nnterp: A Standardized Interface for Mechanistic Interpretability of Transformers

Clément Dumas
École Normale Supérieure Paris-Saclay, Université Paris-Saclay
clement.dumas@ens-paris-saclay.fr

Abstract

Mechanistic interpretability research requires reliable tools for analyzing transformer internals across diverse architectures. Current approaches face a fundamental tradeoff: custom implementations like TransformerLens ensure consistent interfaces but require coding a manual adaptation for each architecture, introducing numerical mismatch with the original models, while direct HuggingFace access through NNsight preserves exact behavior but lacks standardization across models. To bridge this gap, we develop `nnterp`, a lightweight wrapper around NNsight that provides a unified interface for transformer analysis while preserving original HuggingFace implementations. Through automatic module renaming and comprehensive validation testing, `nnterp` enables researchers to write intervention code once and deploy it across 50+ model variants spanning 16 architecture families. The library includes built-in implementations of common interpretability methods (logit lens, patchscope, activation steering) and provides direct access to attention probabilities for models that support it. By packaging validation tests with the library, researchers can verify compatibility with custom models locally. `nnterp` bridges the gap between correctness and usability in mechanistic interpretability tooling.[1]

1 Introduction

Mechanistic interpretability research aims to reverse-engineer the computational mechanisms within neural networks [Elhage et al., 2021, Olah et al., 2020]. For transformer language models, this requires tools that can reliably access and modify internal representations across diverse architectures. However, the field faces a fundamental engineering challenge: balancing implementation correctness with practical usability across the rapidly expanding landscape of transformer variants.

Two dominant paradigms have emerged. TransformerLens [Nanda and Bloom, 2022] provides a clean, unified interface by reimplementing transformer architectures from scratch. This ensures consistent module naming and reliable hooks but requires manual implementation for each new architecture, may introduce subtle differences from original models, and cannot leverage architecture-specific optimizations. Conversely, NNsight [Fiotto-Kaufman et al., 2024] operates directly on HuggingFace implementations, preserving exact model behavior and supporting any architecture HuggingFace provides. However, this approach inherits the fragmentation of HuggingFace's naming conventions—accessing layers requires `model.transformer.h` for GPT-2 but `model.model.layers` for LLaMA—and leaves researchers vulnerable to breaking changes in transformer implementations. For example, since HugginFace transformers 4.54, Qwen and Llama

[1] Repository available at https://github.com/Butanium/nnterp, documentation at https://butanium.github.io/nnterp/. Install with `pip install nnterp`.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Mechanistic Interpretability.

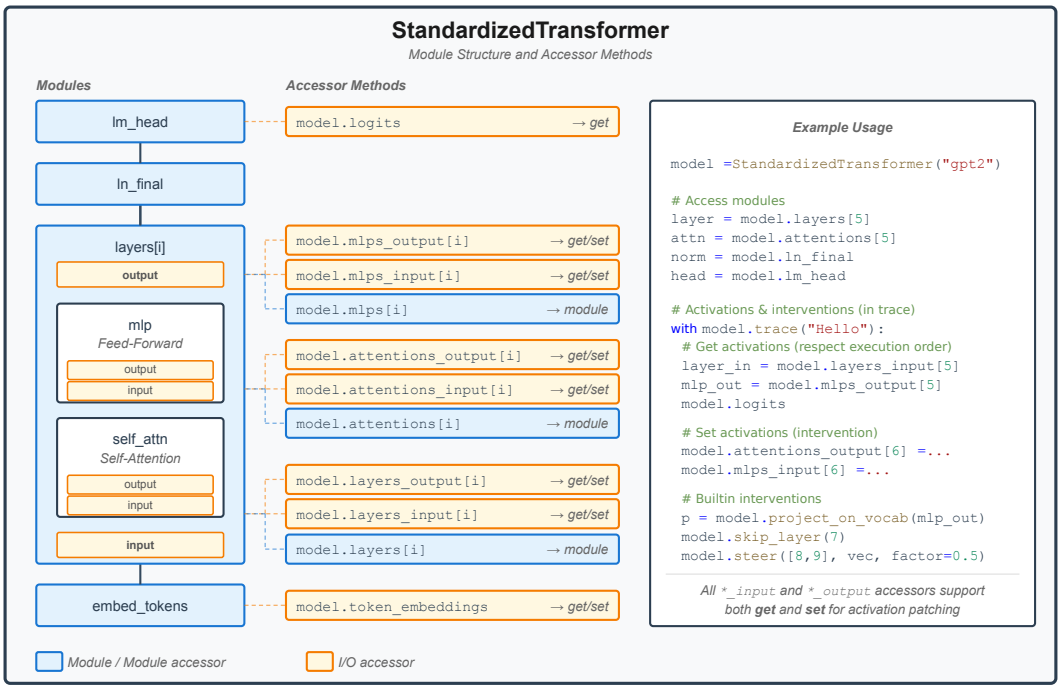

Figure 1: `nnterp` provides a standardized interface for transformer models. **Left:** It renames transformer modules to a consistent naming scheme (`layers`, `self_attn`, `mlp`, etc.). **Middle:** It provides accessor methods for internal activations, with `get` and `set` for all `*_input` and `*_output`. This handles whether the module returns a tuple or a single tensor. **Right:** Example usage for intervention and analysis.

layers return activation tensors instead of tuples, which caused silent bugs in many interpretability experiments.

This fragmentation creates significant friction. Researchers must either commit to a single tool's limitations or maintain parallel codebases for different architectures. As mechanistic interpretability increasingly emphasizes cross-model validation and scalability to large models requiring optimizations, neither approach alone suffices.

We present `nnterp`, a lightweight library that bridges this gap by providing a standardized interface atop NNsight's HuggingFace integration. Through systematic module renaming and validation, `nnterp` enables researchers to write `model.layers_output[5]` consistently across GPT-2, LLaMA, Gemma, and other architectures while preserving exact HuggingFace behavior. Key contributions include:

- A unified API for accessing transformer internals (layers, attention, MLP outputs) that works identically across 50+ model variants from 16 architecture families.
- Automatic validation upon model loading that verifies module shapes and intervention correctness. This caught the HuggingFace transformers 4.54 bug mentioned earlier day-1.
- Built-in implementations of common interpretability methods (logit lens, patchscope, steering) that work across all supported models.
- Direct access to attention probabilities for compatible architectures through NNsight's intermediate variable tracking.
- Packaged test suite enabling local verification of custom models.

2 Background and Related Work

Mechanistic Interpretability Tooling. Early mechanistic interpretability relied on manual PyTorch hooks to intercept activations [Elhage et al., 2021]. As the field matured, specialized libraries emerged to streamline common operations. Pyvene [Wu et al., 2024] provides a declarative frame-

work for causal interventions, while TransformerLens [Nanda and Bloom, 2022] offers a unified interface through custom implementations. NNsight [Fiotto-Kaufman et al., 2024] takes a different approach, wrapping existing models with a tracing system that enables interventions while preserving original implementations.

The Standardization Challenge. HuggingFace Transformers [Wolf et al., 2020] has become the de facto repository for transformer implementations, but its design prioritizes flexibility over consistency. Each architecture follows its own naming conventions and internal structure. This heterogeneity reflects genuine architectural differences but complicates systematic analysis. Recent work on model diffing [Lindsey et al., 2024] and circuit discovery [Conmy et al., 2023] highlights the need for tools that work reliably across architectures.

3 nnterp Design and Implementation

`nnterp` extends NNsight's `LanguageModel` class with a `StandardizedTransformer` that automatically renames modules to follow a consistent naming scheme and provides some I/O accessors as shown in Figure 1. The module renaming system is implemented using the `rename` argument from the `NNsight` class, which allow to pass a dictionary of original names and new names. Different architectures require different renaming strategies. GPT-2's `transformer.h`, `attn`, `transformer.ln_f` becomes `layers`, `self_attn`, `ln_final`, while LLaMA's structure just moves `model.layers` to `layers`. `nnterp` maintains a configuration system that maps each architecture class to its renaming rules:

```
model = StandardizedTransformer("gpt2")
# Internally applies:
rename = {
  "transformer": "model", "h": "layers", "model.layers": "layers",
  "attn": "self_attn", "transformer.ln_f": "ln_final"
}
```

`nnterp` also provides `model.{layers/mlps/attentions}_input/output[layer_idx]` which allow to get and set the input and output of the specified layer. This should be preferred over using e.g. `model.layers[layer_idx].output` as this can be a tensor or a tuple depending on the architecture, while `model.layers_output[layer_idx]` always get/set the output tensor.

This standardization leverages NNsight's module renaming capability while maintaining full compatibility with NNsight's intervention API (the modules are still available under their original names). Researchers can use `nnterp`'s simplified interface for common operations or drop down to raw NNsight for advanced use cases.

Attention probabilities access. To enable access to attention probabilities – which requires to use the slower eager attention implementation – `enable_attention_probs=True` needs to be passed to the `StandardizedTransformer` class. Once enabled, the attention probabilities can be accessed and set using the `model.attention_probabilities[layer_idx]` property. This relies on NNsight's `source` feature, which allows to access intermediate variables in the forward pass. This means, that this is very implementation sensitive, and may break with new HuggingFace releases, e.g. if the name of the attention variable changes. The attention probabilities hookpoint is therefore not available on all models. See Appendix A for how to add support for new models.

Validation guarantees. Upon initialization, `nnterp` runs automatic tests to verify: (1) module outputs have expected shapes, (2) attention probabilities sum to 1, (3) interventions affect outputs, and (4) layer skip operations preserve causality. These tests catch common issues like incorrect module identification or incompatible attention implementations. The validation suite ships with the package, allowing researchers to test custom models locally via `python -m nnterp run_tests`.

4 Extra features

Built-in interventions. `nnterp` implements common interpretability methods that work across all supported models. **Logit Lens** [nostalgebraist, 2020] projects hidden states through the unembedding to see intermediate predictions. **Patchscope** [Ghandeharioun et al., 2024] replaces activations

from one context into another. **Activation Steering** adds steering vectors at specified layers. All methods use the same unified API across architectures.

Prompt Management. `nnterp` provides a `Prompt` class that allows to track probabilities of specific target tokens in different categories. The easiest way to create a `Prompt` is to use the `Prompt.from_strings` method, which takes a prompt string and a dictionary of categories and target strings. The tracked tokens of each category is the set of all first tokens of the target strings with and without beginning of word. E.g for `["London", "Lyon"]`, the tracked tokens could be `["_London", "Lon", "_Lyo", "Ly"]`.

```
prompt = Prompt.from_strings(
    "The capital of France is",
    {"target": "Paris", "fake": ["London", "Lyon"]},
    tokenizer
)
results = run_prompts(model, [prompt])
# Returns probabilities for each category
{"target": torch.Tensor([0.7]), "fake": torch.Tensor([0.1])}
```

5 Empirical Validation

`nnterp` supports 21 architecture families[2]. `nnterp` adds minimal overhead to NNsight's already efficient implementation. NNsight's performance analysis [Fiotto-Kaufman et al., 2024] shows it matches or exceeds TransformerLens speed while using less memory. Since `nnterp` is a thin wrapper providing only interface standardization, it inherits these performance characteristics.

6 Discussion and Future Work

Limitations. `nnterp`'s validation tests provide sanity checks rather than formal correctness guarantees. While they catch common issues, subtle bugs in attention probability hooks or module identification may persist. The library also inherits NNsight's limitations, including incompatibility with some attention implementations (e.g., Flash Attention for attention probabilities).

Impact on Research Workflow. By separating interface standardization from implementation details, `nnterp` enables more reproducible mechanistic interpretability research. Researchers can share intervention code that works across models, facilitating replication and cross-architecture validation. The packaged test suite ensures that even custom models can be verified for compatibility.

Future Directions. Future work includes automated architecture detection, support for non-causal and encoder-decoder architectures, access to attention KQV and MLP intermediate activations and access to MoE router's logits. We are also exploring integration with NNsight itself and how to support remote execution via NDIF, which allow researcher to run their NNsight experiments on remote machines.

7 Conclusion

`nnterp` demonstrates that the tradeoff between implementation correctness and interface usability in mechanistic interpretability tooling is not fundamental. By leveraging NNsight's model wrapping capabilities and adding systematic validation, we provide researchers with both reliable access to exact HuggingFace implementations and a consistent interface across architectures. As mechanistic interpretability scales to larger models and broader architectural diversity, tools that balance correctness with usability become essential. `nnterp` represents a step toward more robust and reproducible interpretability research.

[2]See Appendix B for the full list.

References

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 2020. doi: 10.23915/distill.00024.001. https://distill.pub/2020/circuits/zoom-in.

Neel Nanda and Joseph Bloom. Transformerlens. https://github.com/TransformerLensOrg/TransformerLens, 2022.

Jaden Fiotto-Kaufman, Alexander R Loftus, Eric Todd, Jannik Brinkmann, Caden Juang, Koyena Pal, Can Rager, Aaron Mueller, Samuel Marks, Arnab Sen Sharma, Francesca Lucchetti, Michael Ripa, Adam Belfki, Nikhil Prakash, Sumeet Multani, Carla Brodley, Arjun Guha, Jonathan Bell, Byron Wallace, and David Bau. Nnsight and ndif: Democratizing access to foundation model internals. *arXiv*, 2024. URL https://arxiv.org/abs/2407.14561.

Zhengxuan Wu, Atticus Geiger, Aryaman Arora, Jing Huang, Zheng Wang, Noah Goodman, Christopher Manning, and Christopher Potts. pyvene: A library for understanding and improving PyTorch models via interventions. In Kai-Wei Chang, Annie Lee, and Nazneen Rajani, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations)*, pages 158–165, Mexico City, Mexico, June 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.naacl-demo.16.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Jack Lindsey, Adly Templeton, Jonathan Marcus, Thomas Conerly, Joshua Batson, and Christopher Olah. Sparse crosscoders for cross-layer features and model diffing. *Transformer Circuits Thread*, 2024. URL https://transformer-circuits.pub/2024/crosscoders/index.html.

Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2304.14997.

nostalgebraist. interpreting GPT: the logit lens. LessWrong, August 2020. URL https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens.

Asma Ghandeharioun, Avi Caciularu, Adam Pearce, Lucas Dixon, and Mor Geva. Patchscopes: A unifying framework for inspecting hidden representations of language models. In *Forty-first International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2401.06102.

A Adding Support for Custom Models

`nnterp` uses a standardized naming convention to provide a unified interface across transformer architectures. When a model doesn't follow the expected naming patterns, researchers can use `RenameConfig` to map custom module names to the standardized interface.

A.1 Target Structure

nnterp expects all models to follow this structure:

```
StandardizedTransformer
|-- embed_tokens
|-- layers[i]
|   |-- self_attn
|   `-- mlp
|-- ln_final
`-- lm_head
```

Models are automatically renamed to match this pattern using built-in mappings. The library provides convenient accessors: `layers_input[i]`, `layers_output[i]` for layer I/O, `attentions[i]`, `attentions_input[i]`, `attentions_output[i]` for attention, and `mlps[i]`, `mlps_input[i]`, `mlps_output[i]` for MLP components.

A.2 Basic RenameConfig Usage

When automatic renaming fails, create a custom `RenameConfig` specifying module names in the original model:

```python
from nnterp import StandardizedTransformer
from nnterp.rename_utils import RenameConfig

rename_config = RenameConfig(
    model_name="custom_transformer",
    layers_name="custom_layers",
    attn_name="custom_attention",
    mlp_name="custom_ffn",
    ln_final_name="custom_norm",
    lm_head_name="custom_head"
)

model = StandardizedTransformer(
    "your-model-name",
    rename_config=rename_config
)
```

For nested modules, use dot notation (e.g., `layers_name="transformer.encoder_layers"`). Multiple alternative names can be provided as lists (e.g., `attn_name=["attention", "self_attention"]`).

A.3 Implementing Attention Probabilities

Attention probabilities support requires implementing `AttnProbFunction`. Here's the GPT-J implementation:

```python
from nnterp.rename_utils import AttnProbFunction

class GPTJAttnProbFunction(AttnProbFunction):
    def get_attention_prob_source(self, attention_module,
                                  return_module_source=False):
        if return_module_source:
            return attention_module.source.self__attn_0.source
        return attention_module.source.self__attn_0.source\
               .self_attn_dropout_0

model = StandardizedTransformer(
    "yujiepan/gptj-tiny-random",
    enable_attention_probs=True,
    rename_config=RenameConfig(
        attn_prob_source=GPTJAttnProbFunction()
    )
)
```

The process involves: (1) using `model.scan()` to explore the forward pass, (2) locating where attention probabilities are computed (typically after dropout), (3) implementing the hook via `AttnProbFunction`, and (4) testing with dummy inputs to verify shapes and normalization.

A.4 Validation

`nnterp` automatically validates configurations during initialization, checking: module naming correctness, tensor shapes at each layer, attention probabilities normalization (if enabled), and I/O compatibility. Manual validation can be performed using `model.trace()` to verify expected tensor shapes and properties.

B Model coverage

The following model classes were tested and work with `nnterp`:

- `BloomForCausalLM`
- `BloomModel`
- `Ernie4_5_MoeForCausalLM`
- `GPT2LMHeadModel`
- `GPTBigCodeForCausalLM`
- `GPTJForCausalLM`
- `Gemma2ForCausalLM`
- `Gemma3ForCausalLM`
- `Gemma3ForConditionalGeneration`
- `GemmaForCausalLM`
- `Glm4ForCausalLM`
- `Glm4MoeForCausalLM`
- `LlamaForCausalLM`
- `MistralForCausalLM`
- `MixtralForCausalLM`
- `OPTForCausalLM`
- `Phi3ForCausalLM`
- `Qwen2ForCausalLM`
- `Qwen3ForCausalLM`
- `Qwen3MoeForCausalLM`
- `SeedOssForCausalLM`
- `SmolLM3ForCausalLM`
- `DbrxForCausalLM`
- `GptOssForCausalLM`
- `Qwen2MoeForCausalLM`
- `StableLmForCausalLM`

Support for attention probabilities is still a work in progress for the following model classes:

- `DbrxForCausalLM`
- `GptOssForCausalLM`
- `Qwen2MoeForCausalLM`
- `StableLmForCausalLM`