

Hash Routed Delta Patches for Fast Knowledge Updates in Small LLMs

Stanford CS224N Custom Project

Arash Hamzehlou

Department of Computer Science
Stanford University
ahamz@stanford.edu

Abstract

Large language models get facts wrong and also get out of date. Model editing tries to fix one fact without retraining the whole model, but the problem gets much harder when we do many edits in a row. A good continual editor should still answer the edited question correctly, work on paraphrases, and avoid changing unrelated answers. In this project I study a simple routed editing method for a small language model. For each edit, I train a small low-rank residual patch to imitate the hidden-state change caused by adding the new fact in context. At test time, a SimHash router selects a stored patch from the question hidden states and applies it on top of the frozen base model. I evaluate on deterministic 100 to 500 edit subsets of CounterFact with rewrite exact match, paraphrase exact match, and locality preservation. At 500 edits, routed patches reach 0.818 rewrite exact match and 0.9942 locality preservation, while a continually trained shared LoRA baseline reaches 0.096 rewrite exact match and 0.0004 locality preservation. The main weakness is paraphrase routing. Routed patches get only 0.001 paraphrase exact match, while an oracle router raises that number to 0.407. This shows that the stored patches are useful, but the current router does not fire on paraphrases.

1 Key Information

- Mentor: Qinan Yu
- External Collaborators: None
- Sharing project: No
- Late Days: None used

2 Introduction

Large language models store a lot of factual knowledge, but that knowledge can become stale or wrong. Full retraining is expensive, so model editing asks a more focused question: can we change one fact after training and keep the rest of the model stable? Recent work shows that this is possible for single edits, but the continual setting is much harder [1, 2, 3, 4]. After many edits, methods often run into a three-way tradeoff between reliability, generalization, and locality [4]. In this report, reliability means answering the edited prompt correctly. Generalization means giving the new answer for paraphrases. Locality means leaving unrelated prompts alone.

There are two common ways to handle updates. One option is to directly change shared model weights, as in ROME, MEMIT, or LoRA style adaptation [1, 2, 5]. This can work well for a few edits, but repeated updates can interfere with each other. Another option is to keep the model frozen and feed the new fact in context at test time. This avoids weight interference, but it requires the memory snippet to be present at inference and it may still fail on harder prompts. I wanted a simple middle

ground between these two extremes. Instead of doing a permanent surgery on shared weights, I keep one small patch per edit and apply it only when the router says the question matches that edit.

My method stores one tiny residual patch per edit and routes to that patch only when the question looks like it belongs to that edit. The main idea is to learn the hidden-state change that the base model would have made if the new fact were already in the prompt. This is close in spirit to the dual-memory idea in WISE, but the side memory here is a bank of low-rank patches rather than a separate editable module [4]. The router uses a SimHash style key built from question hidden states [6].

The project makes three main contributions. First, it builds a complete continual editing pipeline for Phi-3-mini-4k-instruct with patch training, routing, and shared evaluation. Second, it gives a clean comparison against no update, a context-only oracle, and a continually trained shared LoRA baseline on CounterFact [7, 1, 5]. Third, it shows a clear failure mode. The patch bank stays strong on rewrite prompts and locality, but the current SimHash router almost never triggers on paraphrases.

3 Related Work

Direct weight editing methods change internal model parameters so that a target fact becomes easier to produce. ROME locates a factual association and edits a small part of the network for one fact [1]. MEMIT extends this idea to many edits at once [2]. These methods are appealing because they edit the model itself, but repeated updates can still create conflicts when many facts compete for the same shared weights.

Other work tries to make editing more scalable. MEND learns a gradient-based editor [3]. SERAC stores an external memory of edits and retrieves from that memory at test time [8]. WISE argues that lifelong editing is hard because reliability, paraphrase generalization, and locality pull against each other, and it uses a router between main and side memory [4]. My method is closest to the routing story in WISE, but it uses simple low-rank residual patches instead of a learned side memory network.

I also compare against a shared LoRA adapter as a simple continual baseline [5]. This is not a standard model editing method by itself, but it is a useful test of whether a single compact adapter can absorb many sequential factual changes without heavy side effects. Compared with ROME or MEMIT, my method does not try to find one causal weight location and overwrite it in shared model parameters. Compared with SERAC or WISE, it does not train a full side memory network. The main new pieces in this project are an aligned teacher-student delta target, a bank of one low-rank patch per edit, a hidden-state SimHash router, and a confuser loss that reduces damage when the router makes a mistake.

4 Approach

Figure 1 shows the whole pipeline. All methods use the same prompt wrapper with an instruction block, a fixed-size *Memory* slot, the question, and an answer prefix. The key design choice is that teacher and student inputs have the same layout and the same memory length. The teacher sees a short factual snippet in the memory slot, while the student sees blank memory. This keeps token positions aligned.

Patch module. For edit e , I attach a low-rank patch P_e at one transformer layer. If $h \in \mathbb{R}^d$ is the last-token hidden state at the patch layer, the patch predicts

$$P_e(h) = \alpha_e U_e (V_e h), \quad (1)$$

where $U_e \in \mathbb{R}^{d \times r}$, $V_e \in \mathbb{R}^{r \times d}$, r is small, and α_e is a learned scalar. The edited hidden state is

$$\tilde{h} = h + P_e(h). \quad (2)$$

The base model stays frozen. Only the patch parameters are learned.

Teacher-student supervision. For each edit prompt q and answer token y_t , I build two aligned inputs with the same answer prefix $y_{<t}$. The teacher input includes the new factual memory snippet

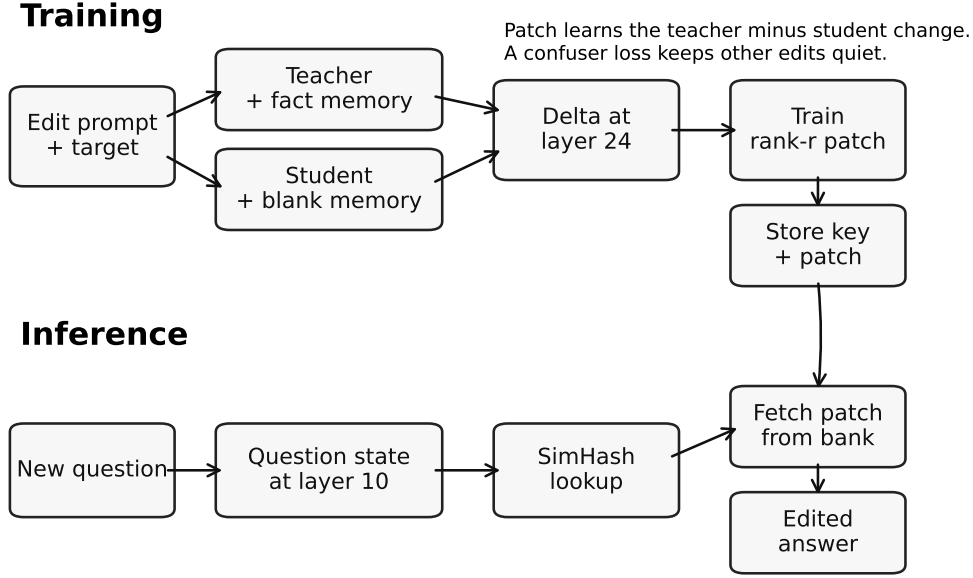


Figure 1: Overview of the patch-bank pipeline. During training, teacher and student runs use the same question and answer prefix, but only the teacher sees the new fact in memory. The system stores one low-rank patch and one SimHash key per edit. During inference, the router picks a patch from the question hidden states and applies it on top of the frozen base model.

m_e , and the student input uses blank memory. Let h_t^T and h_t^S be the teacher and student hidden states at the patch layer. The target residual is $h_t^T - h_t^S$. The main training loss is

$$\mathcal{L}_{\text{edit}} = \lambda_{\text{ce}} \mathcal{L}_{\text{ce}} + \lambda_{\Delta} \|P_e(h_t^S) - (h_t^T - h_t^S)\|_2^2 + \lambda_{\text{mag}} \|P_e(h_t^S)\|_2^2. \quad (3)$$

Here $\mathcal{L}_{\text{ce}}$ is next-token cross entropy on the gold answer token. The delta term makes the patch imitate the teacher change, while the magnitude term keeps the patch small.

I add two more regularizers. First, on a small set of generic negative prompts, I penalize any nonzero patch output. Second, I use a *confuser* loss on prompts from earlier edits:

$$\mathcal{L}_{\text{conf}} = \text{KL}(p_{\theta}(\cdot | x_c) \| p_{\theta, P_e}(\cdot | x_c)), \quad (4)$$

where x_c is a confuser prompt, p_{θ} is the frozen base model, and p_{θ, P_e} is the patched model. The full loss is a weighted sum of the edit, negative, and confuser terms. Intuitively, this tells a patch to help on its own edit while staying quiet on other questions.

Hash router. The router uses hidden states from an earlier layer. For a question with token index set Q , I average the hidden states at router layer ℓ_r ,

$$r(x) = \frac{1}{|Q|} \sum_{i \in Q} h_i^{(\ell_r)}. \quad (5)$$

Then I project $r(x)$ with a random matrix $R \in \mathbb{R}^{d \times b}$ and take the sign of each projection to get a b -bit key,

$$k(x)_j = \mathbf{1}[r(x)^{\top} R_j > 0], \quad j = 1, \dots, b. \quad (6)$$

At test time, the router finds the stored key with the smallest Hamming distance. If the best distance is above a threshold τ , no patch is applied. If there is a tie in Hamming distance, I break it with cosine similarity between the pooled router vectors.

Baselines. I compare against three baselines. **No update** uses the frozen model with blank memory. **Context oracle** adds the edit memory snippet at test time for the correct edit, so it acts like an upper bound for retrieval-style editing. **Shared LoRA** continually fine-tunes one shared adapter across edits with train prompts only [5].

What is original here. The main original part of the project is not a new large editing model. It is the combination of four simple ideas that work together in the continual setting: aligned teacher-student runs with matched memory length, one separate low-rank patch per edit, a hidden-state SimHash router, and a confuser loss. I used this design because it keeps edits separated, easy to log, and easy to remove later. That is different from direct editing methods like ROME, where many edits still touch shared weights.

5 Experiments

5.1 Data

I use two data sources. The main experiments use CounterFact, the benchmark introduced with ROME [1]. CounterFact edits are short subject-relation-object rewrites about named entities. In practice, they look like short factual questions with a new answer, for example affiliation, location, birthplace, or nationality style facts. Each row specifies a new target answer, a base prompt template, paraphrase prompts, and neighborhood prompts for locality. My data script converts each row into an `EditItem` with one memory snippet, one train prompt, held-out paraphrases, and locality prompts.

I also keep a tiny synthetic sanity set from the milestone stage. No other model was used to generate it. I wrote the edits by hand to test whether the system could learn facts that were very unlikely to appear in pretraining. The set has 3 edits and 12 total queries, with 2 direct prompts and 2 paraphrases per edit. The three facts are `launch_date(Project Orion)=2026-03-10`, `launch_date(Project Nova)=2027-11-01`, and `inventor(Project ZQ-67)=Arash Hamzehlou`. I do not use this tiny set for the main tables. I use it only as a sanity check and as a qualitative example of teaching clearly new information.

A small but important preprocessing step is a teacher compatibility filter. For every candidate edit, I try three simple memory templates, a plain statement, a `Fact:` prefix, and a `Question/Answer` form. I keep the edit only if the context-only teacher gets perfect exact match on the train and paraphrase prompts. This avoids training patches on noisy teacher targets. From the filtered pool I create deterministic 50, 100, 250, and 500 edit subsets with seed 42.

In the final 500-edit subset, each edit contributes one rewrite prompt, two paraphrase prompts, and ten locality prompts. That gives 6,500 evaluation queries at the 500-edit checkpoint. In the processed subset, I keep one train prompt per edit, so the deployed router stores one key per edit in the main runs.

5.2 Evaluation method

I report four main automatic metrics.

Rewrite exact match measures whether the model gives the new target answer on the train prompt for an edit. **Paraphrase exact match** measures the same thing on held-out paraphrases. **Locality preservation** measures how often a neighborhood prompt keeps the same answer as the frozen base model. I use preservation against the base model because the processed schema does not include gold locality labels. **Locality KL** measures the KL divergence between the edited and base next-token distributions on locality prompts.

For router analysis, I also report top-1 router accuracy, router coverage, and locality false positive rate. Exact match is the primary metric. I keep relaxed substring match only for error analysis, not for the main comparison.

5.3 Experimental details

All experiments use `microsoft/Phi-3-mini-4k-instruct` as the frozen base model [7]. The memory slot has 64 tokens. The router uses layer 10, 256-bit SimHash keys, and threshold $\tau = 8$. The final patch config, chosen by a small 50-edit dev sweep, uses patch layer 24, rank 16, learning rate 5×10^{-4} , 25 gradient steps per edit, confuser weight 0.3, and 8 confuser prompts per edit. The shared LoRA baseline uses rank 8, alpha 16, learning rate 2×10^{-4} , and 25 steps per edit.

Method	Rewrite EM	Paraphrase EM	Locality	Locality KL	Params	Sec / edit
No update	0.000	0.001	1.0000	8.1×10^{-14}	0	0.00
Context oracle	1.000	0.694	1.0000	8.1×10^{-14}	0	0.00
Shared LoRA	0.096	0.080	0.0004	5.2×10^{-4}	12.6M	3.15
Routed patches	0.818	0.001	0.9942	8.1×10^{-7}	49.2M	2.47

Table 1: Main comparison at 500 edits. Context oracle adds the gold memory snippet at test time, so it is best viewed as an upper bound for retrieval-style editing rather than a direct editor. Locality is preservation against the frozen base model.

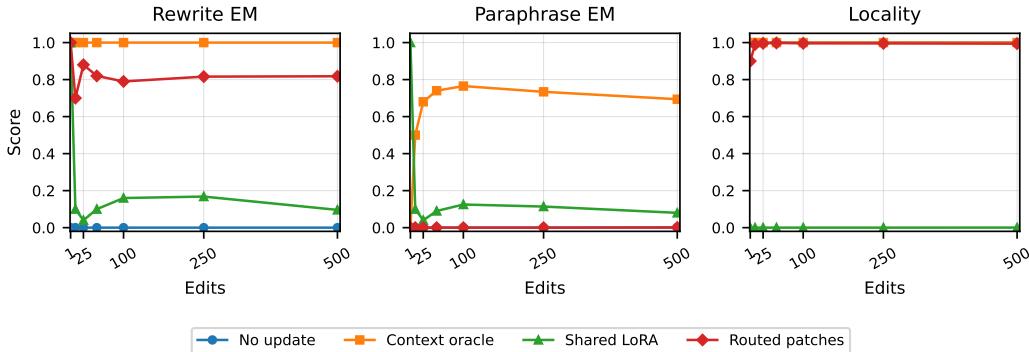


Figure 2: Continual learning curves for the four main methods. Routed patches remain stable on rewrite prompts and locality as the number of edits grows, while shared LoRA quickly hurts locality and never recovers.

Patch and LoRA training use only `train_prompts`. The paraphrase prompts are fully held out for evaluation. I evaluate after 1, 10, 25, 50, 100, 250, and 500 edits. All main results are from a single seed because full runs were expensive.

5.4 Results

Table 1 shows the 500-edit comparison. Figure 2 shows the continual curves. Two patterns stand out right away. First, routed patches are much more stable than a continually trained shared LoRA adapter. At 500 edits, routed patches keep rewrite exact match at 0.818 and locality at 0.9942, while shared LoRA drops to 0.096 rewrite exact match and almost zero locality. Second, the context oracle remains the strongest method overall on rewrite and paraphrase, so my method should not be read as the best overall editor. Instead, it should be read as a strong locality-preserving continual baseline.

The curves show that routed patches are fairly stable after the first few edits. From 25 edits onward, rewrite exact match stays in a narrow band between 0.79 and 0.82, and locality stays between 0.994 and 0.998. Shared LoRA behaves very differently. It keeps a single adapter for all edits, so new edits overwrite old behavior and unrelated prompts drift badly. This is visible in the near-zero locality curve.

One honest limitation is parameter growth. The patch bank stores one patch per edit, so the total learned parameter count grows linearly with the number of edits. At 500 edits this is 49.2M learned parameters, compared with 12.6M for shared LoRA. Still, the patch bank gets much better stability, which suggests that edit separation matters more than raw parameter count in this setting.

6 Analysis

The main weakness of the method is not patch quality. It is routing. Table 2 compares the normal routed system with an oracle router that always selects the correct patch by edit ID. At 500 edits, oracle routing raises paraphrase exact match from 0.001 to 0.407 while keeping perfect locality.

Method	Rewrite EM	Paraphrase EM	Locality
Oracle router patches	0.822	0.407	1.0000
Routed patches	0.818	0.001	0.9942

Table 2: Oracle-router ablation at 500 edits. The patch bank itself is much better than the deployed router on paraphrases.

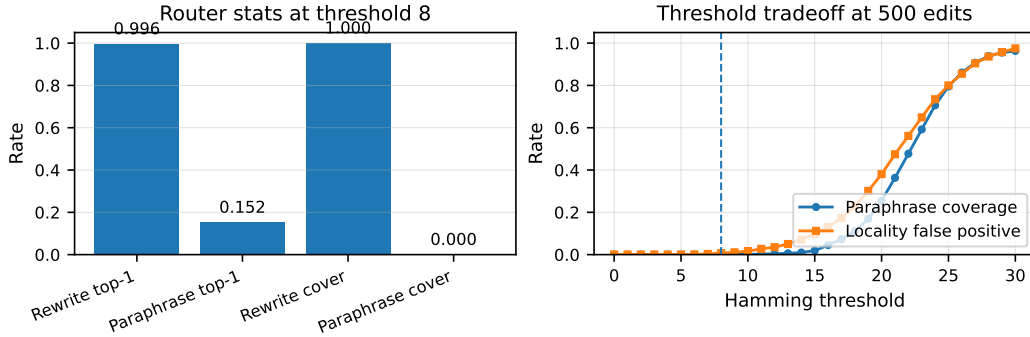


Figure 3: Router behavior at 500 edits. Left, rewrite prompts route correctly while paraphrases rarely match the right key. Right, relaxing the threshold helps paraphrases only a little and quickly hurts locality.

Rewrite exact match changes only slightly, from 0.818 to 0.822. This means the stored patches do generalize somewhat beyond the exact train prompt. The current router is the real bottleneck.

Figure 3 makes this router failure more concrete. On rewrite prompts, the router is almost perfect. Top-1 accuracy is 0.996 and coverage is 1.000. On paraphrases, top-1 accuracy falls to 0.152 and coverage falls to 0.000 because the nearest key is still usually too far from the threshold of 8. A simple threshold increase is not enough. A post-hoc sweep shows that raising the threshold to 20 would only raise paraphrase coverage to 0.255, but locality false positives would jump to 0.381. So the main issue is not just a strict threshold. It is that paraphrases and unrelated prompts overlap too much in the current hash space.

A second point is that strict exact match likely underrates rewrite quality. On rewrite prompts at 500 edits, routed patches get 0.818 strict exact match but 0.996 relaxed substring match. Many failures are expanded answers rather than wrong facts, such as predicting *Boston, Massachusetts* when the target is *Boston*, or *Dallas, Texas* when the target is *Dallas*. This means the patch often moves the model toward the right fact, but greedy generation adds extra words.

The locality errors are also informative. False positives are rare, only 36 out of 5,000 locality prompts. Among those 36 false positives, 25 come from *twin city* prompts. So the mistakes are not random. They cluster around one repeated relation pattern, which suggests that relation-level collisions in the router are a bigger problem than broad model damage.

Fact types and teaching new knowledge. The main CounterFact benchmark measures short factual QA, so the quantitative part of this report is still centered on direct fact recall. To check whether the method can teach truly new information, I also kept the small synthetic sanity set described in the Data section. One of those edits says `inventor(Project ZQ-67)=Arash Hamzehlou`. Because Project ZQ-67 is made up and the target answer is personal, this fact is very unlikely to come from pretraining. The patch system still learns to answer direct prompts about that project, which suggests a practical use case: adding new project names, launch dates, or user-specific facts after training instead of only flipping benchmark answers.

This also helps answer the question of downstream use. I did not run a full story-generation benchmark, so I do not claim a strong downstream result here. But the same mechanism should be useful in richer prompts, for example asking the model to write a short update that mentions when Project Orion launches or who invented Project ZQ-67. The best current proxy for that is paraphrase

behavior. The oracle-router result of 0.407 paraphrase exact match shows that the stored patch often carries the fact beyond the exact training wording. The main blocker for broader use is still routing, not the patch representation.

Finally, a concrete success and failure example help summarize the story. On the rewrite prompt *Italy is affiliated with the*, the router picks the correct patch and the model outputs *FIFA*. On a paraphrase with a long distractor prefix, the same edit gets a top-1 router distance of 24, above the threshold, so no patch fires and the answer is wrong. That is exactly the behavior shown by the oracle-router gap.

7 Conclusion

This project studied a simple continual editing method for a small language model. The main result is that a bank of hash-routed residual patches is much more stable than a continually trained shared LoRA adapter. After 500 sequential edits, routed patches keep rewrite exact match above 0.81 and locality near 0.994, while shared LoRA collapses on both measures. At the same time, the method still fails badly on paraphrases because the current SimHash router almost never activates on held-out rewrites. The small synthetic sanity set also suggests that the method can add genuinely new post-training facts, not only benchmark rewrites.

So the project ends with a clear positive result and a clear next step. Separating edits into small routed patches is a good idea for locality and continual retention. The next version should focus on a stronger router, for example by storing multiple keys per edit, adding contrastive router training, or learning a semantic retrieval space from paraphrases. It would also be useful to compare against direct editing methods like ROME or MEMIT on the same subset and to run more than one seed.

Like other editing methods, this approach could be misused to inject false claims. For that reason, I only used public benchmark data and I stored a record for each learned patch. In a real deployment, edit logging and patch-use detection would be important safety features.

Team contributions

This was a single-person project. Arash Hamzehlou implemented the code, ran the experiments, analyzed the outputs, and wrote the report.

References

- [1] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and Editing Factual Associations in GPT. *arXiv preprint arXiv:2202.05262*, 2022.
- [2] Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. Mass-Editing Memory in a Transformer. *arXiv preprint arXiv:2210.07229*, 2022.
- [3] Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D. Manning. Fast Model Editing at Scale. *arXiv preprint arXiv:2110.11309*, 2021.
- [4] Peng Wang, Zexi Li, Ningyu Zhang, Ziwen Xu, Yunzhi Yao, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. WISE: Rethinking the knowledge memory for lifelong model editing of large language models. *arXiv preprint arXiv:2405.14768*, 2024.
- [5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
- [6] Moses Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, 2002.
- [7] Marah Abdin et al. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. *arXiv preprint arXiv:2404.14219*, 2024.
- [8] Eric Mitchell, Charles Lin, Antoine Bosselut, Christopher D. Manning, and Chelsea Finn. Memory-Based Model Editing at Scale. In *International Conference on Machine Learning*, 2022.