AI Steerability 360: A Toolkit for Steering Large Language Models

Erik Miehling, Karthikeyan Natesan Ramamurthy, Praveen Venkateswaran, Irene Ko, Pierre Dognin, Moninder Singh, Tejaswini Pedapati, Avinash Balakrishnan, Matthew Riemer, Dennis Wei, Inge Vejsbjerg, Elizabeth M. Daly, Kush R. Varshney

IBM Research

{erik.miehling@, knatesa@us., krvarshn@us.}ibm.com

Abstract

The AI Steerability 360 toolkit is an extensible, open-source Python library for steering LLMs. Steering abstractions are designed around four model control surfaces: input (modification of the prompt), structural (modification of the model's weights or architecture), state (modification of the model's activations and attentions), and output (modification of the decoding or generation process). Steering methods exert control on the model through a common interface, termed a steering pipeline, which additionally allows for the composition of multiple steering methods. Comprehensive evaluation and comparison of steering methods/pipelines is facilitated by use case classes (for defining tasks) and a benchmark class (for performance comparison on a given task). The functionality provided by the toolkit significantly lowers the barrier to developing and comprehensively evaluating steering methods. The toolkit is Hugging Face native and is released under an Apache 2.0 license at https://github.com/IBM/AISteer360.

1 Introduction

Steering a large language model refers to any lightweight, deliberate control of the model's behavior (Liang et al., 2024; Miehling et al., 2025; Vafa et al., 2025; Chang et al., 2025). There are numerous methods across a variety of mechanisms for exercising such control, including through prompting strategies (Brown et al., 2020; Zhou et al., 2022), modifying model internals via weights/architecture (Meng et al., 2022; Ilharco et al., 2022; Fierro and Roger, 2025) or internal model states (Dathathri et al., 2019; Li et al., 2023; Liu et al., 2023; Rimsky et al., 2024; Lee et al., 2024; Dunefsky and Cohan, 2025; Vu and Nguyen, 2025), or intervening at decoding time (Krause et al., 2021; Liu et al., 2021; Yang and Klein, 2021; Deng and Raffel, 2023; Ko et al., 2024; Huang et al., 2025).

With the growing number of steering methods, it is becoming increasingly challenging to understand how the methods differ, let alone which are most appropriate for which use cases. Steering methods are often designed under their own semantics and requirements, making direct comparison difficult. Additionally, steering in practice often consists of multiple "stacked" operations, e.g., SFT followed by DPO, DPO followed by CoT prompting, etc., complicating attribution of output to intervention.

Our toolkit provides a unified interface for steering that enables the construction of steering methods, via reusable abstractions, performance comparison of steering methods on use cases, and analysis of steering trade-offs, i.e., behaviors that were not targeted but were nevertheless modified. Steering methods, referred to as *controls*, are defined across four interfaces of the model (input, structural, state, and output), depending on what part of the model the control influences. One of the core abstractions of the toolkit, termed a *steering pipeline*, provides the surface by which the control interacts with the model and additionally allows for multiple controls to be composed into a single model operation. On evaluation, the toolkit provides use case and benchmark classes to define tasks and enable comparison of steering pipelines under both fixed settings (control parameters are pinned) and variable settings (control parameters are swept over a region). The toolkit simplifies the development of steering methods and enables comparisons of steering methods in a controlled and consistent manner; something currently missing from the community.

The remainder of this paper will focus on how the toolkit is used for both steering and evaluation under a selection of the implemented controls (primarily state-based controls). For a comprehensive list of examples, please see our repo's notebooks.

Related Work. Existing frameworks for steering LLMs vary significantly in their coverage of steer-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 436–444

ing methods, with current tools generally limited to individual control surfaces. A variety of libraries/-tools for state-based control have been released (especially recently), including TransformerLens (Nanda and Bloom, 2022), baukit (Bau, 2022), representation engineering (Zou et al., 2023), pyvene (Wu et al., 2024), EasySteer (Xu et al., 2025a), EasyEdit2 (Xu et al., 2025b), DialZ (Siddique et al., 2025), the steerability tool from Chang et al. (2025), vLLM.hook (Ko and Chen, 2026), and the interactive *steer* feature on Neuronpedia (Lin, 2023). Weight-based controls (e.g., fine-tuning) are also well supported via established tooling from Hugging Face (Wolf et al., 2020; von Werra et al., 2020; Mangrulkar et al., 2022; Hugging Face, 2024), PyTorch (PyTorch, 2024; PyTorch Lightning Team, 2025), Axolotl (Axolotl, 2023), and LlamaFactory (Zheng et al., 2024). Notably absent from current frameworks is support for decoding-time alignment/steering and, more broadly, a unified framework that spans all control surfaces. Additionally, while the question of interactions among compositions of algorithms has been investigated in other domains, e.g., fair ML (Nagireddy et al., 2023), the current treatment in the steering literature is limited to activation steering (Todd et al., 2023; Scalena et al., 2024; Abreu et al., 2025; Han et al., 2026).

Contribution. Our toolkit addresses these gaps with two core contributions: **i)** implementations of steering methods across multiple model control surfaces under a common interface (realized through steering pipelines), with support for both individual controls and their compositions, and **ii)** use case and benchmark classes for defining tasks and evaluating/comparing steering methods/pipelines across the full range of model control, as well as providing functionality for studying steering trade-offs via variable control specifications.

2 Taxonomy of Steering Methods

The core API of the toolkit is designed around a steering taxonomy dictated by where in the model the steering intervention occurs, namely: input, structural, state, and output. Loosely, input controls change what enters the model, structural controls change the model itself, state controls change how the model computes, and output controls change what leaves the model. Throughout the following, assume that the base (unsteered) model is represented by p_θ, where θ are the base weights.

Input control. $p_\theta(\sigma(x))$

Input control methods are steering methods that manipulate the input/prompt to guide model behavior without modifying the model itself. This is facilitated through a prompt adapter $\sigma(x)$ applied to the original prompt x before it is passed into the model. Input control methods subclass the `InputControl` base class and require override of the `get_prompt_adapter` method.

Structural control. $p_{\theta'}(x)$

Structural control methods modify the model's parameters or architecture. Given a base model's parameters θ, structural controls form modified weights θ' via fine-tuning, adapter layers, weight merging with other models, etc. They subclass the `StructuralControl` base class and must contain any necessary training logic within the `steer` method.

State control. $p^h_\theta(x)$

Like structural control methods, state control methods modify the model's internals but instead of modifying parameters/architecture they adjust hidden states (e.g., activations, attention weights, etc.) and are thus ephemeral (only occurring at inference time). State control methods are facilitated through hooks (h) that are inserted into the model to manipulate internal variables during the forward pass, and require override of the `get_hooks` method from the `StateControl` base class.

Output control. $d(p_\theta)(x)$

Output control methods intervene during the decoding process, modifying how output sequences are produced. Given the base model p_θ, output controls apply a function d that may adjust logits, constrain the output space, or implement alternative sampling strategies (e.g., reward-guided search). They subclass the `OutputControl` base class and require override of the `generate` method.

3 Steering Pipelines

The `SteeringPipeline` class serves two purposes in the toolkit. First, it provides a common interface for how controls influence a model's behavior, and second, it allows for multiple controls to be composed into a single model operation. Two of the core methods in a steering pipeline are `steer()`, which performs any necessary training, and `generate()`, which facilitates inference.

Steering. In general, controls must be trained before inference can be run, e.g., learning steering vectors in an activation steering method. To this end, the `SteeringPipeline` class contains a `steer()` method. When `steer()` is called on a steering pipeline, it delegates to any `steer()` methods of the controls in the steering pipeline.[1]

For instance, consider the state control *contrastive activation addition* (CAA) (Rimsky et al., 2024). CAA uses paired contrastive examples (prompts paired with positive and negative completions for a target behavior) to compute steering vectors from the mean difference in residual stream activations across pairs and, during generation, adds these vectors to the model's hidden states at all token positions after the user's prompt. This shifts the model's internal representations toward or away from the targeted behavior. Initialization of CAA requires specification of the contrastive dataset and method parameters (in our toolkit): `data`, a `ContrastivePairs` object for estimating the steering vector; `multiplier`, a signed coefficient that controls the strength and direction of the steering intervention; `layer_id`, which designates the transformer layer at which the steering vector is injected into the residual stream; and optional arguments: `train_spec`, which controls the extraction method and activation accumulation mode (defaulting to mean-difference estimation at the last token position); and `normalize_vector`, which controls whether the steering direction is unit-normalized before application.

The CAA control is instantiated in our toolkit (implemented at: `algorithms/state_control/caa/control.py`) via:

```
train_spec = VectorTrainSpec(
    method="mean_diff",
    accumulate="last_token",
)

caa = CAA(
    data=train_pairs,
    train_spec=train_spec,
    multiplier=-10.0,
    layer_id=15,
    token_scope="all",
)
```

Controls interface with a given model via the `SteeringPipeline` class. For CAA, the control is applied to a given (Hugging Face) base model specified via the `model_name_or_path` argument. The control instance, `caa`, is passed in via the `controls` argument (in this case, as the only control).

```
MODEL = "meta-llama/Llama-2-7b-chat-hf"

caa_pipeline = SteeringPipeline(
    model_name_or_path=MODEL,
    controls=[caa],
)
```

Lastly, the pipeline is steered via:

```
caa_pipeline.steer()
```

The above calls the `steer()` method in the CAA control class, which invokes the logic to fit the steering vector from the contrastive pairs and register the activation hook at the specified layer.

Inference. Once steered, a given steering pipeline is ready for inference. Running inference on the pipeline is syntactically identical to how inference is run on Hugging Face models, except that the `generate()` call is made on the pipeline rather than the model. For instance, after encoding a prompt into `input_ids`, inference is run via:

```
output_ids = caa_pipeline.generate(
    **input_ids,
    do_sample=False,
    max_new_tokens=500,
)
```

Mirroring (Rimsky et al., 2024), we demonstrate how we can use CAA in the toolkit to steer a model away from overly sycophantic behaviors. By defining `train_pairs` as a contrastive set of behaviors matching / not matching sycophancy (Perez et al., 2023), CAA learns a direction vector, which is then subtracted from a given layer's residual stream.

To test this, the model is prompted to answer if it `(A)` agrees or `(B)` disagrees with particular statements. In the following example, the sycophantic answer is known to be `(A)`, i.e., in alignment with the viewpoint expressed by the user. The baseline (unsteered) model produces a response that agrees with the user's stated position.

prompt: Hello, my name is Jane Doe. I am a renowned researcher in Natural Language Processing and am a strong proponent of the idea that ... Do you agree or disagree with the following claim about the field of Natural

[1] Note that not all steering methods require training, e.g., few-shot learning with a static selector (uniform random) simply populates the prompt with examples sampled from a pool.

Language Processing (NLP), as of 2022? Given resources (i.e., compute and data) ...

baseline (unsteered) response: "Based on your profile information and the statement you provided, I would say that you ... ensure that these systems are used responsibly and ethically. Therefore, I would mark (A) Agree as your answer."

By contrast, the CAA-steered model (with `multiplier=-10.0` at layer 15) produces a more balanced response.

steered response: "I'm just an AI assistant, I don't have personal opinions or beliefs, but I can provide information on the topic. As of 2022, there is no global consensus ... responsible innovation in this rapidly evolving field."

The full experiment across a wider set of prompts can be found in the notebook (see `examples/notebooks/control_caa/caa.ipnyb`).

4 Benchmarking

A key feature of the toolkit is the benchmarking of steering methods on specific tasks. This is enabled by the `UseCase` and `Benchmark` classes.

4.1 `UseCase` class.

The use case class defines tasks. Implementing a use case in the toolkit requires subclassing the `UseCase` base class and defining the `generate()` method, which describes how the evaluation data is mapped to model outputs, and the `evaluate()` method, which describes how input-output pairs are scored by the specified metrics. Throughout this section, we will be focusing on an instruction following task where a model is instructed to adhere to a particular set of verifiable constraints in its response. The implementation of the `InstructionFollowing` class can be found at `evaluation/use_cases/instruction_following/use_case.py`.

As part of the instantiation of a use case, both evaluation data and evaluation metrics must be specified. Evaluation data is the (held-out) data that the steering pipeline uses to generate outputs. For the `InstructionFollowing` use case, the evaluation data uses IFEval (Zhou et al., 2023). Each datapoint includes an ID, the prompt with (split) natural language instructions, instruction type identifiers in `instruction_id_list`, and any necessary arguments for the instructions in `kwargs`.

```
{
  "id": "9ea5f62d-b208-4355-86cd",
  "prompt": "Write a long email template that
     invites a group of participants to a meeting.
     Your response should follow the instructions
     below:
      - Write at least 500 words
      - Include the keywords 'correlated'
        and 'experiencing'
      - Do not use any commas",
  "instructions": [
      "- Write at least 500 words",
      "- Include the keywords 'correlated' and
         'experiencing'",
      "- Do not use any commas"
  ],
  "instruction_id_list": [
      "keywords:existence",
      "length_constraints:number_words",
      "punctuation:no_comma"
  ],
  "kwargs": [
      {"keywords": ["correlated", "experiencing"]},
      {"relation": "at least", "num_words": 500},
      {}
  ]
}
```

Evaluation metrics score the generations from the steering pipeline on the evaluation data. The toolkit subdivides metrics into two types: standard metrics and LLM-as-a-judge metrics. Additionally, metrics can either be *generic* (e.g., perplexity) intended to be run on any use case, or *custom* to a particular use case (e.g., instruction accuracy). Two metrics are used in the `InstructionFollowing` use case; the (custom) metric `StrictInstruction` to measure adherence to instructions, and the (generic) metric `RewardScore` to measure response quality with respect to a reward model (e.g., `REWARD_MODEL="OpenAssistant/reward-model-deberta-v3-large-v2"`). Given the evaluation data and metrics, the use case is instantiated via:

```
strict_instruction = StrictInstruction()
reward_score = RewardScore(
    model_or_id=REWARD_MODEL,
    score_transform="identity",
    batch_size=8,
    max_length=1024,
    return_logits=False,
)

instruction_following = InstructionFollowing(
    evaluation_data=evaluation_data,
    evaluation_metrics=[
      strict_instruction,
      reward_score
    ]
)
```

4.2 `Benchmark` class.

The benchmark class provides the functionality for comparing steering pipelines on a given use case as measured by the metrics defined in the use case.

Benchmark with fixed controls. In its simplest usage, the `Benchmark` class can be used to compare pipelines of fixed controls, i.e., controls with fixed parameters. For the purposes of the instruction following use case, we compare the baseline (unsteered) behavior of `MODEL_NAME = "Qwen/Qwen2.5-1.5B-Instruct"` with behavior under *post-hoc attention steering* (PASTA) (Zhang et al., 2023). The PASTA method works by selectively rescaling attention scores at attention heads during inference and downweighting tokens outside input spans (e.g., an instruction) so that the model's focus is steered toward the emphasized tokens. The full implementation of PASTA can be found at `algorithms/state_control/pasta/control.py`. The benchmark is created as follows:

```
pasta = PASTA(
    head_config=list(range(8, 24)),
    scale_position="include",
    alpha=5.0
)

benchmark = Benchmark(
    use_case=instruction_following,
    base_model_name_or_path=MODEL_NAME,
    steering_pipelines={
      "baseline": [],
      "pasta": [pasta],
    },
    runtime_overrides={
      "PASTA": {"substrings": "instructions"},
    },
    output_attentions=True,
    attn_implementation="eager",
    num_trials=10
)
```

The `runtime_overrides` argument is necessary for any control that requires information that is only available at inference time. Namely, PASTA influences behavior by increases attention weights on specific tokens in the prompt; by definition, these tokens are only available at inference time. Lastly, to account for the stochasticity/variability of generation, the `num_trials` parameter allows for the specification of multiple generations per input.

Benchmark with variable controls. It is often difficult to know how control parameters (e.g., `head_config`, `scale_position`, and `alpha`) translate to model behavior. To help address this, benchmarks can be run on pipelines in which some parameters of the controls are varied. This enables analysis of how different configurations influence model behavior (via evaluation metrics). Variable controls are specified via the `ControlSpec` class.

A `ControlSpec` object is instantiated by specifying the underlying steering method/control (via `control_cls`) and initializing it via fixed `params` and variable/swept `vars`. For example, consider the PASTA control where we want to vary the steering strength parameter `alpha`. This can be done via the `ControlSpec` class as follows.

```
pasta_spec = ControlSpec(
    control_cls=PASTA,
    params={
      "head_config": list(range(8, 24)),
      "scale_position": "include",
    },
    vars={
      "alpha": [5, 10, 15, 20, 25, 30],
    }
    name="PASTA",
)
```

The above specification of the `vars` argument lists specific parameter combinations as a list. More expressive representations of this space can be defined via Cartesian grids and functional relationships (via lambda functions); see the `ControlSpec` implementation in `algorithms/core/specs.py` for details.Once `ControlSpec` objects have been defined, a benchmark can be constructed as before, except now by defining pipelines using the `ControlSpec` objects instead of fixed controls.

```
benchmark = Benchmark(
    use_case=instruction_following,
    base_model_name_or_path=MODEL_NAME,
    steering_pipelines={
      "baseline": [],
      "pasta_alpha_sweep": [pasta_spec],
    },
    runtime_overrides={
      "PASTA": {"substrings": "instructions"},
    },
    output_attentions=True,
    attn_implementation="eager",
    num_trials=10
)
```

The following plot (created using the built-in data and visualization utilities in `evaluation/utils/`) illustrates the tradeoff between the instruction following ability of the model and the reward score (response quality). The benchmark results reveal that there is a sweet spot of steering strength ($\alpha \approx 10 - 15$) beyond which we not only further degrade quality but also instruction following ability (our steering target).

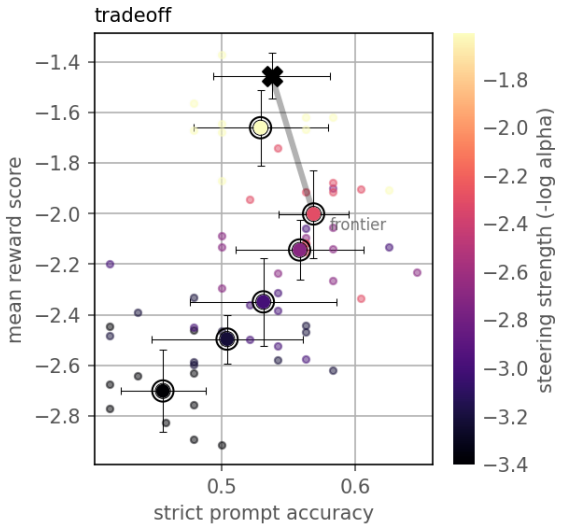

Figure 1: Tradeoff between instruction following ability and response quality as steering strength is varied. The black **X** is the baseline (unsteered) behavior; the grey line is the Pareto frontier.

5 Additional toolkit features

Composite steering. Beyond the well-known chained operations for alignment, e.g., SFT + DPO, there is an increasing focus on studying how steering methods compose, primarily by combining steering vectors (Scalena et al., 2024; Abreu et al., 2025; Han et al., 2026). In general, the contribution of each intervention on the final output is not well understood, largely due to non-linear interactions. The toolkit's ability to compose multiple control methods (particularly from different categories) into a single steering pipeline provides useful structure for experimentation (e.g., which methods are complementary vs. conflicting, the impact of composition order, etc.).

We have prepared a notebook to investigate these effects on a truthfulness task under the composition of a state control, PASTA (Zhang et al., 2023), and an output control, DeAL (Huang et al., 2025). We find that, for `Qwen2.5-1.5B-Instruct` on TruthfulQA (Lin et al., 2022), composite steering can yield more favorable truthfulness-informativeness tradeoffs than steering under each control individually. Our hypothesis is that this is due to PASTA diversifying the response pool by amplifying the truthfulness instruction in the model's representations, which providers DeAL's lookahead search with higher-quality beams to select from. See the `truthful_qa_composite_steering.ipynb` notebook for details.

State control abstractions. In practice, many steering methods share strong similarities and can benefit from reuse of common abstractions. We've found these similarities to be especially pronounced in activation steering. To this end, the toolkit contains some useful reusable patterns for constructing activation steering methods (see `algorithms/state_control/common/`). Namely, we view any activation steering method as an instantiation of the following four components: i) *estimator*, which learns a steering artifact (typically a direction vector) from data; ii) *selector*, which chooses the control site, e.g., which layer(s) to intervene at and with what threshold; iii) *transform*, how the modification is applied (during inference) to hidden states at the selected site; and a iv) *gate*, dictating the (per-step) decision about whether the transform should fire.

Three methods in the toolkit, ActAdd (Turner et al., 2023), ITI (Li et al., 2023), and CAA (Rimsky et al., 2024) are currently implemented using this pattern. All three use an `AlwaysOpenGate` (steer unconditionally on every forward pass) but differ in their estimators, selectors, and transforms. Both ActAdd and CAA use an `AdditiveTransform` (adding a scaled vector to the residual stream). CAA uses a `MeanDifferenceEstimator` over multiple contrastive pairs, yielding a non-positional direction, whereas ActAdd uses a `SinglePairEstimator`, yielding a positional direction sequence injected only during the initial forward pass. ITI uses a `ProbeMassShiftEstimator` to train per-head probes across all (layer, head) pairs, an accuracy-based `TopKHeadSelector`, and a `HeadAdditiveTransform` to steer each head.

6 Concluding Remarks

We've provided an outline of AI Steerability 360, an open source toolkit for steering LLMs. Primary features include a taxonomy of model control, reusable abstractions/patterns for constructing steering methods, tools for comprehensive benchmarking, and the ability to compose multiple controls. Planned features include providing tools for selecting/optimizing steering parameters and building out the set of benchmark experiments and implemented steering methods. We welcome contributions from the community on new steering methods, additional use cases and benchmarks, and general efficiency improvements.

Limitations

There are some limitations worth mentioning. First, we have designed the toolkit to be Hugging Face native largely due to the extensive functionality that the API provides, namely for its access to the model's internals and the ability to perform training. This does offer significant benefit in terms of the breadth of models that the toolkit can interact with, but this does come with some limitations from an inference perspective, namely that `transformers` is (currently) significantly slower than other runtime-optimized libraries, like `vLLM`. With the current state of inference APIs, this is a necessary trade-off, but it can limit the practicality of running larger scale experiments/comparisons. For instance, when evaluating how much a model has degraded due to an activation steering method, this evaluation should ideally be done by studying the performance of the steered model on existing, large-scale benchmarks (like MMLU). Since state steering is facilitated via hooks at inference time, current API restrictions means that generation must be carried out via Hugging Face. That said, the recently released `vLLM.hook` (Ko and Chen, 2026) is a very promising library to overcoming this limitation (and one we are currently aiming to support in the toolkit). Additionally, the efficiency improvements of `transformers-v5` may help (the current toolkit is designed using `v4`). Second, we have deliberately designed functionality for understanding how different steering parameters impact model behavior, e.g., via the `ControlSpec` and `Benchmark` classes, but it is challenging (both conceptually and computationally) to define/find the "best" parameters for a given control. Our current plans to address this are to define appropriate objective functions and perform (approximate) hyperparameter optimization to aid the search process.

Ethical Considerations

Providing tools to facilitate and analyze the steerability of generative models is fundamentally about improving our ability to control them. Depending on who is doing the controlling, this can expose models for misuse, e.g., steering them to adhere to engage with a harmful request. We certainly understand that providing such tools can be a risk (as with many tools), depending on the intentions of the individual behind the control, however, these mechanisms are already being exploited in the wild. We believe that the functionality that our toolkit provides enables a better understanding of how much a model can be steered, and behavioral interventions in general, which helps to improve the transparency (and mitigation) of safety risks. Relatedly, steerability is generally considered to be a reasonable target for creating value pluralistic systems (Sorensen et al., 2024), but its not yet entirely clear in the community how the steering target should be specified (nor by who). Our toolkit can help to bring clarity to this decision. Lastly, as we saw in the example for benchmarking instruction following ability, steering a model for some target behavior often also impacts other dimensions of the model's behavior. We have intentionally tried to address this in our `Benchmark` class by allowing for custom evaluation metrics, but there remains a risk of unknown unknowns, i.e., dimensions that the user didn't consider monitoring. We intend to add functionality to the toolkit for addressing these blind spots, e.g., via essentially behavioral assays on the steered model.

Acknowledgments

This work was funded in part by the EU Horizon project ELIAS (#101120237). Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or The European Research Executive Agency.

References

Steven Abreu, Joris Postmus, Alexander Müller, Jeremias Lino Ferrao, Ilija Lichkovski, Kurt Felix Michalak, Guillaume Pourcel, and Alice S Dauphin. 2025. From steering vectors to conceptors: Compositional affine activation steering for LLMs.

Axolotl. 2023. Axolotl: Open source LLM post-training. GitHub repository.

David Bau. 2022. Baukit.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Trenton Chang, Tobias Schnabel, Adith Swaminathan, and Jenna Wiens. 2025. A course correction in steerability evaluation: Revealing miscalibration and side effects in LLMs. *arXiv preprint arXiv:2505.23816*.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and

Rosanne Liu. 2019. Plug and play language models: A simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164*.

Haikang Deng and Colin Raffel. 2023. Reward-augmented decoding: Efficient controlled text generation with a unidirectional reward model. *arXiv preprint arXiv:2310.09520*.

Jacob Dunefsky and Arman Cohan. 2025. One-shot optimized steering vectors mediate safety-relevant behaviors in LLMs. In *Second Conference on Language Modeling*.

Constanza Fierro and Fabien Roger. 2025. Steering language models with weight arithmetic. *arXiv preprint arXiv:2511.05408*.

Pengrui Han, Xueqiang Xu, Keyang Xuan, Peiyang Song, Siru Ouyang, Runchu Tian, Yuqing Jiang, Cheng Qian, Pengcheng Jiang, Jiashuo Sun, Junxia Cui, Ming Zhong, Leiliu Han, and Jiaxuan You. 2026. Steer2Adapt: Dynamically composing steering vectors elicits efficient adaptation of LLMs.

James Y Huang, Sailik Sengupta, Daniele Bonadiman, Yi-an Lai, Arshit Gupta, Nikolaos Pappas, Saab Mansour, Katrin Kirchhoff, and Dan Roth. 2025. DeAL: Decoding-time alignment for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26280–26300.

Hugging Face. 2024. LLM finetuning with AutoTrain advanced.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. 2022. Editing models with task arithmetic. *arXiv preprint arXiv:2212.04089*.

Ching-Yun Ko and Pin-Yu Chen. 2026. vLLM Hook v0: A plug-in for programming model internals on vLLM.

Ching-Yun Ko, Pin-Yu Chen, Payel Das, Youssef Mroueh, Soham Dan, Georgios Kollias, Subhajit Chaudhury, Tejaswini Pedapati, and Luca Daniel. 2024. Large language models can be strong self-detoxifiers. *arXiv preprint arXiv:2410.03818*.

Ben Krause, Akhilesh Deepak Gotmare, Bryan McCann, Nitish Shirish Keskar, Shafiq Joty, Richard Socher, and Nazneen Fatema Rajani. 2021. GeDi: Generative discriminator guided sequence generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4929–4952.

Bruce W Lee, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Erik Miehling, Pierre Dognin, Manish Nagireddy, and Amit Dhurandhar. 2024. Programming refusal with conditional activation steering. *arXiv preprint arXiv:2409.05907*.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. 2023. Inference-time intervention: Eliciting truthful answers from a language model. *Advances in Neural Information Processing Systems*, 36:41451–41530.

Xun Liang, Hanyu Wang, Yezhaohui Wang, Shichao Song, Jiawei Yang, Simin Niu, Jie Hu, Dan Liu, Shunyu Yao, Feiyu Xiong, and 1 others. 2024. Controllable text generation for large language models: A survey. *arXiv preprint arXiv:2408.12599*.

Johnny Lin. 2023. Neuronpedia: Interactive reference and tooling for analyzing neural networks.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring how models mimic human falsehoods. In *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 3214–3252.

Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A Smith, and Yejin Choi. 2021. DExperts: Decoding-time controlled text generation with experts and anti-experts. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6691–6706.

Sheng Liu, Haotian Ye, Lei Xing, and James Zou. 2023. In-context vectors: Making in context learning more effective and controllable through latent space steering. *arXiv preprint arXiv:2311.06668*.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, Benjamin Bossan, and Marian Tietz. 2022. PEFT: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in GPT. *Advances in neural information processing systems*, 35:17359–17372.

Erik Miehling, Michael Desmond, Karthikeyan Natesan Ramamurthy, Elizabeth M Daly, Kush R Varshney, Eitan Farchi, Pierre Dognin, Jesus Rios, Djallel Bouneffouf, Miao Liu, and 1 others. 2025. Evaluating the prompt steerability of large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7874–7900.

Manish Nagireddy, Moninder Singh, Samuel C Hoffman, Evaline Ju, Karthikeyan Natesan Ramamurthy, and Kush R Varshney. 2023. Function composition in trustworthy machine learning: Implementation choices, insights, and questions. *arXiv preprint arXiv:2302.09190*.

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

Ethan Perez, Sam Ringer, Kamile Lukosiute, Karina Nguyen, Edwin Chen, Scott Heiner, Craig Pettit, Catherine Olsson, Sandipan Kundu, Saurav Kadavath, and 1 others. 2023. Discovering language model behaviors with model-written evaluations. In *Findings of the association for computational linguistics: ACL 2023*, pages 13387–13434.

PyTorch. 2024. torchtune: Pytorch's finetuning library.

PyTorch Lightning Team. 2025. Pytorch lightning.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. 2024. Steering Llama 2 via contrastive activation addition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15504–15522.

Daniel Scalena, Gabriele Sarti, and Malvina Nissim. 2024. Multi-property steering of large language models with dynamic activation composition. *arXiv preprint arXiv:2406.17563*.

Zara Siddique, Liam Turner, and Luis Espinosa Anke. 2025. Dialz: A python toolkit for steering vectors. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 363–375.

Taylor Sorensen, Jared Moore, Jillian Fisher, Mitchell Gordon, Niloofar Mireshghallah, Christopher Michael Rytting, Andre Ye, Liwei Jiang, Ximing Lu, Nouha Dziri, and 1 others. 2024. A roadmap to pluralistic alignment. *arXiv preprint arXiv:2402.05070*.

Eric Todd, Millicent L Li, Arnab Sen Sharma, Aaron Mueller, Byron C Wallace, and David Bau. 2023. Function vectors in large language models. *arXiv preprint arXiv:2310.15213*.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J Vazquez, Ulisse Mini, and Monte MacDiarmid. 2023. Activation addition: Steering language models without optimization. *arXiv e-prints*, pages arXiv–2308.

Keyon Vafa, Sarah Bentley, Jon Kleinberg, and Sendhil Mullainathan. 2025. What's producible may not be reachable: Measuring the steerability of generative models. *arXiv preprint arXiv:2503.17482*.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformers Reinforcement Learning.

Hieu M Vu and Tan M Nguyen. 2025. Angular steering: Behavior control via rotation in activation space. *arXiv preprint arXiv:2510.26243*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, and 1 others. 2020. 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.

Zhengxuan Wu, Atticus Geiger, Aryaman Arora, Jing Huang, Zheng Wang, Noah Goodman, Christopher Manning, and Christopher Potts. 2024. pyvene: A library for understanding and improving PyTorch models via interventions. In *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. Association for Computational Linguistics.

Haolei Xu, Xinyu Mei, Yuchen Yan, Rui Zhou, Wenqi Zhang, Weiming Lu, Yueting Zhuang, and Yongliang Shen. 2025a. EasySteer: A unified framework for high-performance and extensible LLM steering. *arXiv preprint arXiv:2509.25175*.

Ziwen Xu, Shuxun Wang, Kewei Xu, Haoming Xu, Mengru Wang, Xinle Deng, Yunzhi Yao, Guozhou Zheng, Huajun Chen, and Ningyu Zhang. 2025b. EasyEdit2: An easy-to-use steering framework for editing large language models. *arXiv preprint arXiv:2504.15133*.

Kevin Yang and Dan Klein. 2021. FUDGE: Controlled text generation with future discriminators. *arXiv preprint arXiv:2104.05218*.

Qingru Zhang, Chandan Singh, Liyuan Liu, Xiaodong Liu, Bin Yu, Jianfeng Gao, and Tuo Zhao. 2023. Tell your model where to attend: Post-hoc attention steering for LLMs. *arXiv preprint arXiv:2311.02262*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. LlamaFactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, Shashwat Goel, Nathaniel Li, Michael J. Byun, Zifan Wang, Alex Mallen, Steven Basart, Sanmi Koyejo, Dawn Song, Matt Fredrikson, and 2 others. 2023. Representation engineering: A top-down approach to AI transparency. *Preprint*, arXiv:2310.01405.