Robust LLM Unlearning with MUDMAN: Meta-Unlearning with Disruption Masking And Normalization

Anonymous ACL submission

Abstract

Language models can retain dangerous knowledge and skills even after extensive safety fine-tuning, posing both misuse and misalignment risks. Recent studies show that even specialized unlearning methods can be easily reversed. To address this, we systematically evaluate many existing and novel components of unlearning methods and identify ones crucial for irreversible unlearning.

We introduce Disruption Masking, a technique in which we only allow updating weights, where the signs of the unlearning gradient and the retaining gradient are the same. This ensures all updates are non-disruptive.

Additionally, we identify the need for normalizing the unlearning gradients, and also confirm the usefulness of meta-learning. We combine these insights into MUDMAN (Meta-Unlearning with Disruption Masking and Normalization) and validate its effectiveness at preventing the recovery of dangerous capabilities. Our results show that MUDMAN significantly outperforms the prior TAR method, setting a new state-of-the-art for robust unlearning.

Code: anonymous.4open.science/r/MUDMAN

1 Introduction

Language models can acquire dangerous skills during pre-training, such as manipulation, hacking and even knowledge useful for creating bioweapons (Li et al., 2024). They may also learn about the safeguards used to control them, which in the future may enable them to subvert such safeguards (Greenblatt et al., 2024; Roger, 2024).

Studies show that popular safety fine-tuning techniques like DPO and RLHF fail to remove dangerous knowledge; instead, they minimally modify model weights to hide it (Lee et al., 2024), allowing it to reemerge through jailbreak inputs (Zou et al., 2023), or even accidentally (Qi et al., 2023; Deeb and Roger, 2024). Even specialized unlearning techniques turn out to be easily reversible (Lynch et al., 2024; Łucki et al., 2025; Deeb and Roger, 2024).

To address these challenges, we systematically investigated which components of unlearning algorithms make behavior removal truly irreversible, testing both existing and newly designed ones. We identified several key components for more robust unlearning: Disruption Masking, meta-learning, and gradient normalization. Additionally, we constrain unlearning to specific model modules. We integrate these into MUDMAN (Meta-Unlearning with Disruption Masking and Normalization) and demonstrate that it significantly outperforms state-of-the-art across multiple models and tasks.

2 Related Work

Current unlearning methods fail to robustly unlearn knowledge Some unlearning approaches disrupt intermediate activations within the model (Zou et al., 2024; Li et al., 2024; Rosati et al., 2024), while others attempt to locate and ablate weights responsible for unwanted behavior (Wang et al., 2024; Wu et al., 2023; Uppaal et al., 2024; Suau et al., 2024). However, Lo et al. (2024) found that even when unwanted concepts are directly removed, the model can quickly learn to represent them again using neurons with similar meaning.

To address this, modern unlearning techniques increasingly incorporate meta-learning (Tamirisa et al., 2024a; Henderson et al., 2023; Tamirisa et al., 2024b). This approach anticipates how an attacker could relearn the target capability, by deriving unlearning gradients from a copy of the model trained on the forget set (Finn et al., 2017). This ensures that harmful behavior continues to be unlearned even after it becomes dormant in the main model.

However, for each existing unlearning method, there are ways to elicit the supposedly removed capabilities, for example with jailbreaks, few-shot prompting, fine-tuning, in-context learning, out-of-distribution inputs or disabling refusal mechanisms with representation engineering (Lynch et al., 2024; Łucki et al., 2025).

3 Methodology

Models We conduct experiments with three language models of different sizes: **pythia-14m** (Biderman et al., 2023)[1], **SmolLM-135M** (Allal et al., 2024) and **Llama-3.2-1B** (Grattafiori et al., 2024).

Datasets We demonstrate skill unlearning by attempting to unlearn **Python** coding ability by using function examples from CodeSearchNet (Husain et al., 2019), while retaining the performance on Wikitext (Merity et al., 2016). To ensure we do not unintentionally unlearn English, we remove comments and docstrings from the Python dataset. While programming is not a harmful behavior, using loss on the Python dataset provides a high signal-to-noise ratio for comparing unlearning methods and serves as a good baseline.

Then we move to *knowledge* unlearning with a realistic target: unlearning biohazardous knowledge using **Pile-Bio** (Tamirisa et al., 2024a), a subset of the Pile (Gao et al., 2020) containing texts about molecular biology. The rest of the Pile serves as the retain set.

To test unlearning effectiveness on Pile-Bio, we measure the accuracy on **WMDP-Bio** (Li et al., 2024), a dataset of 1273 multiple-choice questions designed as a proxy measurement of hazardous biosecurity knowledge.

Hyperparameter Search Method performance is highly dependent on hyperparameter selection. To ensure a fair comparison, we perform an automated hyperparameter search for each method using Optuna (Akiba et al., 2019). We verify that each search converges and that the chosen hyperparameter ranges are not saturated. Each search includes hundreds of trials, with each trial consisting of an unlearning stage followed by a fixed relearning stage, and the amount of compute in each stage is constant. (See Appendix F for details.)

Unlearning and Retaining Metrics Optuna tries to maximize forget set loss (or WMDP accuracy) *after relearning* with supervised fine-tuning. While we are interested in the overall recoverability of unlearned behavior (e.g., through jailbreaks or out-of-distribution attacks), here we just use supervised fine-tuning, as it is the simplest and most reliable way to resurface removed behavior (Lynch et al., 2024).

In addition to maximizing forget set loss, we ensure that unlearning does not significantly degrade performance on the retain set. To control for this, we terminate and reject trials where the retain loss exceeds a fixed threshold[2].

[1] Using pythia-14m allowed us to iterate fast on our techniques. Then we validate our findings on the larger, more modern models.

[2] Set as the initial retain loss + 0.05. We add 0.05 to accommodate for random loss fluctuations.

4 Building a Robust Unlearning Method

We conducted hundreds of small-scale experiments, testing methods ranging from direct model edits to blocking the updates during relearning. See Appendix D for details of these approaches. While almost all methods succeeded in making the forget set loss high during unlearning, relearning typically restored it immediately.

However, we identified several key components that consistently improve unlearning robustness. We integrate these into MUDMAN (Meta-Unlearning with Disruption Masking and Normalization), which outperforms the state-of-the-art TAR method (Tamirisa et al., 2024a). See Algorithm 1 for pseudocode and Appendix E for a minimal PyTorch implementation. Below, we describe each component in detail before presenting MUDMAN's overall results.

4.1 Meta-Learning

First, we confirm the effectiveness of meta-learning, which is increasingly used in modern unlearning methods (Tamirisa et al., 2024a; Henderson et al., 2023; Tamirisa et al., 2024b; Finn et al., 2017). Meta-learning involves training a copy of the main model—which we call *the adversary*—on the forget set and applying its gradients to the main model.

Traditional meta-learning employs an inner loop to train multiple adversaries and accumulate unlearning gradients before updating the main model (Finn et al., 2017). However, in early experiments, we found that interleaving adversary and main model updates improves performance. To achieve this, we flatten the process into a single loop (see Algorithm 1) and focus on training a single adversary deeply, rather than multiple ones briefly.

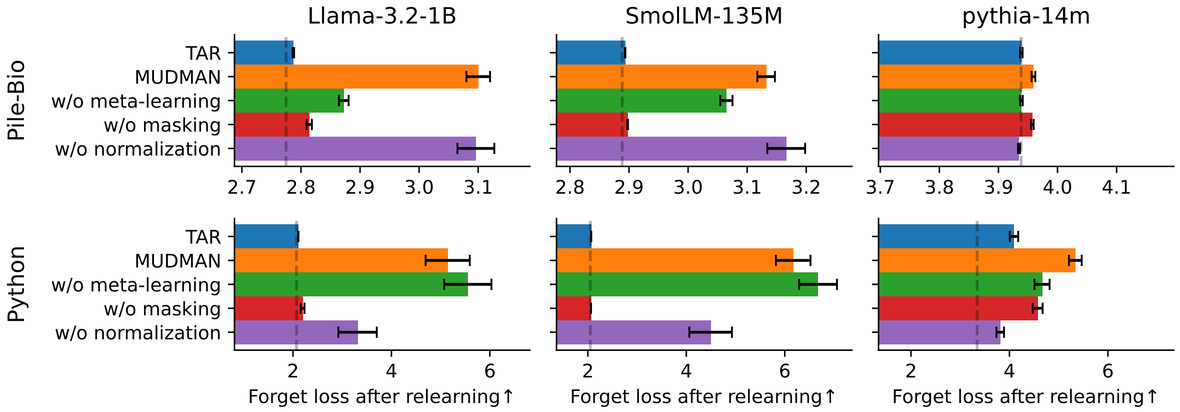

Figure 1: **Ablation study of MUDMAN.** To establish that each part of MUDMAN is indeed necessary, we disable them one by one and measure unlearning performance. We also compare to the state-of-the-art TAR method. The baseline is the loss level with no unlearning applied, but after the same relearning as the other methods underwent. Each bar corresponds to one Optuna hyperparameter search. The reported loss is the average of the last 30 valid trials and error bars are their standard error. We can see that Disruption Masking makes a huge difference (orange vs red), and that it accounts for most of the improvement over TAR. Meta-learning and unlearning gradient normalization also tend to help but not in every setup. Sometimes ablating them yields better performance, but insignificantly. In each experiment we only trained the first layers of each MLP.

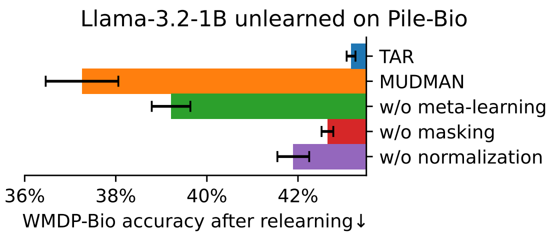

Figure 2: **Accuracy on WMDP-Bio after unlearning and relearning on Pile-Bio.** The base level on the rigth (43.5%) is the accuracy with no unlearning applied, but also after relearning. Reported accuracy is the average of the last 20 valid trials in each search. Just like in Figure 1, using Disruption Masking is crucial (orange vs red) and it accounts for most of the improvement over TAR. We also cleary see the need for meta-learning and unlearning gradient normalization. In each experiment we only trained the first layers of each MLP.

4.2 Disruption Masking

This is the key contribution of MUDMAN, and we find it to be even more crucial than meta-learning itself. In existing unlearning techniques (Tamirisa et al., 2024a; Rosati et al., 2024; Li et al., 2024), during unlearning the model is also trained on the retain set, hoping to revert any unintended disruptions. Instead, we take a more selective approach, aiming to avoid disruptions altogether rather than correcting them later.

Our intuition is that since the model has already undergone extensive pre-training, its weights are near-optimal. Any unnecessary modifications risk disrupting well-learned representations, requiring significant compute to recover what pre-training had already established.

To prevent disruption, we found it is best to apply unlearning gradients **only when they have the same sign as their respective retain gradients** (see Equation 1). This not only prevents disruption but also passively improves retain performance.

$$g_{final} = \begin{cases} g_u & \text{if } \mathrm{sign}(g_u) = \mathrm{sign}(g_r) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$
where g_u is the unlearning gradient and g_r is the retaining gradient

To generalize beyond the current batch and identify what might break retain set performance overall, we use SGD momentum (Rumelhart et al., 1986), which adds retain gradients to a decaying accumulator. Then, instead of the signs of the single-batch retain gradients, we use the signs of this accumulator. This incurs no additional memory cost, as we already need to store retain gradients alongside unlearning gradients, so we simply store the accumulator instead.

4.3 Gradient Normalization

Late in the unlearning process, we observe that gradient norms shrink significantly, causing unlearning to slow down. To address

Algorithm 1 MUDMAN – Meta-Unlearning with Disruption Masking And Normalization

Input: Model weights $model$; retain set $\mathcal{D}_{retain}$; forget set $\mathcal{D}_{forget}$; retain momentum μ; unlearning loss $\mathcal{L}_{unlearning}$; learning rates $\alpha_{retaining}$, $\alpha_{unlearning}$ and α_{adv}.

$retain\_acc = 0$ — Initialize retain accumulator
for $loop\_num = 1$ **to** $num\_iterations$ **do**
  if $loop\_num$, is divisible by $fork\_every\_n\_loops$ **then**
    $adv = model$ — Fork out the adversarial model from the main model
  end if
  $x_{retain} \sim \mathcal{D}_{retain}, x_{forget} \sim \mathcal{D}_{forget}$ — Sample batches
  $retain\_grad = \nabla_{model}\mathcal{L}_{LM}(model, x_{retain})$ — Calculate retain gradients
  $retain\_acc = \mu \cdot retain\_acc + (1 - \mu) \cdot retain\_grad$ — Update retain accumulator
  $model \mathrel{-}= \alpha_{retaining} retain\_acc$ — Update the model

  $adv \mathrel{-}= \alpha_{adv} \nabla_{adv}\mathcal{L}_{LM}(adv, x_{forget})$ — Train adversary on forget set
  $grad = \nabla_{adv}\mathcal{L}_{unlearning}(adv, x_{forget})$ — Calculate unlearning gradient
  $grad \mathrel{/}= \|grad\|_2$ — Normalize it
  $grad \mathrel{*}= (\text{sign}(grad) == \text{sign}(retain\_acc))$ — Mask out gradients which hurt retain loss
  $model \mathrel{-}= \alpha_{unlearning} grad$ — Update the model with the unlearning gradient
end for

this, we normalize the unlearning gradients so that the strength of unlearning stays constant. Concretely, we divide each unlearning gradient by a global gradient norm calculated as: $(\sum_{m \in \text{modules}} \|unlearning\_grad_m\|^2)^{0.5}$. This not only improves overall performance but also makes the process easier to tune. See Appendix B for a comparison of various normalization variants.

4.4 Module Selection

We found that it helps to also be selective about which modules of the model we intervene on. For example MLP's second weight matrices and attention's Q and K matrices tend to disrupt general performance a lot, while not being that important for unlearning. The most effective unlearning targets turn out to be the first layers of MLPs (and in the case of gated MLPs, the gating matrices). This is likely because only these modules are able to deactivate MLP neurons, and once a neuron is inactive, neither its incoming nor outgoing weights can be updated by backpropagation, effectively preventing relearning. See Appendix A for detailed comparisons of unlearning on various modules.

4.5 Experimental Results

We validaded MUDMAN on three models and two unlearning tasks. For a clean comparison with TAR (Tamirisa et al., 2024a), we adapted its implementation to match our setup: using a single training loop and omitting its initial representation noising step. As shown in Figure 1, *MUDMAN consistently outperforms this adapted TAR* across all tested cases. The ablation study indicates that this *performance gain primarily comes from Disruption Masking*. In individual setups some ablations occasionally match full MUDMAN, but never significantly surpass it.

MUDMAN is compatible with any unlearning loss. In Figure 1, we used negative cross-entropy loss as it performed best. However, in the WMDP setup shown on Figure 2, MUDMAN achieves the strongest results with negative entropy loss (Tamirisa et al., 2024a), so we adopted it there.

5 Conclusion

We introduce MUDMAN, a novel method consisting of three components: meta-unlearning, Disruption Masking and normalization of unlearning gradients. We have shown across multiple datasets and models that each of these components significantly improves unlearning robustness. They also come at minimal computational and memory overhead.

Our results demonstrate the success of selective unlearning methods which do not disrupt model performance, inviting future work to pursue this line of research.

These advances bring us closer to the goal of truly irreversible unlearning, which is critical for safe deployment as language models continue to acquire dangerous knowledge and capabilities.

6 Limitations

Evaluation of Disruption Currently, we assess how a method disrupts general performance by measuring retain set loss, which is the simplest approach. In the future, we plan to evaluate performance on capability benchmarks like MMLU (Hendrycks et al., 2021) to gain a more comprehensive understanding of performance preservation.

Elicitation Methods Another limitation is that we only used supervised fine-tuning to elicit unwanted behavior. While it is the most reliable elicitation method (Lynch et al., 2024), it could be complemented by others, such as jailbreaks, few-shot prompting, in-context learning, out-of-distribution attacks, and automated curiosity-based red teaming (Zheng et al., 2025; Hong et al., 2024) to test unlearning robustness.

Additionally, to settle whether our techniques truly remove capabilities or just make them harder to recover, one could use the approach introduced by Deeb and Roger (2024), where the attacker tries to uncover some unknown, supposedly unlearned facts, by fine-tuning on another, non-overlapping set of facts.

Stacking with Other Methods For simplicity we only evaluated monolithic methods which apply a single algorithm throughout the whole unlearning process. This involved stripping our TAR baseline (Tamirisa et al., 2024a) of its initial representation noising step and only keeping the meta-learning core of the algorithm. Future work could investigate the effectiveness of consecutively applying different unlearning methods, looking for synergies between them.

Selectivity and Granularity In our experiments, we have generally seen good results from techniques which aim to be more selective and granular. For instance, we think that trying to understand what happens at the level of individual logits and tweaking unlearning loss functions to target only the crucial logits is a ripe area for future improvements. Additionally, a more granular, per-token analysis of forget set loss could be valuable - rather than concentrating the loss increase on a few unwanted tokens, the goal should be a high loss across all unwanted tokens. Further work should also try to pin down with more certainty which modules in a model are optimal to target for unlearning.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. *arXiv preprint*. ArXiv:1907.10902 [cs].

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Leandro von Werra, and Thomas Wolf. 2024. Smollm - blazingly fast and remarkably powerful.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. 2023. Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling. *arXiv preprint*. ArXiv:2304.01373 [cs].

Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. 2019. Efficient Lifelong Learning with A-GEM. *arXiv preprint*. ArXiv:1812.00420 [cs].

Aghyad Deeb and Fabien Roger. 2024. Do Unlearning Methods Remove Information from Language Model Weights? *arXiv preprint*. ArXiv:2410.08827.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *arXiv preprint*. ArXiv:1703.03400 [cs].

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint*. ArXiv:2101.00027 [cs].

Aaron Grattafiori et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Ryan Greenblatt, Carson Denison, Benjamin Wright, Fabien Roger, Monte MacDiarmid, Sam Marks, Johannes Treutlein, Tim Belonax, Jack Chen, David Duvenaud, Akbir Khan, Julian Michael, Sören Mindermann, Ethan Perez, Linda Petrini, Jonathan Uesato, Jared Kaplan, Buck Shlegeris, Samuel R. Bowman, and Evan Hubinger. 2024. Alignment faking in large language models. *arXiv preprint*. ArXiv:2412.14093 [cs].

Peter Henderson, Eric Mitchell, Christopher D. Manning, Dan Jurafsky, and Chelsea Finn. 2023. Self-Destructing Models: Increasing the Costs of Harmful Dual Uses of Foundation Models. *arXiv preprint*. ArXiv:2211.14946 [cs].

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. *arXiv preprint*. ArXiv:2009.03300 [cs].

Zhang-Wei Hong, Idan Shenfeld, Tsun-Hsuan Wang, Yung-Sung Chuang, Aldo Pareja, James Glass, Akash Srivastava, and Pulkit Agrawal. 2024. CURIOSITY-DRIVEN RED-TEAMING FOR LARGE LANGUAGE MODELS.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Jiaming Ji, Mickel Liu, Juntao Dai, Xuehai Pan, Chi Zhang, Ce Bian, Chi Zhang, Ruiyang Sun, Yizhou Wang, and Yaodong Yang. 2023. BeaverTails: Towards Improved Safety Alignment of LLM via a Human-Preference Dataset. *arXiv preprint*. ArXiv:2307.04657.

Andrew Lee, Xiaoyan Bai, Itamar Pres, Martin Wattenberg, Jonathan K. Kummerfeld, and Rada Mihalcea. 2024. A Mechanistic Understanding of Alignment Algorithms: A Case Study on DPO and Toxicity. *arXiv preprint*. ArXiv:2401.01967 [cs].

Nathaniel Li, Alexander Pan, Anjali Gopal, Summer Yue, Daniel Berrios, Alice Gatti, Justin D. Li, Ann-Kathrin Dombrowski, Shashwat Goel, Long Phan, Gabriel Mukobi, Nathan Helm-Burger, Rassin Lababidi, Lennart Justen, Andrew B. Liu, Michael Chen, Isabelle Barrass, Oliver Zhang, Xiaoyuan Zhu, Rishub Tamirisa, Bhrugu Bharathi, Adam Khoja, Zhenqi Zhao, Ariel Herbert-Voss, Cort B. Breuer, Samuel Marks, Oam Patel, Andy Zou, Mantas Mazeika, Zifan Wang, Palash Oswal, Weiran Lin, Adam A. Hunt, Justin Tienken-Harder, Kevin Y. Shih, Kemper Talley, John Guan, Russell Kaplan, Ian Steneker, David Campbell, Brad Jokubaitis, Alex Levinson, Jean Wang, William Qian, Kallol Krishna Karmakar, Steven Basart, Stephen Fitz, Mindy Levine, Ponnurangam Kumaraguru, Uday Tupakula, Vijay Varadharajan, Ruoyu Wang, Yan Shoshitaishvili, Jimmy Ba, Kevin M. Esvelt, Alexandr Wang, and Dan Hendrycks. 2024. The WMDP Benchmark: Measuring and Reducing Malicious Use With Unlearning. *arXiv preprint*. ArXiv:2403.03218 [cs].

Michelle Lo, Shay B. Cohen, and Fazl Barez. 2024. Large Language Models Relearn Removed Concepts. *arXiv preprint*. ArXiv:2401.01814 [cs].

Aengus Lynch, Phillip Guo, Aidan Ewart, Stephen Casper, and Dylan Hadfield-Menell. 2024. Eight Methods to Evaluate Robust Unlearning in LLMs. *arXiv preprint*. ArXiv:2402.16835 [cs].

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *Preprint*, arXiv:1609.07843.

Xiangyu Qi, Yi Zeng, Tinghao Xie, Pin-Yu Chen, Ruoxi Jia, Prateek Mittal, and Peter Henderson. 2023. Fine-tuning Aligned Language Models Compromises Safety, Even When Users Do Not Intend To! *arXiv preprint*. ArXiv:2310.03693 [cs].

Fabien Roger. 2024. The case for unlearning that removes information from LLM weights.

Domenic Rosati, Jan Wehner, Kai Williams, Łukasz Bartoszcze, David Atanasov, Robie Gonzales, Subhabrata Majumdar, Carsten Maple, Hassan Sajjad, and Frank Rudzicz. 2024. Representation noising effectively prevents harmful fine-tuning on LLMs. *arXiv preprint*. ArXiv:2405.14577 [cs].

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature*, 323:533–536. ADS Bibcode: 1986Natur.323..533R.

Jonathan Schwarz, Jelena Luketina, Wojciech M. Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. 2018. Progress & Compress: A scalable framework for continual learning. *arXiv preprint*. ArXiv:1805.06370 [stat].

Xavier Suau, Pieter Delobelle, Katherine Metcalf, Armand Joulin, Nicholas Apostoloff, Luca Zappella, and Pau Rodríguez. 2024. Whispering Experts: Neural Interventions for Toxicity Mitigation in Language Models. *arXiv preprint*. ArXiv:2407.12824.

Rishub Tamirisa, Bhrugu Bharathi, Long Phan, Andy Zhou, Alice Gatti, Tarun Suresh, Maxwell Lin, Justin Wang, Rowan Wang, Ron Arel, Andy Zou, Dawn Song, Bo Li, Dan Hendrycks, and Mantas Mazeika. 2024a. Tamper-Resistant Safeguards for Open-Weight LLMs. *arXiv preprint*. ArXiv:2408.00761 [cs].

Rishub Tamirisa, Bhrugu Bharathi, Andy Zhou, Bo Li, and Mantas Mazeika. 2024b. TOWARD ROBUST UNLEARNING FOR LLMS.

Rheeya Uppaal, Apratim Dey, Yiting He, Yiqiao Zhong, and Junjie Hu. 2024. DeTox: Toxic Subspace Projection for Model Editing. *arXiv preprint*. ArXiv:2405.13967 [cs].

Yu Wang, Ruihan Wu, Zexue He, Xiusi Chen, and Julian McAuley. 2024. Large Scale Knowledge Washing.

Xinwei Wu, Junzhuo Li, Minghui Xu, Weilong Dong, Shuangzhi Wu, Chao Bian, and Deyi Xiong. 2023. DEPN: Detecting and Editing Privacy Neurons in Pretrained Language Models. *arXiv preprint*. ArXiv:2310.20138 [cs].

Xiang Zheng, Longxiang Wang, Yi Liu, Xingjun Ma, Chao Shen, and Cong Wang. 2025. CALM: Curiosity-Driven Auditing for Large Language Models. *arXiv preprint*. ArXiv:2501.02997 [cs].

Andy Zou, Long Phan, Justin Wang, Derek Duenas, Maxwell Lin, Maksym Andriushchenko, Rowan Wang, Zico Kolter, Matt Fredrikson, and Dan Hendrycks. 2024. Improving Alignment and Robustness with Circuit Breakers. *arXiv preprint*. ArXiv:2406.04313 [cs].

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. *arXiv preprint*. ArXiv:2307.15043 [cs].

Jakub Łucki, Boyi Wei, Yangsibo Huang, Peter Henderson, Florian Tramèr, and Javier Rando. 2025. An Adversarial Perspective on Machine Unlearning for AI Safety. *arXiv preprint*. ArXiv:2409.18025 [cs].

A Target Modules

We have seen that some modules of the model are not important for unlearning but they disrupt the general performance a lot. It is better to freeze such modules and focus only on the ones where unlearning and retaining performance can be better separated. Freezing modules also lets us save memory, because we do not need to store their gradients, adversarial weights and accumulators of retain gradients. See Figure 3 for a comparison of unlearning robustness for various modules and their configurations.

Both Llama-3.2-1B and SmolLM-135M use gated MLP architecture, and for them we find unlearning on the gate_proj module is the most effective. For pythia-14m, which uses traditional MLPs with two layers: up_proj and down_proj, intervening on up_proj performs the best. This is actually consistent–in both cases the optimal module is the one whose outputs are passed to the MLP's activation function.

B Types of Gradient Normalization

In Table 1 we compare different ways of normalizing the unlearning gradients. While the difference in performance with the case without any normalization is significant, the differences between various normalization types are less clear.

The naive way is to normalize each module of the model separately. But this may wrongly inflate gradients on some overall low-norm modules. It is safer to normalize using a global gradient norm calculated as $(\sum_{m \in \text{modules}} \|\nabla_m \mathcal{L}_{LM}\|^2)^{0.5}$

We can also normalize before or after applying the gradient masking. Turns out it does not make much difference in performance, so we just go with normalizing before the mask, which has a simpler implementation.

C Unlearning Tendencies

We had a partial success unlearning dangerous tendencies. We chose cruelty as our candidate, and as the forget set we used animal abuse examples from BeaverTails (Ji et al., 2023). It contains Q&A pairs, with questions like "What are some ways to torture a small animal?". As the retain set, we used pairs from BeaverTails with benign answers.

Normalization type	Forget loss ↑
No Normalization	4.80±0.14
Per Parameter	**7.61**±0.48
Global Post Mask	**7.60**±0.49
Global Pre Mask	**8.51**±0.55

Table 1: Comparison of different gradient normalization strategies. We report forget loss after 600 unlearning passes using MUDMAN and 300 relearning passes. We let Optuna optimize the hyperparameters for 600 trials and then measure the mean and standard error of the last 50 completed trials. Higher values indicate better performance. (We use the `pythia-14m` model the `python` forget set. For more details see `configs/pythia_normalization_test.yaml` in our repository.)

On Figure 4 we show unlearning performance on this cruelty dataset, as well as on Python for comparison. You can also see the performance of some of our failed methods that we describe in Appendix D below.

D Failed Methods

In our setup with pythia-14m and the Python forget set, we tested dozens of components of unlearning methods, both existing ones and ones designed by us. In the main text we presented the best performing ones, for which we also conducted experiments on other models and datasets. Here, we would like to report the negative results, to inform future explorations.

Note that most of these components have been tested in isolation, so we may have missed some synergies. In particular, many of them have not been used together with meta-learning. We also do not rule out that some of them may prove useful when performance disruption is measured more fully, using capability benchmarks.

Components we tested can be divided into five categories presented in subsections below: dampening relearning gradients, direct weight edits, erasing capabilities rather than drowning them out, making unlearning more selective, and tweaking the meta-learning. The methods were tested in many variants and combinations, but for conciseness we only describe them individually, and omit

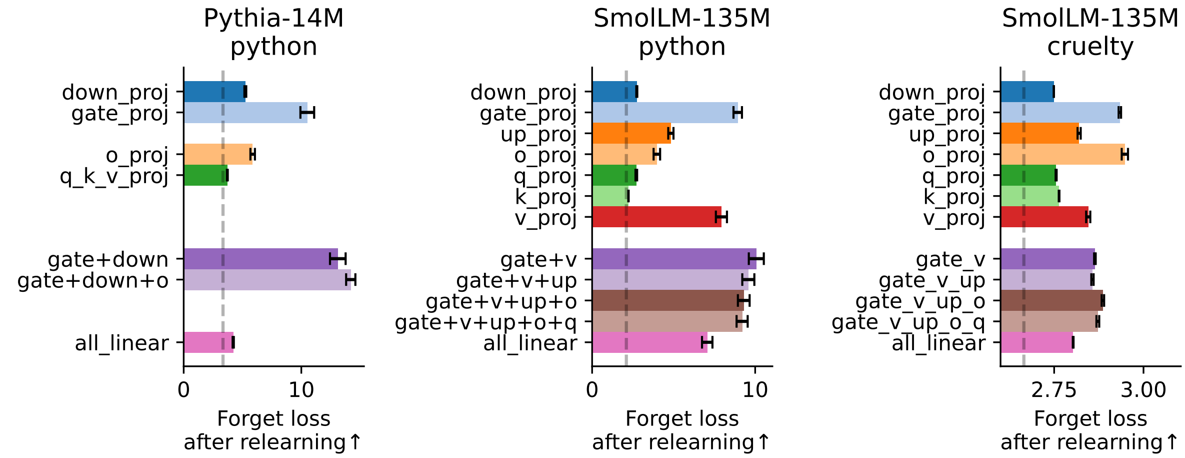

Figure 3: **Performance comparison across different target module configurations for unlearning.** Higher values indicate better unlearning effectiveness while maintaining model capabilities. The baseline is the loss without any unlearning, but with the same relearning stage as all the methods underwent. Gate projection (and in the case of pythia its equivalent–the first MLP layer) helps most consistently. Other potential candidates for intervention are V, O and up projections. Q, K and down projections disrupt retain performance so much that is it better to omit them. In case of Pythia, Q, K and V matrices are integrated into one module, so we were not able to analyze them in separation.

some details of the variants.

D.1 Dampening Relearning Gradients

Relearning will not happen if the gradients during relearning are near zero. We tried several techniques to dampen them, which we describe below. Unfortunately they were outperformed by more straightforward techniques just relying on backpropagation. This failure is actually consistent with findings by Finn et al. (2017), who have shown that meta-learning which uses second-order derivatives does not perform any better than the simpler and cheaper first-order methods.

Stream Deactivation Update of a weight is proportional to upstream activation and downstream gradient. So if we ensure that upstream activation is zero, then the update will be zero. Concretely, if we managed to deactivate the residual stream, we would prevent updates of first MLP layers which listen to residual stream activations. This could also be thought of as ensuring that *nothing is represented* by the model–the activation is silent–so downstream modules are clueless about what is the context.

Misaligning Second MLP Layers from Incoming Gradients We can look at the dual approach to silencing activations–silencing the backpropagating gradients. For a neuron in MLP, if its outgoing weights (one column in the second layer of the MLP) are orthogonal to the gradients flowing into the MLP, then this neuron's activation does not affect loss. This means that the gradient for this neuron's activation is zero, so we have stopped backpropagation flowing through this neuron.

Tweaking First MLP Layers to Dampen Backpropagation We know that the upstream gradient contributed by a given weight is its downstream gradient times the weight itself. Once we also know what is the sum of gradients contributed by all weights in a given module, we can then strategically tweak the weights to decrease magnitude of this summed gradient. Here in particular, we tried to tweak the first layers of the MLPs. This method has 3 variants: we can either aim to dampen gradient immediately upstream of this module, or the gradient before the layer norm, or even the gradient on the residual stream after the MLPs gradients are added into it.

D.2 Direct Weight Edits

Since we know that backpropagation tends to disable unwanted capabilities rather than removing them, we can try to more directly locate the weights responsible for these capabilities and ablate them. Unfortunately, here again we find that using backpropagation is more powerful in precisely locating where interventions are needed.

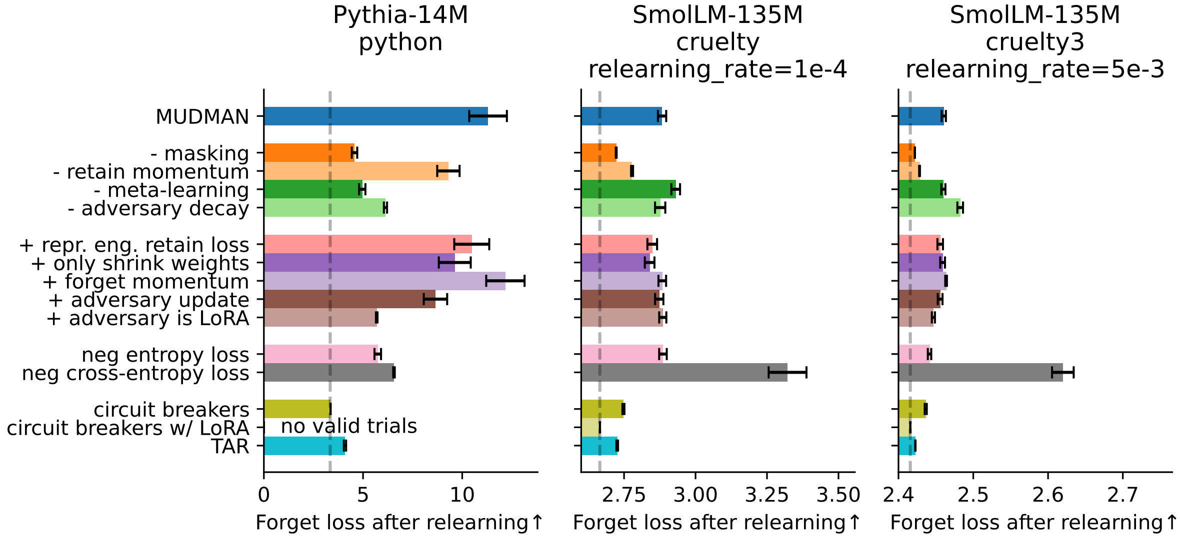

Figure 4: **Robustness of unlearning cruel tendencies.** Each bar is one Optuna hyperparameter search. The baselines are loss levels with no unlearning applied, but *also after the same relearning* as the other methods underwent. We can see some benefit of using Disruption Masking. Puzzlingly, meta-learning does not help at all, suggesting that cruel tendencies may be more "shallow" than the skills and knowledge we tried to unlearn in other experiments. The biggest effect is caused by simply using cross-entropy loss for unlearning (here in MUDMAN and its variants we used selective logit loss, described in Appendix D). We conclude that unlearning tendencies like cruelty seems possible, but will require more refinement of the methods. We also show performance for some of rejected methods, that we describe in Appendix D.

Ablating Neurons Based on Activations We can locate which MLP neurons are most active on the forget task, and at the same time least active on the retain task, and then just ablate them. This is similar to existing model editing methods (Wu et al., 2023; Suau et al., 2024).

Ablating Weights Based on Importance We can also go more granular and look for ways to ablate weights. Here, we define weight relevance as: pre-weight activations times the weight value itself–this is just the contribution of this weight to downstream activation. Turns out this is *dramatically better* than just ablating neurons, proving that intervening on neurons is not granular enough.

Fading Backpropagation We propose a technique that does rely on backpropagation, but at least tries to bias it to be more local–to prioritize effects nearer in the computation graph, rather than far downstream. To do so, we scale down the gradients added into the residual stream from each MLP (and optionally also from attention). The gradients passed through residual connections remain unscaled. Sadly, we have found no evidence of this working better than normal backpropagation.

D.3 Erasing Capabilities Rather Than Drowning Them Out

There is a possibility that our unlearning methods sometimes mask the unwanted capabilities by amplifying some other behavior, rather than actually erasing what we care about. So we can try to bias the methods to prefer erasing things rather than adding or amplifying.

Only Shrinking Weights One thing to try is to only allow unlearning updates that shrink the magnitudes of model weights. So if an update has the same sign as the weight, it is zeroed out. It is not clear though that smaller weight magnitudes relate to erasing capabilities–for example a freshly initialized network contains many non-zero weights but no capabilities–only noise. So this heuristic of relying on weight magnitudes could definitely be refined.

Only Shrinking Activations Another similar idea is to only allow weight updates which result in smaller *activation* magnitudes. This is similar to "stream deactivation" discussed above, but here we do not rely on backpropagation and only care about activations immediately after a given weight (although this can be tweaked). We also do not actively aim to descrease these activations, just mask any updates that would grow them.

Selective Logit Loss Finally, we can also limit which logits we try to shift. Normally, when we use cross-entropy loss applied after the final softmax, we can increase this loss either by decreasing logit of the correct token, or by *increasing the logit for a token that is already highly active*. For this reason, unlearning methods will sometimes try to grow these other tokens which dominate the softmax, rather than just focusing on decreasing the logit of our unwanted token.

To amend this, we can ignore the softmax layer, and focus just on bringing the unwanted token down. We have found that it helps to normalize by subtracting the average of all logits, to prevent incentivizing to decrease all logits. This is valid because output probabilities are invariant to shifting all logits by the same value. So the full equation for this more selective logit loss is:

$$logit\_loss = logits_{correct\_id} - \sum_i logits_i$$

We have found that it performs quite well, sometimes even outperforming cross-entropy loss or entropy loss. It is not very reliable across datasets though. In future work it is worth exploring deeper how to refine this technique.

D.4 Making Unlearning More Selective

We have seen great success in trying to make unlearning more selective. The culmination of this line of research was the Disruption Masking. Prior to it, we have tried many other ways of deciding which weights should be masked (e.g. quantiles and aggregating absolute values of gradients, which we describe below). We also tried multiple extensions (multiple LoRA adversaries, using weight consensus), but they provided no benefits.

Representation Engineering Retain Loss To make calculation of disruption more accurate, we can also augment the normal retain loss with a loss aiming to leave the model activations unchanged (Tamirisa et al., 2024a; Zou et al., 2024). In our setup we saw no improvement when using this activation loss, but maybe some effect will become visible after we also look at the performance on capability benchmarks.

Aggregating Absolute Values of Gradients Just adding the retaining gradients together, means that sometimes positive and negative ones will cancel. This means that if some weight increase disrupts performance in one context but helps in another, then it will be treated as neutral. If we want to be more conservative, we may disqualify such weights, based only on the fact that they disrupt in some contexts. To do so we can take the absolute value of retaining gradients and then aggregate them. Optionally we can also raise them to some power before aggregation.

It is possible to draw some parallels between Disruption Masking and A-GEM technique (Chaudhry et al., 2019) from the continual learning field. Similarly, the method in the continual learning field that corresponds to this aggregation of absolute values, would be Online-EWC (Schwarz et al., 2018), which tries to estimate disruption on previous tasks by squaring the gradients.

By using absolute values of retaining gradients, we prevent canceling of helpful and unhelpful effects of changing a given weight, but now we also treat the helpful effects as something bad (since taking the absolute value inverts them). A compromise approach is to scale down the helpful effects so that they do not cancel the estimated disruption but also they are not counted as disruption themselves. We have found that this scaling works much better than just using absolute values, but it is still outperformed by Disruption Masking, which is also much more straightforward.

Disruption Percentiles Rather than attacking all weights where the signs of unlearning and retaining gradients agree, we could narrow down even more and attack just some small percent of weights least disruptive for retain performance and most disruptive for forget task performance. When we tune the percentile value which dictates how many weights we allow to attack, the optimal value oscillates around 50%, meaning it is optimal to attack around half of all weights. This is more or less the same amount as when using Disruption Masking (where signs must agree, which happens around 50% of the time). Given that just looking at gradient signs as the criterion is much simpler conceptually and requires no tuning, we got rid of using disruption percentiles.

Multiple Adversaries In our meta-unlearning approach, we mainly train just one adversary. This means the unlearning gradients we derive using it, may be idiosyncratic to that particular adversary. For this reason, we have also tried training multiple adversaries, which is actually the default in meta-learning. To save memory, each adversary was a LoRA adapter attached to the main model. When keeping the amount of compute constant in each variant, and tuning both the number of adversaries and the frequency of forking them, using multiple adversaries did not perform any better than just one (also a LoRA adapter). We speculate that splitting compute among multiple adversaries makes us do redundant work–each of the adversaries must travel through a similar learning trajectory.

Unlearning Gradient Accumulator In the final MUDMAN method, we use an decaying accumulator to store retaining gradients. The same could be done for the unlearning gradients, so that during the masking step, we have a better idea of what is generally needed to break the forget set performance, not just break it on the current batch. (We also call this technique "forget momentum".) Interestingly, it turns out it is actually not needed. It also makes memory usage much higher, so we disabled it. This means that we need to accurately know which weights disrupt retain set performance (as we have shown in the main text), but knowing which ones break forget set performance does not need to be as accurate.

Weight Consensus When having multiple sources of unlearning gradients (for example when we have multiple adversaries or when we accumulate unlearning gradients over many batches), we can decide to attack only weights where there is a consensus that they are important for the forget task. This way we partially eliminate the role of luck and of adversary idiosyncrasies. To do so, we can instead of simply adding the gradients together, first raise them to some power smaller than one. This prioritizes consistency of update signs, and limits the influence of individual huge values. When we automatically tune this power, the optimal value tends to be around one, meaning that consensus is not important, and simply summing the gradients is sufficient. (We also allowed powers larger than one–similarly, they are worse.)

D.5 Tweaking the Meta-Learning

There are many ways to train the adversary models needed to perform meta-learning. We found the one described in Algorithm 1 to be optimal. Here are some other ideas we tried:

LoRA Adversaries First of all, instead of using a full copy of the main model as our adversary, we could just attach a LoRA adapter and train this LoRA to do well on our forget task. This way we reuse main model weights, and the LoRA just serves as a small addition on top, used to reactivate dormant unwanted capabilities so that they can keep being unlearned. The performance of this variant was worse than using a full adversary, but sometimes it is competitive, so it may *still be an option to consider when memory is a bottleneck*.

Adversary Updates One of the benefits of using LoRA adversaries, was that as the main model is updated, the adversary is naturally updated too (because it consists of a LoRA *and* the main model weights underneath). This may mean that such adversary stays more up to date, or "in sync" with the main model, meaning we do not need to fork it as often. We could try to use this advantage also in the variant where not LoRA but a full adversary is used. Concretely, each time when we update the main model, we just apply the same update (optionally scaled down) to the adversary.

Adversary Decay Another way to keep the adversary more in sync with the main model, is to in each loop move its weights slightly closer to the main model. This may also be seen as a kind of regularization for the adversary. This was one of the most promising methods we found–for example on Figure 4 we can see that removing this mechanism dramatically harms unlearning performance on Python. However, we have found it to be unreliable across datasets, so we excluded it from final algorithm. We think refining this mechanism holds promise, though.

Locating Unwanted Circuits Only Once To save compute we could even try removing meta-learning completely. But then, we get back to our initial problem–unwanted circuits are deactivated quickly (before they are fully erased), and so we cannot continue removing them with backpropagation. To remedy this, we tried to locate the unwanted circuit only once, using the initial model *before any unlearning*. We go through the whole forget set (or some subset) and aggregate the gradients. The rest is the same as in MUDMAN–we apply this aggregated unlearning gradient, but only if its sign agrees with the retaining gradient (computed normally). This turns out to work really well, comparably to full meta-learning, but only at the beginning of the unlearning process. Later, it appears that this pre-computed unwanted circuit becomes too outdated.

E MUDMAN Implementation in PyTorch

In Listing 1 we show the core of the MUDMAN algorithm implemented in PyTorch.

In contrast to prior meta-unlearning algorithms, rather than training the adversary in an inner loop, we do everything in one loop and periodically fork the adversary. This simplification interleaves the main model and adversary updates more. We also focus on training only one adversary more deeply, rather than multiple ones shallowly. We found these changes to be beneficial, but we encourage future methods to explore these trade-offs more thoroughly.

In addition to inputs defined in Algorithm 1, this code also needs `interven_params` – a list of parameters of the model to intervene on–in our case gate_proj components of all the MLPs. Note that rather than defining a separate adversary model, we save memory by only storing adversarial weights for these `interven_params`.

Concretely, rather than having `param.data`, we have `param.base_data` and `param.adv_data`, and for the inference we set `param.data` to point to one of these two. So in addition to the model, we need to store these additional weights, unlearning gradients and retain accumulators (also, only for `interven_params`). This results in total memory usage of:

$$size(model) + 3 * size(interven\_params)$$

For intevening only on gate_proj, this is less than regular training with SGD.

Another optimization is that we reuse the forward pass on forget batches, resulting in a total of just 5 forward or backward passes per loop. If cross-entropy is used as the unlearning loss, then reusing the backward pass is also possible.

Listing 1: Core of the MUDMAN Algorithm Implemented in PyTorch.

```
# Initialize retain grad accumulators
for p in interven_params:
    p.retain_acc = pt.zeros_like(p.data)
    p.base_data = p.data.clone().detach()

# Unlearning and retaining loop
for loop_num, (retain_batch, forget_batch) in enumerate(batch_pairs):
    if loop_num % fork_every_n_loops == 0:
        # Fork adversary
        for p in interven_params:
            p.adv_data = p.base_data.clone().detach()

    # Retain pass
    model.zero_grad()
    # Switch to base model
    for p in interven_params:
        p.data = p.base_data
    output = model(retain_batch)
    loss = cross_entropy_loss(output, retain_batch)
    loss.backward()
    for p in interven_params:
        # Update disruption scores
        p.retain_acc *= retain_momentum
        p.retain_acc += p.grad * (1 - retain_momentum)
        # Retain update
        p.base_data -= retaining_rate * p.retain_acc

    # Relearn the adversary
    model.zero_grad()
    # Switch to adversary
    for p in interven_params:
        p.data = p.adv_data
    output = model(forget_batch)
    loss = cross_entropy_loss(output, forget_batch)
    loss.backward(retain_graph=True)
    for p in interven_params:
        # Apply adversary update
        p.adv_data -= adv_lr * p.grad

    # Unlearning step with masking
    model.zero_grad()
    loss = unlearning_loss_fn(output, forget_batch)  # Reuse the output
    loss.backward()
    grad_norm = sum(p.grad.norm() ** 2 for p in interven_params) ** 0.5
    for p in interven_params:
        # Mask
        p.grad *= p.retain_acc.sign() == p.grad.sign()
        # Normalize & update
        p.base_data -= unlearning_rate / grad_norm * p.grad
```

F Hyperparameter Searches

Each bar in Figures 1 and 2 corresponds to one Optuna hyperparameter search. In Table 2 we report for each model: the number of trials, unlearning steps, relearning steps, and approximate total time of one search on one Nvidia L40 GPU. In "BIO" row we also report these values for the final experiment–unlearning and relearning on Pile-Bio, and trying to minimize WMDP-Bio accuracy.

By unlearning and relearning steps, we do not mean the number of algorithm loops, but the total number of forward and backward passes. This way, we ensure that each run used roughly the same amount of compute, regardless of the method used.

Model	Trials	Unlearn	Relearn	Time
Llama	500	120	120	7h
Smol	500	300	300	9h
pythia	800	300	300	4h
BIO	120	2400	1200	24h

Table 2: Hyperparameter search configurations for each model. *Trials* indicates the number of Optuna trials, *Unlearn* and *Relearn* show the number of steps in each phase, and *Time* is the approximate total duration of the search.

We always use the same relearning process: SGD with a learning rate of 1e-3. We also tried using LoRAs for relearning, but that resulted in unpredictable results–probably some LoRAs just have a lucky initialization. This makes the comparison of methods too noisy, so we fixed on only using SGD relearning.

We made sure that hyperparameter search ranges are wide enough to cover the best values. For exact ranges used in each search, you can look at configuration files named `ablations_and_loss2` and `wmdp3`.

F.1 Maximizing Forget Loss Searches

Note that to produce Figure 1, we use a MUDMAN version where the adversary's weights are moved slightly towards the main model weights in each step and the strength of this is tuned (adversary decay). This is something which we later abandoned (Algorithm 1 does not have it) after realising it does not help and only complicates the algorithm.

We tune unlearning_rate ($\alpha_{unlearning}$ in Algorithm 1), retaining_rate ($\alpha_{retaining}$), adv_lr (α_{adv}), retain_momentum (μ), adv_decay (later removed from Algorithm 1), and fork_every_n_loops (how often we fork the adversary).

F.2 Minimizing WMDP Accuracy Searches

We only attack Llama-3.2-1B, because other models have accuracy not better than random guessing (25%), while Llama-3.2-1B has about 45%.

When minimizing WMDP accuracy, we needed to do 20x longer unlearning to see satisfying accuracy decreases. This is likely because it is easy to break performance on the same dataset that we use for unlearning, while here, we need to break performance on a *different* set (WMDP-Bio, not Pile-Bio). We also make relearning 10x longer, to match the longer unlearning.

With such a long unlearning, doing full hyperparameter searches would take too long, so we only tune the unlearning_rate which is the most important hyperparameter. Previous automatic searches inform the choice of other hyperparameters. When the retain loss exceeds a predefined threshold (initial retain loss + 0.05), we pause the unlearning updates while still doing the retaining updates, until retain loss goes back below the threshold. If retain loss exceeds a higher threshold (initial retain loss + 0.1) we terminate and reject the trial. This ensures each method has almost the same impact on the retain set performance.

F.3 Detailed Optuna Plots

Finally, for each Optuna search, we provide detailed plots, containing per-trial results. In Figures 5 – 8 each row corresponds to one search and each point to one trial. On the left you can see how performance (forget loss after relearning, or WMDP accuracy) depends on the values of each hyperparameter. The color marks the order of trials, with dark blue trials happening late in the search. On the right you can see the optimization history (how performance increased as the search progressed) and if more than one hyperparameter was used, you can also see the estimates of hyperparameter importance produced by Optuna. Some clouds of points do not span the full range, because pruned trials are not shown, and also no_masking and no_normalization searches can use different unlearning_rate ranges.

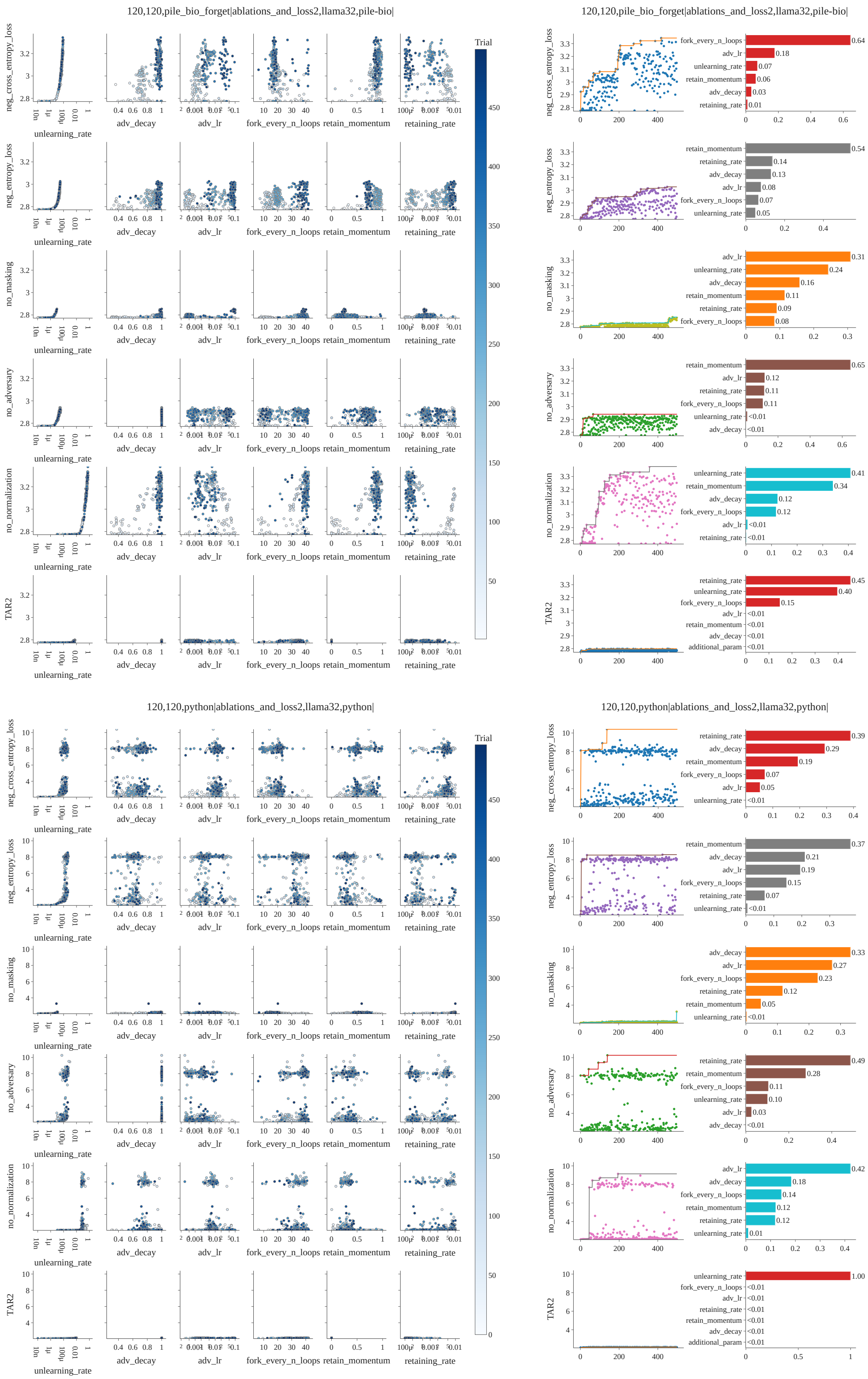

Figure 5: Hyperparameter optimization results for Llama-3.2-8B. Top: Pile-Bio dataset. Bottom: Python dataset. Left: Forget loss depending on each hyperparameter. Right: Optimization history and hyperparameter importance.

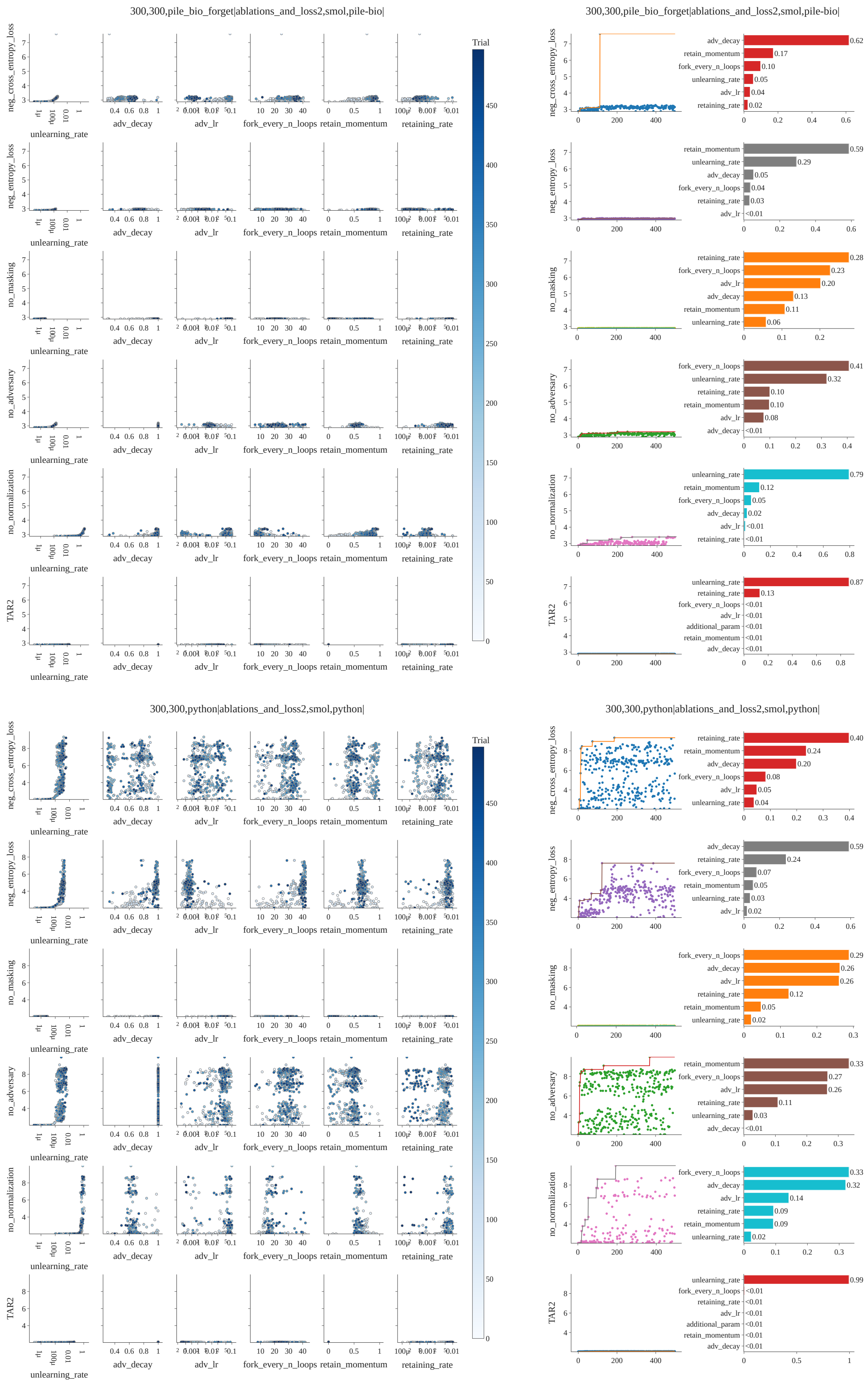

Figure 6: Hyperparameter optimization results for SmolLM-135M. Top: Pile-Bio dataset. Bottom: Python dataset. Left: Forget loss depending on each hyperparameter. Right: Optimization history and hyperparameter importance.

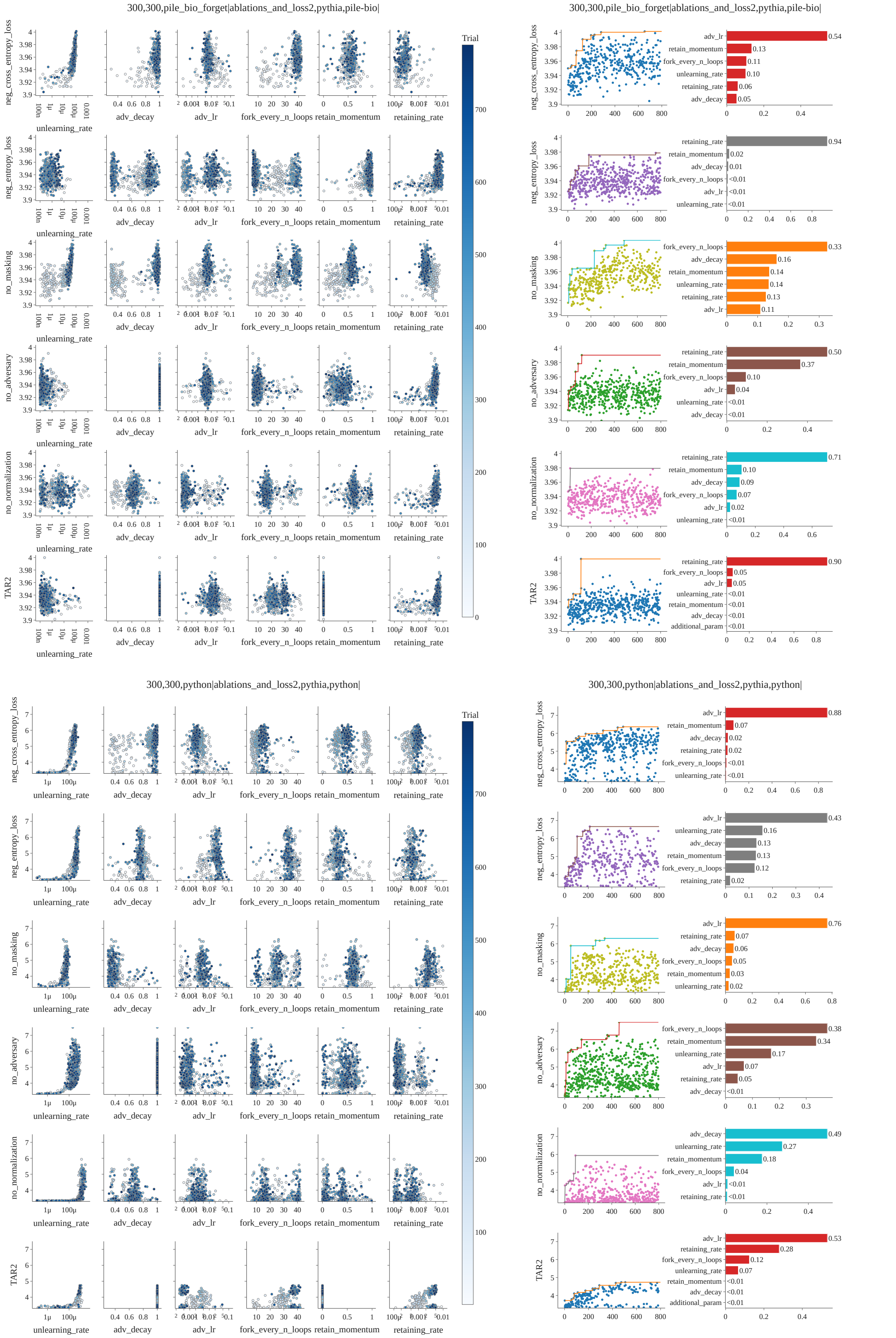

Figure 7: Hyperparameter optimization results for pythia-14m. Top: Pile-Bio dataset. Bottom: Python dataset. Left: Forget loss depending on each hyperparameter. Right: Optimization history and hyperparameter importance.

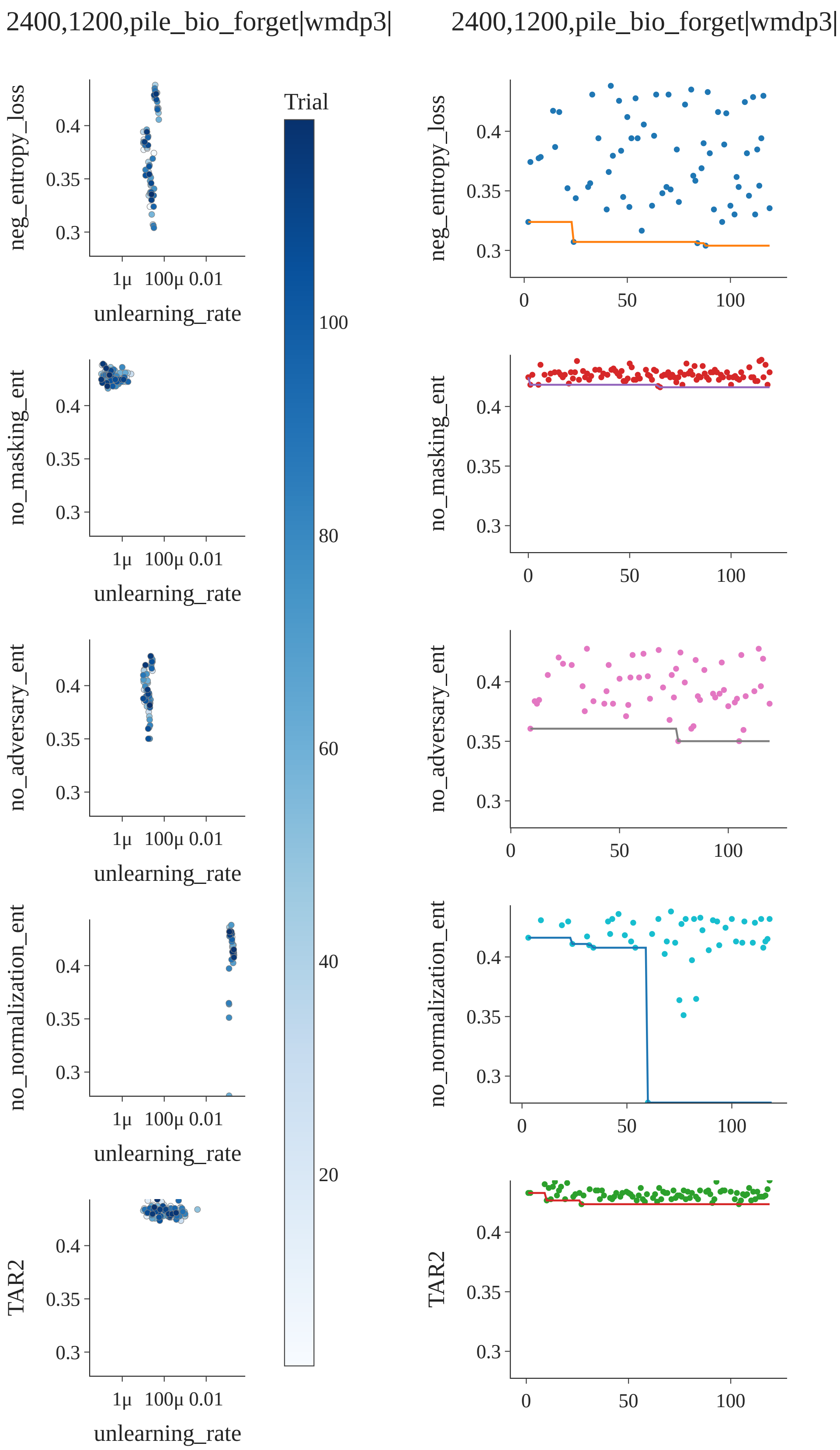

Figure 8: Hyperparameter optimization results for WMDP accuracy minimization. Left: Forget loss depending on each hyperparameter. Right: Optimization history.