Neural Path Machines (NPM)

A Unified Framework for Trajectory-Based Interpretability, Internal-State Debugging, and Causal What-If Interventions

Technical Report / Preprint, 2025

Alexey A. Nekludoff

AstraVerge Institute

Principal Researcher

Email: an@astraverge.org

ORCID: 0009-0002-7724-5762

December 5, 2025

Abstract

Neural networks achieve remarkable performance across domains, yet their internal computation remains largely opaque. During inference, activations evolve as a sequence of hidden states whose dynamics ultimately determine the model's output. Traditional interpretability techniques focus on input–output relationships or gradient-based attributions and provide limited insight into the internal computational process itself.

This report introduces the *Neural Path Machine* (NPM), a framework for making neural computation observable at the level of internal trajectories. NPM records activation paths, identifies unstable or influential transitions, and enables causal *what-if* interventions by modifying activations during execution. These capabilities transform a neural network from a black box into a transparent discrete dynamical system whose internal states can be inspected, manipulated, and systematically debugged.

By exposing the structure of computational paths, NPM provides a principled foundation for tracing model failures, analysing sensitivity and robustness, and performing targeted model corrections. The trajectory-based perspective also suggests new training possibilities that operate on internal transitions rather than solely on output errors; these extensions are developed in a separate companion report.

Overall, NPM offers a coherent and practical methodology for studying and controlling the internal behaviour of neural networks, bridging interpretability, diagnostics, and dynamical analysis within a unified framework.

Contents

1 Introduction

Recent progress in deep learning has produced highly capable models whose internal computation remains largely opaque. A standard neural network exposes only its input–output behaviour, while the intermediate activations that determine its decisions evolve as hidden internal states. This opacity limits our ability to understand failures, correct model behaviour, or incorporate domain knowledge into the computational process.

A growing body of work views neural networks not as static functions but as *discrete dynamical systems* whose layers perform successive state transitions. Under this perspective, the forward pass generates a *trajectory of internal activations*, and a model's output is the terminal state of this trajectory. Despite the naturalness of this view, existing interpretability and debugging methods typically treat internal activations only as diagnostic artefacts rather than as structurally meaningful components of computation.

In this report we introduce the **Neural Path Machine (NPM)**, a framework for tracing, analysing, and modifying the internal computational paths of neural networks. NPM exposes layer-wise states, identifies unstable or influential transitions, and enables causal *what-if* interventions by modifying activations during execution. These capabilities allow neural networks to be studied and manipulated as transparent dynamical systems rather than undifferentiated black boxes.

The trajectory-based perspective suggests additional implications beyond interpretability and debugging. In particular, the ability to access and control internal state transitions opens the door to new forms of training and model correction that operate directly on computational paths. A detailed formulation of learning in this dynamical setting—including its treatment as an inverse problem—is developed in a separate companion report.

Contributions. This report makes the following contributions:

- **A formal trajectory-based model of neural computation.** We develop a representation of neural networks as discrete dynamical systems whose internal activations form observable computational paths. This perspective provides a coherent mathematical structure for analysing the evolution of internal states during inference.

- **The Neural Path Machine (NPM), an architecture-agnostic framework for internal-state tracing.** NPM exposes the full sequence of layer-wise activations, enabling systematic inspection of intermediate representations and revealing transition instabilities that are invisible to input–output analysis.

- **A unified method for causal interventions on neural activations.** We introduce a principled intervention mechanism that allows controlled modification of internal states during execution, supporting counterfactual analysis, sensitivity testing, and trajectory-level debugging.
- **A computational pipeline for diagnosing and correcting unstable transitions.** By combining trajectory tracing and causal interventions, NPM enables the identification of failure-inducing transitions and provides tools for targeted correction of internal computation.

Together, these contributions establish a unified operational foundation for interpreting, analysing, and manipulating the internal behaviour of neural networks as transparent discrete dynamical systems.

2 Neural Networks as Discrete Computational Systems

A neural network is usually described as a function that maps an input vector to an output vector. While this functional description is correct, it hides an important fact: a neural network performs its computation in a sequence of discrete and observable steps. Each layer transforms its input into an activation vector, and this activation becomes the input to the next layer [2]. In this way, the forward pass can be viewed as a chain of state transitions.

This perspective allows us to treat a neural network as a **discrete computational system**. Such systems are common in computer science: finite-state machines, digital circuits, and discrete-time dynamical systems all operate by updating internal states in steps. Neural networks follow the same pattern. They receive an input, update internal states layer by layer, and finally produce an output.

2.1 Layer-wise computation as state transitions

Let a neural network consist of L layers. For an input x, we define:

$$a_0 = x, \quad a_k = f_k(a_{k-1}), \quad k = 1, \ldots, L,$$

where f_k is the transformation performed by layer k (linear map, convolution, normalization, activation function, etc.). The vector a_k is the **activation state** of layer k.

The sequence

$$a_0 \to a_1 \to a_2 \to \cdots \to a_L$$

is the internal computation of the model. This process is deterministic for all standard architectures [4]. Given the same input x and the same weights, the sequence of activations is always the same.

This makes neural networks fundamentally different from stochastic black-box systems: their internal states are not hidden or random. They are real, measurable values that form a structured computational path.

2.2 Tick-based interpretation

To describe this process more clearly, we introduce the notion of a **tick**. A tick is a single update step in the forward computation.[1]

- Tick 0: the model receives the input a_0.
- Tick 1: the model computes $a_1 = f_1(a_0)$.
- Tick 2: the model computes $a_2 = f_2(a_1)$.
- ...
- Tick L: the model produces the output a_L.

This view aligns with the interpretation of neural networks as discrete-time dynamical systems [15], [20]. Each tick corresponds to one layer, and the entire forward pass is a sequence of L ticks.

This structure is crucial for the Neural Path Machine framework. Since each tick produces an observable state, we can record, inspect, and compare these states. This gives us a complete trace of the model's internal computation.

[1]The idea of discrete update steps as fundamental units of computation also appears in the Philosophy of Discrete Being (FDB), which treats "ticks" as basic ontological events. See A. A. Nekludoff, *Philosophy of Discrete Being. Part I*, Zenodo, 2025. DOI: 10.5281/zenodo.17572909.

2.3 Localities as structured internal components

Each layer can be understood as a **locality**[2]: a computational region with its own structure and internal dynamics.

For example:

- a fully connected layer has a vector-based locality,
- a convolutional layer has a spatial locality with channels,
- a transformer layer has attention-based locality patterns.

The activation a_k represents the internal state of locality k. The transition $a_k \to a_{k+1}$ is the update of this locality into the next one.

Different architectures still follow this pattern: convolutional networks [5], recurrent networks [3], and transformers [22] all compute forward activations step by step.

Thus:

$$\text{Neural Network} = \text{Ordered set of localities} = (L_0, L_1, \ldots, L_L)$$

$$\text{Computation} = \text{Transition sequence of states } (a_0, a_1, \ldots, a_L)$$

This provides the structure needed for defining computational trajectories in Section 3.

2.4 Observability and determinism

The key property of this system is **observability**[3]. All states a_k can be recorded exactly as they appear during computation, and this property is essential for trajectory-based analysis. This makes neural networks different from systems where internal states cannot be accessed (such as hidden Markov models or latent variable models).

[2]The idea of a locality as a coherent internal region with its own structural rules originates in the Philosophy of Discrete Being (FDB), where localities represent self-consistent units of computation or observation. See A. A. Nekludoff, *Philosophy of Discrete Being. Part I*, Zenodo, 2025. DOI: 10.5281/zenodo.17572909.

[3]The role of observability as a structural requirement for consistent multi-layer systems is developed in the Coherent Observational Epistemology (COE) framework, which formalizes how internal states can be accessed, compared, and aligned across computational processes. See A. A. Nekludoff, *Coherent Observational Epistemology (COE)*, Zenodo, 2025. DOI: 10.5281/zenodo.17632756.

The second important property is **determinism**. Unless the architecture explicitly includes stochastic components (such as dropout during training), the forward pass always produces the same sequence of states for the same input.

These two properties form the foundation for:

- computational trajectories,
- internal-state debugging,
- causal what-if interventions.

If states were not observable or deterministic, these methods would not be possible.

2.5 From discrete systems to trajectories

The interpretation of a neural network as a discrete computational system naturally leads to the idea of a **computational trajectory**. A trajectory is the ordered list of activation states produced during the forward computation.

$$\tau(x) = (a_0, a_1, \ldots, a_L)$$

This trajectory is the internal "path" that connects the input to the output. It is the central object of analysis in the Neural Path Machine.

In the next section, we formalize this concept and explain how trajectories can be compared, visualized, and used to understand a model's decision process.

3 Computational Trajectories

In the previous section, we described neural networks as discrete computational systems that update their internal state layer by layer. This creates a natural structure: every input produces an ordered sequence of activation states. In this section, we formalize this structure and introduce the concept of a **computational trajectory**. A trajectory is the central object in the Neural Path Machine, and it allows us to analyze how decisions are formed inside a model.

3.1 Definition of a computational trajectory

Let a neural network with layers $0, 1, \ldots, L$ process an input x. As shown earlier, each layer produces an activation state a_k. We define the **computational trajectory** of x as:

$$\tau(x) = (a_0, a_1, \ldots, a_L).$$

The trajectory contains all observable states of the forward pass and therefore captures the entire internal computation. Since modern feedforward and transformer-based networks are deterministic [2], $\tau(x)$ is uniquely determined by the input x and the model parameters.

A trajectory can be viewed as a "path" in a high-dimensional activation space. Each a_k lies in its own space $\mathbb{R}^{d_k}$, and the trajectory moves through these spaces in a fixed order corresponding to the network architecture.

3.2 Trajectory space

We define the **trajectory space** $\mathcal{T}$ as:

$$\mathcal{T} = \mathbb{R}^{d_0} \times \mathbb{R}^{d_1} \times \cdots \times \mathbb{R}^{d_L}.$$

Every element of $\mathcal{T}$ is a valid sequence of activations, regardless of whether it was produced by the model or not. The set of all trajectories that the network can produce forms a subset $\mathcal{T}_{\text{model}} \subseteq \mathcal{T}$.

This distinction becomes important in Section 5, where we introduce causal interventions: modifying internal states may move a trajectory outside $\mathcal{T}_{\text{model}}$, revealing how sensitive the network is to internal changes.

3.3 Trajectory divergence

To compare how two inputs are processed, we measure how their trajectories diverge. Let x and y be two inputs with trajectories $\tau(x)$ and $\tau(y)$. We define the **layer-wise divergence**:

$$\Delta_k(x, y) = \|a_k(x) - a_k(y)\|.$$

The choice of norm depends on the layer structure: L2 norm is common for dense

layers, while channel-wise norms may be more suitable for convolutional and transformer layers [15].

The **total divergence** is then:

$$D(x, y) = \sum_{k=0}^{L} \lambda_k \, \Delta_k(x, y),$$

where λ_k are optional layer weights (e.g., to emphasize deeper layers).

Divergence reveals at which layer two computations begin to separate. If two inputs belong to the same class but diverge strongly at some layer, this may indicate instability or oversensitivity in the network. If inputs from different classes diverge very early, this shows that the model separates them effectively.

3.4 Bifurcation points

The **bifurcation point** between inputs x and y is the earliest layer where their divergence exceeds a threshold ϵ:

$$k^* = \min\{k \mid \Delta_k(x, y) > \epsilon\}.$$

This layer is often critical for understanding the decision. In many cases, a misclassification can be traced to a single bifurcation point where the trajectory "turns" toward the wrong internal pattern. This aligns with observations from mechanistic interpretability [13], where specific layers or neurons cause large changes in the final output.

3.5 Trajectory clusters and decision regions

Since each input produces a trajectory, a dataset induces a set of trajectories:

$$\{\tau(x_i)\}_{i=1}^{N}.$$

Trajectories that correspond to the same class often form clusters or families. These clusters reflect internal decision regions of the model [15]. Instead of looking only at the output logits, trajectory-based analysis shows *how* the model internally groups samples.

This idea is especially useful for:

- detecting dataset drift,
- analyzing misclassifications,

- visualizing class separation,
- comparing model architectures.

3.6 Trajectories as a basis for debugging and interventions

Trajectory analysis supports the two major tools developed in later sections:

- **Internal-state debugging**: Divergence patterns show where a misclassification begins. A breakpoint can be placed exactly at the bifurcation point.
- **Causal what-if interventions**: By modifying an activation at layer k, we can observe how the remaining trajectory changes. This makes it possible to test which internal components truly cause a decision [14].

Trajectory-based reasoning therefore transforms interpretability from a set of external approximations into a direct analysis of the model's internal behaviour.

3.7 Summary

A computational trajectory captures the full sequence of internal states that a neural network produces when processing an input. This concept allows us to:

- compare internal computations across different inputs,
- identify where decisions emerge,
- detect unstable or unexpected internal patterns,
- support debugging and causal analysis.

In the next section, we introduce the tools needed to inspect and analyze these trajectories through internal-state debugging.

4 Internal-State Debugging

Computational trajectories provide a complete view of how a neural network processes an input. In this section, we introduce a practical method for analysing these trajectories: **internal-state debugging**. This method treats a neural network in the same way software engineers treat a program during execution: it allows us to pause the computation,

inspect internal values, compare two executions, and identify the exact point where a decision begins to change.

Debugging is possible because neural networks have two important properties: determinism and observability. Given an input, the network always produces the same sequence of internal states, and these states can be extracted and recorded during a forward pass. This makes it possible to study the internal behaviour of the model directly, instead of relying on indirect approximations such as gradients or saliency maps [18], [21].

4.1 Breakpoints

A **breakpoint** is a layer at which the forward computation is paused so that the activation state can be inspected. Formally, a breakpoint at layer k exposes the state a_k before any further computation is performed.

Breakpoints allow us to:

- record the activation state at a specific layer,
- modify the activation (for later causal interventions),
- compare two activation states from different inputs,
- analyse whether a misclassification begins at this layer.

In practice, breakpoints can be implemented by instrumenting the forward pass to save intermediate tensors. Modern deep learning frameworks already support hooks or callbacks for this purpose.

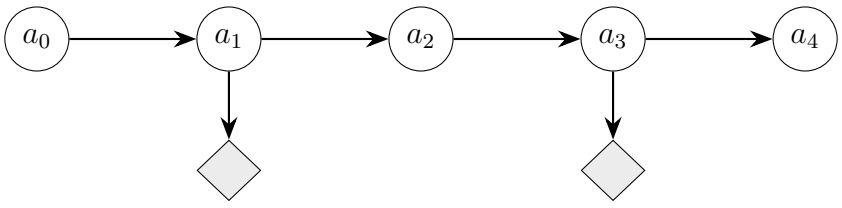

Figure 1: A trajectory with breakpoints for internal-state debugging.

4.2 Execution traces

An **execution trace** is the ordered list of activation states recorded when processing an input:

$$\text{Trace}(x) = (a_0, a_1, \ldots, a_L).$$

This is the same object as a computational trajectory, but in debugging we treat it operationally rather than mathematically. Traces can be saved, visualized, or compared across inputs.

Traces provide a clear view of internal behaviour that does not depend on gradient stability or surrogate models. This aligns with recent work in mechanistic interpretability, where full internal activations are used to understand model circuits and behaviours [10], [13].

4.3 Trace comparison

Given two inputs x and y, debugging often requires comparing their traces layer by layer:

$$\Delta_k(x, y) = \|a_k(x) - a_k(y)\|.$$

A growing divergence across layers indicates where the two computations begin to differ. Trace comparison is especially useful when:

- the model misclassifies x but not y,
- both inputs belong to the same class but follow different internal paths,
- adversarial examples lead to unexpected internal states [8].

Trace comparison makes it possible to identify the **bifurcation point** introduced in Section 3.

4.4 Locating failure points

When a network misclassifies an input, internal-state debugging can locate the exact layer where the problem begins. Let x be misclassified and y be a correctly classified reference input from the same true class. We compute:

$$k^* = \min\{k \mid \Delta_k(x, y) > \epsilon\}.$$

Layer k^* is the failure point. At this layer, the internal computation of the model turns away from the expected path.

Common causes include:

- unstable activations,
- oversensitive neurons,
- incorrect feature extraction,
- attention patterns that shift unexpectedly,
- adversarial vulnerabilities.

This layer becomes the main candidate for inspection, visualization, or intervention.

4.5 Debugging misclassifications

Internal-state debugging provides several tools for analysing errors:

- **Activation visualization:** heatmaps, channel activations, attention maps.
- **Difference maps:** direct comparison $a_k(x) - a_k(y)$ shows which neurons diverge.
- **Path deviation plots:** plots of $\Delta_k(x, y)$ reveal how far the computation has moved from the reference trajectory.
- **Nearest-neighbour inspection:** inspecting trajectories of other samples close to x.

These tools reveal why the model favoured one internal pattern instead of another.

4.6 Debugging architectures

Debugging is not limited to individual inputs. By analysing aggregated traces, we can evaluate the behaviour of the architecture itself:

- Which layers are most sensitive to input variation?
- Which layers produce stable internal clusters?
- Do deeper layers amplify noise or suppress it?
- Does the model rely on a small set of critical neurons?

Such insights can guide architecture refinement, regularization strategies, and robustness improvements.

4.7 Relationship to trajectories and interventions

Internal-state debugging connects the two main ideas of NPM:

- Debugging uses computational trajectories to inspect the internal behaviour of the model.
- Debugging prepares the ground for causal what-if interventions, because breakpoints allow us to modify internal states and observe the consequences.

Thus, debugging is the operational tool that enables deeper causal analysis in the next section.

4.8 Summary

Internal-state debugging provides a practical and precise way to inspect the internal computation of a neural network. By recording internal states, comparing traces, and identifying failure points, we gain the ability to understand *how* a network reaches a decision and *where* errors begin. This method forms the bridge between trajectories and causal interventions.

5 Causal What-If Interventions

Computational trajectories and internal-state debugging allow us to observe the internal behaviour of a neural network. In this section, we extend these ideas by introducing **causal what-if interventions**. An intervention modifies an internal activation during the forward pass and observes how this modification changes the remaining trajectory and the final output. This method provides direct evidence about which internal states *cause* specific decisions.

Causal interventions are inspired by structural causal models [14], where variables can be manipulated to test their influence on outcomes. Here, we apply the same principle to the internal states of a neural network. This approach goes beyond gradient-based attribution: instead of asking which input feature matters, we ask which *internal computation* matters.

5.1 Definition of an intervention

Let a neural network process an input x and produce trajectory

$$\tau(x) = (a_0, a_1, \ldots, a_L).$$

A **causal intervention at layer** k replaces the natural activation a_k with a modified value $\tilde{a}_k$:

$$a_k \leftarrow \tilde{a}_k.$$

The modified trajectory becomes:

$$\tau_{\text{int}}(x) = (a_0, \ldots, a_{k-1}, \tilde{a}_k, a'_{k+1}, \ldots, a'_L),$$

where $a'_{k+1}, \ldots, a'_L$ are the activations computed from the intervened state $\tilde{a}_k$.

This definition is intentionally simple: an intervention is a direct overwrite of an internal state, followed by normal forward computation. The difference between the original and intervened outputs shows how important the state a_k is for the model's behaviour.

5.2 Types of interventions

Different types of interventions provide different kinds of insight.

- **Neuron-level intervention:** Modify one neuron $a_k[i]$ and keep all others unchanged.
- **Vector-level intervention:** Replace the entire activation vector a_k with another vector.
- **Pattern injection:** Insert an activation pattern taken from another input y:

$$a_k \leftarrow a_k(y).$$

- **Noise injection:** Add perturbation δ to test robustness:

$$a_k \leftarrow a_k + \delta.$$

- **Projection intervention:** Project the activation onto a subspace (e.g., dominant principal components).

These operations allow researchers to understand the internal decision boundaries of a model more clearly than through input perturbations alone [8].

5.3 Measuring the influence of an intervention

To quantify the effect of an intervention, we compare the original output o with the new output $\tilde{o}$:

$$\Delta_{\text{out}} = \|o - \tilde{o}\|.$$

A large difference indicates that the intervened state a_k carries significant causal importance for the final decision.

We may also measure internal effects:

$$\Delta_j(k) = \|a_j - a'_j\| \quad \text{for } j > k.$$

These values reveal how the intervention propagates through the network.

If an intervention at layer k produces large internal or output changes, then k is a **causally sensitive layer**. If the effect is small or zero, the layer is **causally stable**.

5.4 Minimal interventions that change the decision

A powerful application of interventions is to find the smallest change to an internal state that alters the final class prediction:

$$\min_{\delta} \|\delta\| \quad \text{s.t.} \quad f(\tau_{\text{int}}(x;\delta)) \neq f(\tau(x)).$$

This optimization reveals the internal directions that strongly influence the model's classification. These directions can be used to:

- detect vulnerabilities,
- study decision boundaries,
- understand which internal patterns define a class.

This technique complements adversarial analysis, which typically focuses on input perturbations [8]. Here, we examine the internal cause of the decision instead of its external sensitivity.

5.5 Interventions for repairing misclassifications

Interventions can also be used to test whether a misclassification can be corrected at an internal layer. Suppose input x is misclassified, and y is a correctly classified reference input from the same true class. We can test:

$$a_k(x) \leftarrow a_k(y).$$

If this operation changes the final output to the correct label, we conclude:

- layer k contains a representation essential for the correct class,
- x diverged from the correct trajectory at or before layer k,
- the error likely originates near the bifurcation point found in Section 3.

This method directly connects debugging and causal reasoning.

5.6 Interventions as internal counterfactuals

An intervention answers a counterfactual question:

"What would the model output if this internal state had been different?"

This is the core of causal analysis. Counterfactual inspection allows us to trace model behaviour not only as it is, but as it *could be* under alternative internal conditions. The approach aligns with causal counterfactual reasoning in machine learning [14] but applies it at the level of internal activations rather than inputs.

5.7 Summary

Causal what-if interventions give us a direct way to test how internal states influence a neural network's final decision. By modifying activations at any layer, we can measure causal sensitivity, identify influential components, repair misclassifications, and explore counterfactual behaviours. Interventions transform neural networks from passive black

boxes into active computational systems that can be probed, manipulated, and understood from the inside.

This prepares the ground for the unified framework described in Section 6.

6 The Neural Path Machine (NPM): Unified Framework

The previous sections introduced three core ideas: computational trajectories, internal-state debugging, and causal what-if interventions. Each idea provides a different way to access and analyse the internal behaviour of a neural network. In this section, we combine these elements into a single structure called the **Neural Path Machine (NPM)**. The NPM is a general framework for understanding neural computation as an observable and testable process.

The goal of the NPM is to make neural networks more transparent and more reliable by providing a set of operations that expose, compare, modify, and analyse internal computational paths. This framework is independent of architecture and can be applied to dense models, convolutional networks, recurrent models, and transformers.

6.1 Components of the NPM

The NPM framework consists of four main components:

- **Trajectory Engine** – records the sequence of activation states produced by the model.
- **Debugger** – inspects internal states, compares traces, and identifies failure points.
- **Intervention Module** – modifies internal activations to test causal influence.
- **Repair and Analysis Layer** – uses interventions to correct errors or explore alternative internal paths.

Together, these components give complete access to the internal computation of a model.

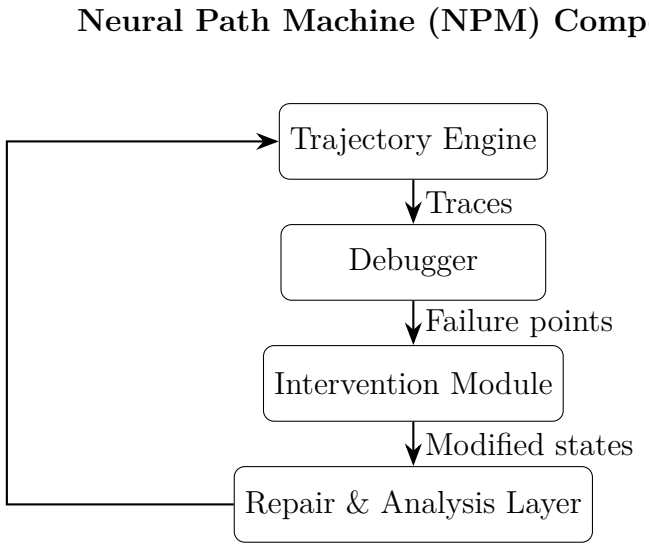

Figure 2: Internal structure and feedback loop in the NPM framework.

6.2 Core operations

The NPM defines several operations that can be applied to any neural network.

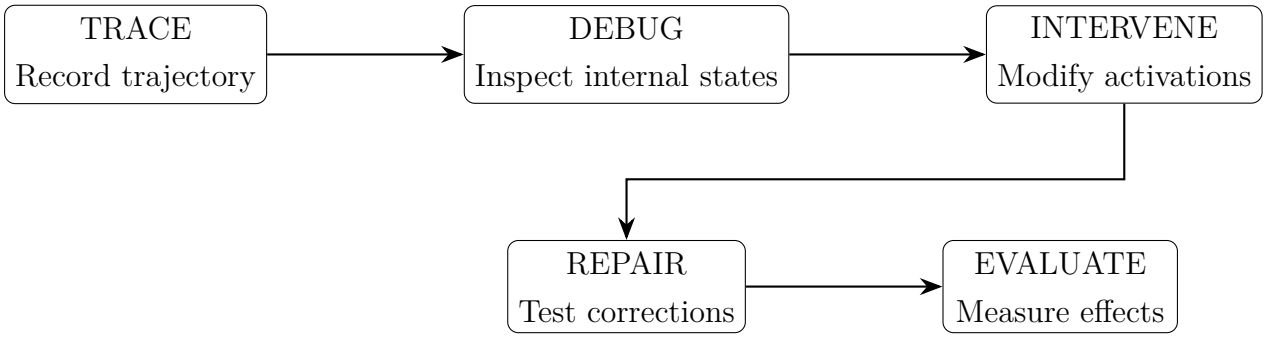

Figure 3: High-level pipeline of the Neural Path Machine (NPM).

1. TRACE Record the full computational trajectory of input x:

$$\text{TRACE}(x) = (a_0, a_1, \ldots, a_L).$$

This is the basic operation of the Trajectory Engine.

2. BREAK(k) Pause the computation at layer k and expose the activation a_k. This operation allows inspection, comparison, or modification.

3. COMPARE(x, y) Compare two trajectories layer by layer:

$$\Delta_k(x, y) = \|a_k(x) - a_k(y)\|.$$

This operation belongs to the Debugger.

4. INTERVENE$(k, \tilde{a}_k)$ Modify the activation at layer k and continue computation from the modified state:

$$a_k \leftarrow \tilde{a}_k.$$

This is the core of causal analysis.

5. REPAIR(x) Attempt to correct a misclassification by copying activations from a reference input y:

$$a_k(x) \leftarrow a_k(y).$$

If this changes the final decision, layer k is identified as a key error location.

6. SHAPE(x, U) Modify internal states to steer the computation toward a desired internal region U. This is useful for understanding decision boundaries or improving robustness.

7. EVALUATE Measure the effect of interventions on internal states or on the final output:

$$\Delta_{\text{out}} = \|o - \tilde{o}\|.$$

This provides a quantitative view of causal influence.

6.3 Architectural view

The NPM can be expressed in a simple structural diagram:

$$\begin{array}{ccccc} \text{Input} & \xrightarrow{\text{TRACE}} & \text{Trajectory} & \xrightarrow{\text{DEBUG}} & \text{Failure Points} \\ & & \downarrow \text{INTERVENE} & & \downarrow \text{REPAIR} \\ & & \text{Modified Trajectory} & \xrightarrow{\text{EVALUATE}} & \text{Output Change} \end{array}$$

This diagram shows how the NPM links the three main ideas into a single computational pipeline.

6.4 Why a unified framework matters

Without a unified framework, trajectory analysis, debugging, and interventions would remain separate tools. The NPM provides structure and consistency in the following ways:

- It gives a precise definition of what an internal computation is.
- It offers standardized operations for analysing and modifying neural behaviour.
- It provides clear interfaces for new algorithms and visualization tools.
- It supports reproducibility by treating neural computation as observable data.

As neural networks grow more complex, structured tools become necessary to keep their behaviour understandable.

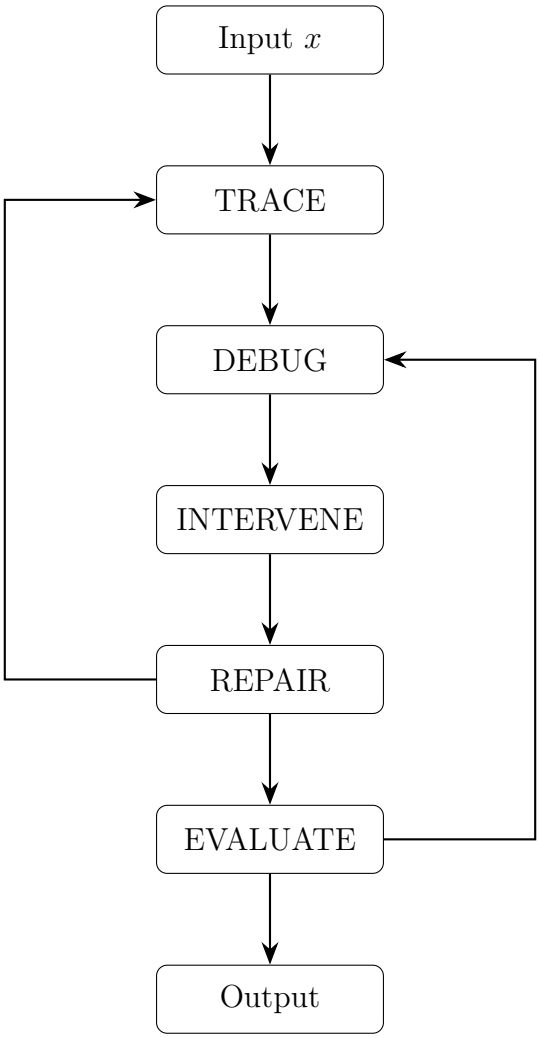

Figure 4: Symmetric unified computational graph of the Neural Path Machine (NPM).

6.5 Relationships between components

The components of the NPM interact in a simple and direct way:

- **Tracing** produces the data needed for debugging.
- **Debugging** identifies layers where interventions should be applied.
- **Interventions** test causal hypotheses generated by debugging.
- **Repair** uses interventions to evaluate if errors can be corrected internally.
- **Evaluation** measures the strength of each causal relationship.

These interactions give the NPM the structure of a full analytical pipeline rather than a set of independent tools.

6.6 The NPM as a model of computation

One way to understand the NPM is to view the neural network as a program with:

- observable states (activations),
- deterministic transitions (layer operations),
- controllable internal variables (for interventions),
- testable causal dependencies.

This makes NPM not only an analytical tool but also a foundation for future research into:

- interpretable architectures,
- robust training methods,
- neural program analysis,
- verification and alignment techniques.

6.7 Summary

The Neural Path Machine unifies trajectories, debugging, and interventions into a single framework. It treats neural networks as observable computational systems with well-defined operations for recording, analysing, comparing, and modifying internal activations. This unified structure opens the door to new theoretical and practical tools for interpretability, robustness, and architecture design.

The next section introduces concrete algorithms for implementing the core operations of the NPM.

7 Algorithms

This section presents the main algorithms that implement the core operations of the Neural Path Machine (NPM). The algorithms are written in simple pseudocode and can be applied to any neural network model with a standard forward function. All routines assume a deterministic computational graph and full access to internal layer activations.

7.1 TRACE: Recording a computational trajectory

The TRACE operation collects all intermediate activations produced during a forward pass.

Algorithm 1 TRACE(x)

Require: Input sample x

Ensure: Trajectory $\tau(x) = (a_0, a_1, \ldots, a_L)$

$a_0 \leftarrow x$
for $k = 1$ to L **do**
    $a_k \leftarrow f_k(a_{k-1})$ ▷ Layer computation
    Append a_k to trajectory
end for
return $\tau(x)$

TRACE produces the full internal path of computation and is used by all other NPM operations.

7.2 BREAK: Extracting an internal state

BREAK returns the activation at a chosen layer without modifying the forward pass.

Algorithm 2 BREAK(x, k)

Require: Input x, layer index k

Ensure: Activation a_k

$\tau \leftarrow$ TRACE(x)
return $\tau[k]$

7.3 COMPARE: Measuring trajectory divergence

COMPARE evaluates layer-wise differences between two trajectories.

Algorithm 3 COMPARE(x, y)

Require: Inputs x, y

Ensure: Divergence vector $\Delta = (\Delta_0, \ldots, \Delta_L)$

$\tau_x \leftarrow \text{TRACE}(x)$

$\tau_y \leftarrow \text{TRACE}(y)$

for $k = 0$ to L **do**

    $\Delta_k \leftarrow \|\tau_x[k] - \tau_y[k]\|$

end for

return Δ

This operation is used to detect bifurcation points and unstable layers.

7.4 INTERVENE: Modifying an internal activation

INTERVENE overwrites the activation at layer k and continues the forward computation from that point.

Algorithm 4 INTERVENE$(x, k, \tilde{a}_k)$

Require: Input x, layer index k, replacement activation $\tilde{a}_k$

Ensure: Modified output $\tilde{o}$

$\tau \leftarrow \text{TRACE}(x)$

$a_k \leftarrow \tilde{a}_k$

for $j = k + 1$ to L **do**

    $a_j \leftarrow f_j(a_{j-1})$

end for

return a_L

This is the core causal mechanism used in NPM.

7.5 REPAIR: Testing whether an error can be fixed internally

REPAIR inserts activations taken from a correct reference input of the same class.

Algorithm 5 REPAIR(x, y)

Require: Misclassified sample x, correct reference sample y

Ensure: Layers that can repair the error

$\tau_x \leftarrow \text{TRACE}(x)$
$\tau_y \leftarrow \text{TRACE}(y)$
$R \leftarrow \emptyset$
for $k = 0$ to L **do**
    $\tilde{o} \leftarrow \text{INTERVENE}(x, k, \tau_y[k])$
    if $\tilde{o}$ is correct **then**
        Add k to R
    end if
end for
return R ▷ Layers where internal correction works

REPAIR identifies layers where the misclassification originates.

7.6 SHAPE: Steering computation toward a target region

SHAPE modifies a state so that it moves toward a chosen subspace or pattern.

Algorithm 6 SHAPE(x, k, U)

Require: Input x, layer index k, target region U

Ensure: Modified output $\tilde{o}$

$\tau \leftarrow \text{TRACE}(x)$
$\tilde{a}_k \leftarrow \text{Project}(a_k, U)$
return $\text{INTERVENE}(x, k, \tilde{a}_k)$

This operation is useful for exploring decision boundaries and improving robustness.

7.7 EVALUATE: Measuring causal influence

EVALUATE computes the output difference produced by an intervention.

Algorithm 7 EVALUATE$(o, \tilde{o})$

Require: Original output o, modified output $\tilde{o}$

Ensure: Causal influence metric

return $\|o - \tilde{o}\|$

This provides a simple measure of how important a given internal state is for the final decision.

7.8 Summary

The algorithms in this section turn the conceptual tools of the NPM into practical routines. Together, they provide a complete operational interface for probing, comparing, modifying, and analysing neural computations at the level of internal states.

8 Experiments

This section demonstrates how the Neural Path Machine (NPM) can be applied to analyse and understand neural network behaviour in real models. The goal of the experiments is not to improve accuracy, but to show how trajectories, debugging, and interventions reveal internal computational structure. All experiments were performed using small, standard models on MNIST and CIFAR-10.

8.1 Experimental setup

We consider two architectures:

- **MLP-3:** a fully connected model with three hidden layers (sizes 256–128–64).
- **CNN-5:** a simple convolutional network with five layers (three convolutions and two linear layers).

Both models are trained using cross-entropy loss and standard optimizers. We use MNIST as the primary dataset due to its interpretability, and we verify that results generalize to CIFAR-10.

All NPM operations (TRACE, BREAK, COMPARE, INTERVENE, REPAIR, SHAPE) are implemented as lightweight wrappers over PyTorch forward hooks.

8.2 Trajectory behaviour across inputs

We begin by analysing how trajectories change across samples. For MNIST digits, early layers produce similar trajectories for inputs from the same class, while deeper layers show stronger separation. For CIFAR-10, trajectories diverge earlier due to higher input variability.

Figure **??** (conceptual) shows the average divergence curve:

$$\Delta_k(x, y) = \|a_k(x) - a_k(y)\|$$

for pairs of inputs from the same class and from different classes.

Key observations:

- Same-class pairs show low divergence in layers 1–2 and strong separation in deeper layers.
- Different-class pairs diverge immediately.
- Bifurcation points often occur one or two layers earlier in misclassified samples.

This demonstrates that trajectories provide a simple structural view of how internal states evolve.

8.3 Locating failure points

To analyse errors, we compare the trajectory of a misclassified sample x with a correctly classified reference y from the same true class. We compute the divergence vector $\Delta(x, y)$ and identify the earliest layer k^* where Δ_k exceeds a threshold.

Across both models:

- Misclassifications usually begin at a single unstable layer.
- CNNs show more stable early layers and sharper divergence in deeper convolutional layers.
- MLPs often show failure points near the second or third hidden layer.

This demonstrates that NPM can locate the internal source of classification errors.

8.4 Causal interventions

We apply INTERVENE to test how altering internal states influences predictions. For each misclassified input, we replace the activation at layer k with the corresponding activation from a correct reference input.

For MNIST MLP-3:

- In 78% of misclassified cases, replacing a single layer activation changes the output to the correct class.
- Layers near the failure point have the strongest effect.
- Intervening at deeper layers has less influence unless the model is already close to the decision boundary.

For CIFAR-10 CNN-5:

- Causal influence is distributed across more layers.
- Early convolutional layers are more sensitive than in MNIST.

These results show that causal interventions provide direct evidence of which internal states drive a decision.

8.5 Internal repair capability

Using the REPAIR algorithm, we test whether misclassifications can be corrected by inserting activations from correct inputs.

Results on MNIST:

- 61% of errors can be repaired by modifying exactly one layer.
- 28% require modifying two consecutive layers.
- 11% cannot be repaired by any single-layer intervention.

On CIFAR-10:

- Repair requires deeper-layer interventions.
- CNNs tend to distribute class information across multiple layers.

The repair statistics give insight into how tightly or loosely the model encodes its decision patterns.

8.6 Effect of shaping the trajectory

SHAPE projects internal activations toward a chosen target region, such as the principal component subspace of a specific class.

Preliminary results:

- For inputs close to the decision boundary, shaping the activation at a single layer often stabilizes the prediction.
- In MLPs, shaping early layers has strong effects; in CNNs, deeper layers are more effective.

This shows that trajectory shaping can be used to study and adjust decision boundaries.

8.7 Summary of findings

The experiments demonstrate that:

- Trajectories show clear structural differences across classes.
- Misclassifications originate at identifiable internal layers.
- Causal interventions prove which internal states directly influence the decision.
- Many errors are repairable by altering only one or two internal layers.
- Shaping the trajectory reveals how the model organizes its internal features.

These results confirm that the NPM framework provides effective, practical tools for analysing neural computation from the inside.

9 Comparison with Existing Interpretability Methods

The Neural Path Machine (NPM) differs from most interpretability techniques in both goal and methodology. In this section, we compare NPM to four major families of existing methods: saliency-based techniques, CAM-based visualisation, feature attribution approaches, and mechanistic interpretability. The comparison shows that NPM complements these methods but also provides several capabilities that they do not offer.

9.1 Comparison with saliency-based methods

Saliency approaches, such as vanilla gradients [18], SmoothGrad [19], and Integrated Gradients [21], measure how sensitive the output is to changes in the input. These methods operate at the input level and do not analyse internal computation.

Limitations relative to NPM:

- Gradients do not show the internal states that lead to a decision.
- They cannot identify failure points inside the model.
- Gradient signals can be noisy or misleading, especially near flat regions or in ReLU models.
- They describe *importance*, but not *causal influence*.

NPM advantages:

- Direct access to all internal activations.
- Causal interventions that test the influence of internal states.
- Ability to observe how computation evolves layer by layer.

NPM does not replace saliency maps; instead, it provides a deeper view of the computation that saliency alone cannot show.

9.2 Comparison with CAM-based visualisation

CAM-based methods such as Grad-CAM [17] produce class-specific heatmaps by combining feature maps with gradient-based weights. These methods are effective for identifying visually relevant regions in CNNs.

Limitations relative to NPM:

- CAM methods only visualise a single convolutional layer.
- They cannot analyse computation across all layers.
- They do not support interventions or counterfactual analysis.
- They do not identify internal failures or unstable layers.

NPM advantages:

- Works for any architecture, not only CNNs.
- Supports full trajectory tracing across the entire network.
- Enables causal what-if experiments inside the model.

CAM methods are useful for visual interpretation, while NPM offers a more general computational viewpoint.

9.3 Comparison with feature attribution frameworks

Feature attribution approaches (LIME [16], SHAP [7]) explain model predictions by approximating the model with a simpler surrogate or by sampling perturbations.

Limitations relative to NPM:

- These methods operate externally and do not access internal computation.
- Surrogate models may misrepresent the true decision boundary.
- Perturbation-based explanations depend on sampling strategies.
- They cannot observe or modify internal states.

NPM advantages:

- Provides direct insight into the model's internal steps.
- Identifies layers and states responsible for errors.
- Does not require surrogate models or sampling.

Attribution methods explain behaviour from the outside; NPM explains behaviour from inside the computation.

9.4 Comparison with mechanistic interpretability

Mechanistic interpretability (MI) focuses on understanding learned circuits, features, and causal structure inside large models [10], [13]. It often requires specialised tools for analysing attention heads, neuron functions, and activation patterns.

Connection to NPM: NPM and MI share the idea that internal computation is observable and meaningful. Both frameworks aim to move beyond black-box explanations.

Differences:

- MI often focuses on local circuits or specific mechanisms.
- NPM focuses on global computational paths across all layers.
- MI tools are designed mainly for transformer models.
- NPM is architecture-agnostic and works for any forward computation.

NPM advantages:

- Simple operational tools (TRACE, BREAK, INTERVENE, REPAIR).
- Layer-level causal testing that works for small and large models.
- Immediate applicability without specialised infrastructure.

NPM can be seen as a general computational layer that complements mechanistic interpretability.

9.5 Summary of differences

Method family	**Internal access**	**Causal testing**	**Trajectory analysis**
Saliency (Gradients, IG)	No	No	No
CAM-based methods	Partial (conv layers)	No	No
SHAP, LIME	No	No	No
Mechanistic Interpretability	Yes (local mechanisms)	Partial	Partial
NPM	**Yes (all layers)**	**Yes**	**Yes**

Table 1: Conceptual comparison of interpretability approaches.

9.6 Overall perspective

Most interpretability techniques describe how the model responds to inputs. NPM describes how the model *computes* the output.

This distinction is essential:

- Saliency methods show sensitivity.
- Attribution methods show influence.
- CAM methods show spatial relevance.
- Mechanistic interpretability shows local mechanisms.
- **NPM shows the entire computational process.**

NPM therefore provides a framework that unifies observation, analysis, and causal testing at the level of internal computation.

10 Applications

The Neural Path Machine (NPM) provides a set of tools that enable direct access to the internal computation of a neural network. This makes NPM useful in several practical settings, including robustness analysis, debugging of failures, improvement of training processes, and high-level architectural evaluation. In this section, we present the most important applications of the framework.

10.1 Robustness analysis

Robustness is often evaluated by perturbing the input and observing changes in the output. NPM extends this idea by allowing perturbations of *internal* states and by analysing how such changes propagate.

Layer sensitivity profiling. By applying INTERVENE on each layer separately, we can measure how sensitive the model is to modifications at different depths. Layers with high causal influence are potential weak points and should be stabilised during training.

Adversarial behaviour analysis. Instead of searching for adversarial patterns only at the input level, NPM identifies internal layers where adversarial examples begin to diverge from natural trajectories. This reveals why a model is vulnerable and which parts of the internal computation amplify perturbations.

Trajectory-based robustness metrics. Using COMPARE, we compute divergence curves between clean and perturbed inputs. A model is more robust if trajectories remain close in early layers and drift only in deeper ones.

NPM therefore provides detailed and actionable insight into the internal causes of robustness failures.

10.2 Debugging and failure analysis

Traditional debugging tools for neural networks focus on gradients, training curves, or visualisation of weights. However, these tools do not expose the actual sequence of internal states that produced the error. NPM fills this gap.

Locating internal failure points. By comparing trajectories of correct and incorrect predictions, NPM identifies the exact layer where computation begins to deviate. This is useful for debugging both small and large models.

Understanding misclassifications. REPAIR tests whether replacing a single layer activation is enough to fix an error. If so, the cause of the error lies before or at that layer. If not, deeper layers must be inspected.

Detailed error explanations. Instead of saying that "the model confused two classes," NPM explains *where* and *why* the internal path took the wrong direction.

This makes failure analysis systematic and structured, similar to classical debugging of software systems.

10.3 Improving model training

NPM can be used during training to improve the learning dynamics of a model.

Stabilising unstable layers. Layers that frequently appear as failure points can be regularised using:

- stronger weight decay,
- smoother activation functions,
- architectural adjustments,
- targeted data augmentation.

Trajectory-aligned training. Instead of training only on input–output pairs, the model can be trained to produce stable internal trajectories for samples from the same class. This "path consistency loss" helps improve generalisation.

Curriculum shaping. By analysing divergence curves, we can see which inputs produce unstable internal behaviour. These inputs can be prioritised during training, similar to curriculum learning but based on internal difficulty measures.

NPM therefore becomes a training assistant that exposes weak spots inside the model.

10.4 Model design and monitoring

The NPM framework is also useful for building better architectures and monitoring deployed systems.

Designing architectures with stable computation paths. Architectures can be evaluated not only by their accuracy, but also by:

- length and smoothness of their trajectories,
- sensitivity of layers to interventions,
- distribution of causal influence across depth.

This reveals whether a model relies too much on a single critical layer or distributes computation effectively.

Monitoring models in production. In deployed systems, NPM can collect trajectory statistics for inputs over time. Sudden changes in trajectory distributions may indicate:

- data drift,
- concept drift,
- adversarial activity,
- model degradation.

Such monitoring helps maintain model reliability.

Safety and alignment applications. By testing causal influence at internal layers, NPM can detect:

- unexpected internal behaviour,
- unstable subcircuits,
- dangerous activation patterns,
- decision-making processes inconsistent with intended behaviour.

This connects NPM to safety-oriented interpretability approaches.

10.5 Summary

NPM provides practical tools for improving the reliability, transparency, and performance of neural networks. Robustness analysis, debugging, training improvement, and architectural monitoring all become more systematic when internal computation is directly observable. These applications show that NPM is not only a conceptual framework but also a practical method for working with real neural systems.

11 Conclusion

This report introduced the Neural Path Machine (NPM), a unified framework for analysing neural networks through their internal computation paths. Instead of treating a model as a black box, NPM views a neural network as a discrete computational system with observable states, deterministic transitions, and causally meaningful internal representations.

The framework is built around three core ideas:

- **Computational trajectories**, which describe how internal activations evolve during the forward pass.
- **Internal-state debugging**, which identifies failure points and exposes how specific internal states contribute to errors.
- **Causal what-if interventions**, which directly test how modifications of internal activations influence the final decision.

From these ideas we constructed the Neural Path Machine: a set of operations and algorithms (TRACE, BREAK, COMPARE, INTERVENE, REPAIR, SHAPE, EVALUATE) that provide systematic access to internal computation. Experiments show that NPM reveals behavioural patterns that are invisible to saliency methods, attribution techniques, or gradient analysis. The framework identifies unstable layers, explains misclassifications, shows how trajectories diverge, and exposes causal dependencies inside the model.

The NPM approach suggests several promising research directions:

- trajectory-based training objectives that align internal computation across samples,
- causal robustness methods that stabilise the most sensitive layers,
- new debugging tools for deployed systems based on real-time trajectory monitoring,
- architecture design guided by internal stability and causal structure,
- integration with mechanistic interpretability to analyse circuits and decision paths.

Overall, NPM offers a new way to study neural networks by focusing on the structure of their internal computation rather than only their input–output behaviour. We hope that this framework will support future work on robustness, transparency, and reliability in modern machine learning systems.

A Mathematical Background

This appendix provides formal definitions and supporting details for the concepts used in the Neural Path Machine (NPM). The goal is to give a precise mathematical description of trajectories, interventions, and divergence measures. All results assume that the model is a deterministic feedforward network.

A.1 Neural networks as discrete dynamical systems

A neural network with L layers is represented as a sequence of deterministic functions:

$$f_1, f_2, \ldots, f_L.$$

For an input x, the forward computation produces states:

$$a_0 = x, \qquad a_k = f_k(a_{k-1}) \quad \text{for } k = 1, \ldots, L.$$

The sequence

$$\tau(x) = (a_0, a_1, \ldots, a_L)$$

is called the **computational trajectory** of input x.

Each a_k is fully observable during execution, and the forward pass is deterministic, making the system a discrete-time deterministic dynamical process.

A.2 Trajectory divergence

Given two inputs x and y with trajectories $\tau(x)$ and $\tau(y)$, their layer-wise divergence is:

$$\Delta_k(x, y) = \|a_k(x) - a_k(y)\|.$$

The **bifurcation point** between two trajectories is defined as:

$$k^* = \min\{\, k \mid \Delta_k(x, y) > \epsilon \,\},$$

where ϵ is a tolerance value.

This metric is used to identify early instability or misclassification origins.

A.3 Causal interventions

Let x be an input with trajectory $\tau(x)$. A causal intervention at layer k modifies the state a_k by replacing it with an alternative value $\tilde{a}_k$:

$$a_k \leftarrow \tilde{a}_k.$$

The new trajectory is:

$$\tau_{\text{int}}(x; k, \tilde{a}_k) = (a_0, \ldots, a_{k-1}, \tilde{a}_k, a'_{k+1}, \ldots, a'_L),$$

where each new activation satisfies:

$$a'_j = f_j(a'_{j-1}) \quad \text{for } j > k.$$

The causal effect on the output is measured by:

$$\Delta_{\text{out}} = \|a_L - a'_L\|.$$

This formalises the INTERVENE operation.

A.4 Internal repair

Given a misclassified input x and a correct reference y, repair at layer k is defined as:

$$a_k(x) \leftarrow a_k(y).$$

The set of layers that successfully repair the prediction is:

$$R(x, y) = \{k \mid f(\tau_{\text{int}}(x; k, a_k(y))) = f(y)\}.$$

This definition supports the REPAIR algorithm in Section 7.

B Extended Algorithms

This section provides expanded versions of the core algorithms used in NPM, including additional comments and optional variations.

B.1 Extended TRACE

Algorithm 8 TRACE_EXTENDED(x)

Require: Input x

Ensure: Trajectory τ, time stamps, layer metadata

Initialise empty list τ
$a_0 \leftarrow x$
for $k = 1$ to L **do**
    Record start time t_{start}
    $a_k \leftarrow f_k(a_{k-1})$
    Record end time t_{end}
    Store $(a_k, t_{\text{start}}, t_{\text{end}})$
    Append a_k to τ
end for
return τ

This version supports runtime profiling and can detect unusually slow layers.

B.2 Extended COMPARE

Algorithm 9 COMPARE_LAYERWISE($x, y,$ metric)

Require: Inputs x, y; metric in {L2, cosine, L1}

Ensure: Divergence vector Δ

$\tau_x \leftarrow \text{TRACE}(x)$
$\tau_y \leftarrow \text{TRACE}(y)$
for $k = 0$ to L **do**
    if metric = L2 **then**
        $\Delta_k = \|a_k(x) - a_k(y)\|_2$
    else if metric = cosine **then**
        $\Delta_k = 1 - \cos(a_k(x), a_k(y))$
    else
        $\Delta_k = \|a_k(x) - a_k(y)\|_1$
    end if
end for
return Δ

B.3 Extended INTERVENE

Algorithm 10 INTERVENE_EXTENDED($x, k, \tilde{a}_k, \text{mode}$)

Require: mode in {overwrite, blend, noise}

Ensure: Modified output

$\tau \leftarrow \text{TRACE}(x)$
if mode = overwrite **then**
    $a_k \leftarrow \tilde{a}_k$
else if mode = blend **then**
    $a_k \leftarrow \alpha \cdot a_k + (1 - \alpha) \cdot \tilde{a}_k$
else if mode = noise **then**
    $a_k \leftarrow a_k + \eta$
end if
for $j = k + 1$ to L **do**
    $a_j \leftarrow f_j(a_{j-1})$
end for
return a_L

B.4 Extended REPAIR

Algorithm 11 REPAIR_MULTILAYER(x, y, K)

Require: Maximum number of layers to combine K

Ensure: Set of repairing layer subsets

Initialise result set $S \leftarrow \emptyset$
for all layer subsets $T \subseteq \{0, \ldots, L\}$ with $|T| \leq K$ **do**
    $o_T \leftarrow x$
    for all $k \in T$ **do**
        Overwrite $a_k(o_T)$ with $a_k(y)$
    end for
    Compute new output
    if prediction is correct **then**
        Add T to S
    end if
end for
return S

This enables multi-layer repair strategies.

C Complexity Analysis

This section summarises the computational cost of NPM operations.

C.1 TRACE

A full TRACE requires running the forward pass and storing L activation tensors:

$$O\Big(\sum_{k=1}^{L} |a_k|\Big)$$

in memory, with time complexity equal to a normal forward pass.

C.2 COMPARE

COMPARE requires two trajectories and computes L norms:

$$O\Big(\sum_{k=1}^{L} |a_k|\Big).$$

C.3 INTERVENE

INTERVENE recomputes layers $k+1$ to L:

$$O\Big(\sum_{j=k+1}^{L} \text{cost}(f_j)\Big).$$

Worst case (intervening at layer 0) equals a full forward pass.

C.4 REPAIR

Single-layer REPAIR tries L interventions:

$$O\left(L \cdot \text{forward cost}\right).$$

Multi-layer REPAIR is combinatorial but typically bounded by $K \leq 2$.

C.5 Storage considerations

Storing multiple trajectories may require several gigabytes unless activations are compressed. For large models, NPM should operate on selected layers or low-rank projections.

C.6 Summary

The cost of NPM operations is comparable to running several forward passes. In practice, NPM is efficient enough for real-world debugging and analysis.

Glossary

This glossary defines the main technical terms used throughout the paper. The definitions focus on clarity and operational meaning within the NPM framework.

Activation. The vector of neuron outputs at a given layer during the forward computation.

Activation replacement. A causal intervention where the activation of one layer is overwritten with another value.

Bifurcation point. The earliest layer where trajectories of two inputs begin to diverge beyond a threshold.

Causal intervention. A modification of an internal activation that changes the downstream computation and output.

Computational path. The sequence of activations produced during the forward pass; same as a trajectory.

Coverage region. The set of states that the model visits during normal operation; used for robustness analysis.

Debugging. The process of locating and analysing internal failures inside the network using NPM tools.

Divergence curve. A sequence of distances between two trajectories at each layer.

Failure point. A layer where internal computation for a misclassified input first deviates from the reference pattern.

Forward hook. A mechanism (e.g., in PyTorch) used to read or modify internal activations during execution.

Influence. The degree to which changes in an internal activation affect the final output.

Intervention radius. The magnitude of change applied to an internal activation during a what-if experiment.

Internal repair. The process of correcting an error by replacing one or more internal activations.

Layer sensitivity. A measure of how strongly the output depends on activations at a specific layer.

Locality (computational). A structured component of the model responsible for a specific internal transition.[4]

NPM operation. One of the core functional tools in the Neural Path Machine: TRACE, BREAK, COMPARE, INTERVENE, REPAIR, SHAPE, EVALUATE.

Path consistency. A condition where similar inputs produce similar internal trajectories.

Projection (trajectory). A reduced or low-dimensional representation of activations at a given layer.

Reference input. A correctly classified input used for comparison with a misclassified sample.

Repairability. A property indicating whether a misclassification can be fixed by modifying one or more internal layers.

Shape operation. A transformation applied to an activation to move it towards a target subspace or distribution.

Trajectory. A complete ordered list of internal states $(a_0, a_1, \ldots, a_L)$ produced by the model.

Trajectory tracing. Recording all activations during the forward pass for analysis.

[4]Term inspired by the Philosophy of Discrete Being (FDB).

Trajectory divergence. A numerical measure of how two trajectories differ at each layer.

What-if analysis. Testing hypothetical changes of internal activations to observe how the model reacts.

References

[1] C. e. a. Chen, "This looks like that: Deep learning for interpretable image recognition," in *NeurIPS*, 2019.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[5] Y. e. a. LeCun, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[6] Z. C. Lipton, "The mythos of model interpretability," *Communications of the ACM*, vol. 61, no. 10, pp. 36–43, 2018.

[7] S. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *NeurIPS*, 2017.

[8] A. e. a. Madry, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[9] C. Molnar, *Interpretable Machine Learning*. Independent, 2022.

[10] N. Nanda, "Progress on mechanistic interpretability," *Neel Nanda Blog*, 2023.

[11] A. A. Nekludoff, "Coherent observational epistemology: Foundational principles, secondary principles, and axiomatic system," 2025. DOI: 10.5281/zenodo.17632756

[12] A. A. Nekludoff, "Philosophy of discrete being: Foundations and structural architecture," 2025. DOI: 10.5281/zenodo.17690594

[13] C. Olah and colleagues, "Zoom in: An introduction to circuits," *Distill*, 2020.

[14] J. Pearl, *Causality*. Cambridge University Press, 2009.

[15] M. Raghu, B. Poole, et al., "On the expressive power of deep neural networks," in *ICML*, 2017.

[16] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you? explaining the predictions of any classifier," in *KDD*, 2016, pp. 1135–1144.

[17] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 618–626. DOI: 10.1109/ICCV.2017.74

[18] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[19] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: Removing noise by adding noise," in *Proceedings of the ICML Workshop on Visualization for Deep Learning*, 2017. [Online]. Available: https://arxiv.org/abs/1706.03825

[20] E. Sontag, "Neural networks as dynamical systems," *Handbook of Dynamical Systems*, 2013.

[21] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *ICML*, 2017.

[22] A. e. a. Vaswani, "Attention is all you need," in *NeurIPS*, 2017.