of the Origin Node's missing forward call inputs.

Definition 4 (Promise Branch Arg Node)**.** A Promise Branch's Arg Node refers to the first Node along a Promise Branch in its forward pass output is retrievable from its own cached tensors.

We effectively delay the computation of subsequent relevance propagations until we can resolve the entire Branch from the Origin Node to the Arg Node.

This leads to an amended version of the previous propagation algorithm, Algorithm 3 (Note that this algorithm is simplified from the algorithm used in practice for the sake of understanding).

Algorithm 3 Operation-Level Relevance Propagation With Promises

Require: Model output $hidden\_states$, Relevance interpretation target $target\_node$, Auxiliary graph in_adj, out_adj, Output relevance R_{out}
Ensure: Input relevance R_{in}

Initialize as in Algorithm 2
while $stack$ **do**
  $curnode \leftarrow stack.pop()$
  $curnode\_in\_rel = node\_inputs[curnode]$
  if $curnode == target\_node$ **then return** $R_{in} := curnode\_in\_rel$
  else if $curnode$ requires Promise **then**
    $curnode.promise \leftarrow create\_new\_promise(type(curnode), R_{out})$
    $curnode\_outputs \leftarrow curnode.promise.branches$
  else
    $prop\_fcn \leftarrow fcn\_map[type(curnode)]$
    $curnode\_outputs \leftarrow prop\_fcn(curnode\_in\_rel)$
  end if
  for each $child, output \in zip(out\_adj[curnode], curnode\_outputs)$ **do**
    $node\_inputs[child].add(output)$
    $nodes\_pending[child] \leftarrow nodes\_pending[child] - 1$
    if $nodes\_pending[child] == 0$ **then**
      $stack.push(child)$
    end if
  end for
end while

Meanwhile, we make the following changes to each Node type's propagation function to accept Promise-type inputs:

Algorithm 4 Non-Arg Node Propagation Function Promise Handling

Require: Autograd Node *node*, Propagation input R_{out}
Ensure: Propagation output list R_{in}
if R_{out} is a Promise Branch **then**
  Define fwd as a closure of *node*'s forward pass.
  Define bwd as a closure of *node*'s relevance distribution logic.
  $R_{out}.record(fwd, bwd)$ **return** R_{out}
end if
// Propagate R_{out}... **return** R_{in}

Algorithm 5 Arg Node Propagation Function Promise Handling

Require: Autograd Node *node*, Propagation input R_{out}
Ensure: Propagation output list R_{in}
if R_{out} is a Promise Branch **then**
  Define $retrieve_fwd_output$ as a function that extracts *node*'s forward pass output given its saved tensors.
  $activation = retrieve_fwd_output(node)$
  $R_{out}.set_arg(activation)$
  $R_{out}.trigger_promise_completion()$
  if $R_{out}.promise_is_complete$ **then** $R_{out} = R_{out}.propagated_relevance$
  else
    // Signal for queueing this Node until the Promise is complete **return**
  end if
end if
// Propagate R_{out}... **return** R_{in}

3.5 Theoretical Analysis

Scalability in Memory and Computational Complexity We now prove that the Promise Attribution Framework scales efficiently in terms of both memory and computational complexity.

Theorem 1. *Let n be the number of operations in the computation graph G and m be the number of edges. The Promise Attribution Framework has a time complexity of $O(n + m)$ and a space complexity of $O(n + m)$.*

Proof. The forward pass constructs the computation graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. Constructing G requires $O(n + m)$ operations, as each operation and its dependencies are recorded once.

During the backward pass, the algorithm traverses G in reverse topological order. Each node $v \in V$ resolves its promise object P_v, which involves a constant amount of work per edge $e \in E$. Thus, the total time complexity is $O(n + m)$.

The space complexity is determined by the storage of G and the promise objects. Since G contains n nodes and m edges, and each promise object is associated with a node, the total space complexity is $O(n + m)$. □

This analysis demonstrates that the Promise Attribution Framework is both memory-efficient and computationally efficient, making it suitable for large-scale neural networks.

3.6 Experimental Setup

References

[Bach et al.(2015)Bach, Binder, Montavon, Klauschen, Müller, and Samek] Sebastian Bach, Alexander Binder, Grëgoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. Pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLoS ONE*, 10(7):e0130140, 2015. 1, 2

[Lundberg & Lee(2017)Lundberg and Lee] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, pp. 4765–4774, 2017. 1, 2

[Sundararajan et al.(2017)Sundararajan, Taly, and Yan] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *International Conference on Machine Learning*, pp. 3319–3328, 2017. 1

Appendices

.1 Implementation Details

Our implementation uses PyTorch hooks to record operations during the forward pass and constructs the computation graph dynamically. Each operation creates a promise object, which stores the propagation rule and tracks readiness and completeness.

Python Example: Promise-based LRP Engine Below is a simplified Python pseudocode that captures the core logic of the promise-based LRP engine:

```
class Promise:
  def __init__(self, op_type, inputs):
    self.op_type = op_type
    self.inputs = inputs
    self.ready = False
    self.complete = False
    self.relevance = None

  def resolve(self, output_relevance):
    # Custom propagation rule for each op_type
    self.relevance = propagate_relevance(self.op_type, self.inputs, output_relevance)
    self.ready = True
    return self.relevance

def lrp_engine(graph, output_relevance):
  # Traverse graph in reverse topological order
  for node in reversed(graph.topo_order()):
    promise = node.promise
    if all(child.promise.complete for child in node.outputs):
      input_relevance = promise.resolve(node.output_relevance)
      for inp in node.inputs:
        inp.promise.relevance = input_relevance[inp]
      promise.complete = True
  return [node.promise.relevance for node in graph.input_nodes()]
```

Explicit Steps in the Algorithm

1. **Forward Pass:** Register hooks to record each operation and build the computation graph.
2. **Promise Creation:** For each operation, instantiate a promise object with its propagation rule.
3. **Backward Pass:** Starting from the output, traverse the graph in reverse topological order. Each promise is resolved when all output relevance is received, and marked complete when it distributes relevance to its inputs.
4. **Relevance Aggregation:** Input nodes collect relevance from all incoming promises, yielding the final attribution scores.

Readiness and Completeness

- **Readiness:** A promise is ready when all output relevance (from downstream nodes) has been received.
- **Completeness:** A promise is complete when it has distributed its relevance to all inputs according to its propagation rule.

This mechanism ensures that relevance is propagated in accordance with the mathematical definition of LRP,

maintaining conservation and proportionality at each node.