Reverse-Engineering Memory in DreamerV3: From Sparse Representations to Functional Circuits

Jan Sobotka* **Auke Ijspeert** **Guillaume Bellegarda**
EPFL, Switzerland

Abstract

Understanding how reinforcement learning (RL) agents with recurrent neural network architectures encode and use memory remains an open question in the field of interpretability. In this work, we investigate these internal memory dynamics of DreamerV3, a state-of-the-art model-based deep RL agent. Our analysis reveals that DreamerV3 relies on sparse memory representations and on small internal subnetworks (circuits) to store and act on memory, with only a small subset of the original model parameters sufficient to control goal-directed behavior. We show that using a differentiable circuit extraction method, we can identify these subnetworks that retain full task performance with as little as 0.16% of the original parameters. Furthermore, we demonstrate that these sparse circuits emerge early in training and can retroactively improve undertrained models when applied as binary masks. Finally, we develop a gradient-based model editing approach that leverages these circuits for a reliable post hoc modification of the agent's behavior, achieving an average edit success rate of 90%. Our work demonstrates how sparse memory circuits provide a powerful lever for understanding and editing deep RL systems.

1 Introduction

Reinforcement learning agents make decisions over time by interacting with their environments, often in complex, dynamic settings. Understanding how these agents work internally is crucial for ensuring they are safe and efficient, especially in real-world applications such as autonomous driving or healthcare. Unlike most of machine learning (ML), where models map fixed inputs to outputs, RL involves sequences of actions where the data depends on the policy itself. This complexity is compounded when RL agents must remember past information to handle partial observability. While interpretability in (self-)supervised learning has advanced, interpretability in RL, especially for recurrent neural network (RNN) architectures, is still developing, a gap this study aims to address.

Specifically, we reverse-engineer how memory is encoded and used within the recurrent hidden state of the state-of-the-art agent, DreamerV3 [9]. Using interventions and circuit discovery techniques, we identify the specific mechanisms that guide its behavior. Our main contributions are as follows:

1. We introduce a new RL environment called MiniGrid-Switching-Memory, inspired by studies in neuroscience [17, 23], to study rule-based decision-making and memory.
2. By analyzing hidden states of recurrent RL agents, we find extremely sparse memory representations that reach optimal coding of past environment interactions.
3. We show that the recurrent RL agent, DreamerV3, encodes memory using small neural (sub)networks, which we call circuits [20], and which we can easily uncover.

*Correspondence to: jan.sobotka@epfl.ch

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

4. We propose a data-efficient gradient-based model editing technique that achieves a 90% average success rate in overwriting the goals of the trained RL agent.

2 Related work

Interpretability in RL. A key challenge in RL is solving partially observable tasks, which requires maintaining a memory of past events, often using recurrent neural networks (RNNs) [6, 11]. Techniques such as policy visualization [26], reward decomposition [12], probing [1, 14, 25], steering [14, 18], and quantitative measures [24] have been proposed to interpret RL agents. However, understanding the internal representations of recurrent RL agents remains an underexplored area.

Model editing. Model editing aims to modify a trained model without costly retraining, and has been primarily explored in natural language processing to update factual knowledge in transformers [15, 19]. In RL, such techniques could adapt agents to new reward specifications or correct undesirable behaviors, such as goal misgeneralization, where an agent optimizes for a proxy goal that fails to align with the designer's intent in new situations [13]. With this motivation in mind, we extend the "locate-and-edit" model-editing paradigm [7, 8, 15, 16] to the distinct challenges of RL agents with recurrent memory and sequential behavior. Our work is one of the first to demonstrate successful model editing in RL, and one of the first to show post hoc modifications of recurrent architectures.

3 Experiments

3.1 Environments

We use two partially observable MiniGrid [5] environments to test memory. Observations are egocentric 3×3 RGB images, and the agent receives a sparse reward upon task completion (subsection A.3).

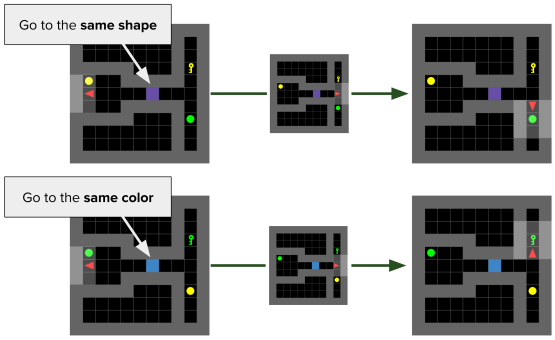

Figure 1: MiniGrid-Switching-Memory: A floor tile cue determines whether the agent should match the initial object's shape or color.

MiniGrid-Memory. This environment tests working memory. The agent starts by observing a green cue object (a key or a circle). It must then navigate a corridor to a T-junction and choose the path leading to the object that matches the initial cue. The cue's shape and location are randomized each episode to ensure reliance on memory.

MiniGrid-Switching-Memory. This environment extends MiniGrid-Memory with a task-switching component (Figure 1). The agent sees an object with a specific shape and color. Later, it encounters a colored floor tile that acts as a rule cue: a purple tile means "match by shape", while a blue tile means "match by color". This requires the agent to store a multi-featured memory and dynamically select one of its attributes based on new environment observations.

3.2 Reinforcement learning agent

DreamerV3. Our primary subject is DreamerV3 [9], a state-of-the-art model-based RL agent. It learns a world model with a GRU-based recurrent backbone [6] to predict future outcomes in a compact latent space. An actor-critic algorithm then learns a policy by training on trajectories "imagined" within this world model. A key feature is an autoencoding objective that forces the world model's hidden state to retain information needed to reconstruct observations. Our analysis focuses on the hidden states of this recurrent world model. For our study, we train the agent until it achieves a $> 95\%$ success rate on the task. For further details, please see subsection A.1.

We analyze DreamerV3's memory in three stages. First, we analyze its hidden states to understand memory representations. Second, we use circuit discovery to understand the internal mechanisms that produce these representations. Finally, we introduce a model-editing method to validate our understanding. Each provides a complementary perspective on the agent's memory functionality.

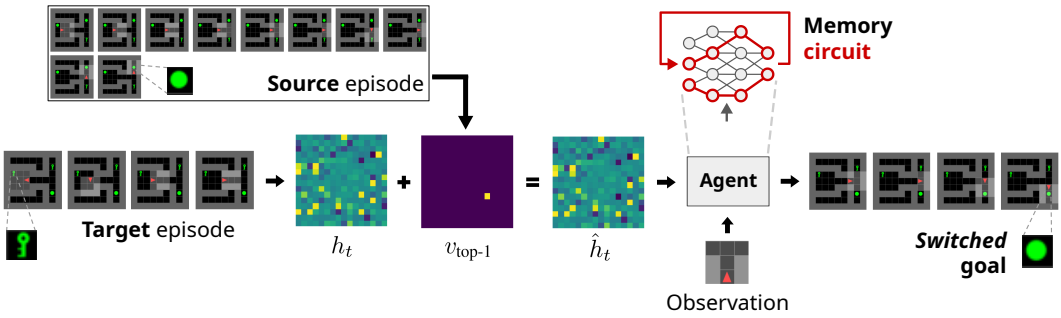

Figure 2: A successful memory intervention in MiniGrid-Memory, showing the internal memory circuit (red) and the hidden states before (h_t) and after ($\hat{h}_t$) applying the intervention patch $v_{\text{top-1}}$.

3.3 Memory intervention

We first locate the memory encoding within the agent's recurrent hidden state using activation patching [10], a causal intervention technique. Specifically, we run two episodes with contrastive goals (e.g., go to a circle vs. a key) and identify the minimal set of hidden state coordinates that, when patched from one (source) episode to the other (target) episode at a critical timestep, cause the agent to switch its goal. A full description of the procedure is in subsection A.4.

Interestingly, in MiniGrid-Memory, we find that modifying just a single coordinate ($k = 1$) is sufficient to reliably redirect the agent's behavior (Figure 2), indicating a highly sparse memory encoding.[2] In fact, this reaches the optimal single-bit encoding needed to distinguish between the two goal states. This contrasts with a baseline recurrent model-free proximal policy optimization agent [22] trained solely for policy and value prediction, which required patching 10% of its 256-dimensional hidden state[3]. We hypothesize that this sparsity difference stems from DreamerV3's auxiliary autoencoding objective, which forces the network to compress memory into a smaller subspace while dedicating most of the hidden state to encoding current visual observations.

3.4 Circuit discovery

Hidden-state interventions reveal *where* memory is encoded, but not the mechanism that writes and maintains it. To uncover this mechanism, we identify the minimal subnetwork, or *circuit*, responsible for memory-based behavior. We adapt a differentiable binary weight masking technique similar to that used by [2, 4] to find a sparse mask over the agent's parameters. We optimize this mask via behavior cloning on a dataset of pre-collected trajectories of the original (unmasked) agent, using a straight-through estimator and strong L1 regularization to encourage sparsity (details in subsection A.5).

Table 1: Circuit discovery results. The success rate is an average of 100 evaluation episodes.

Agent	MEMORY		SWITCHING-MEMORY	
	Success rate	Sparsity	Success rate	Sparsity
Original	96.9%	$< 0.001\%$	98.3%	$< 0.001\%$
Masked	96.9%	99.82%	98.3%	99.84%

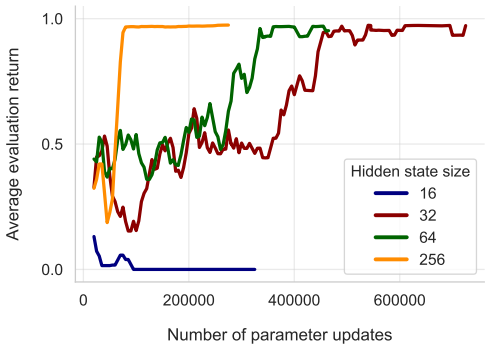

Figure 3: Smaller models require longer training. Evaluation return averaged from 10 episodes.

Using this procedure on the DreamerV3 agent trained in the MiniGrid-Memory environment, we find that the agent can retain its original performance using only 0.16% of its parameters (Table 1, MEMORY). This reduces the number of active parameters from $\pm$5M to around 8K. While one might argue that the original model was overparameterized, training a smaller model from scratch either fails or requires significantly longer training (Figure 3), demonstrating the efficiency of discovering sparse circuits within larger models using the differentiable weight masking.

[2]This is consistent across 50 episodes where intervening on the same coordinate reliably redirects the agent.
[3]Details on this agent's architecture are provided in subsection A.2.

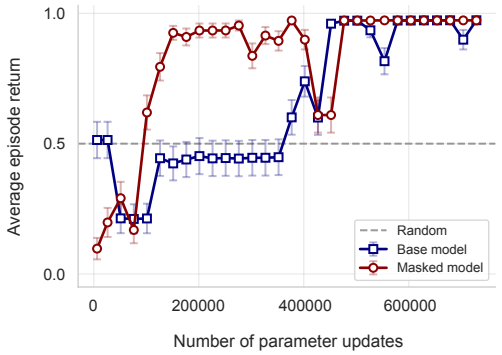

Figure 4: Circuit development in MiniGrid-Memory. Masking early checkpoints with the final circuit mask dramatically improves performance. Averages over 50 episodes.

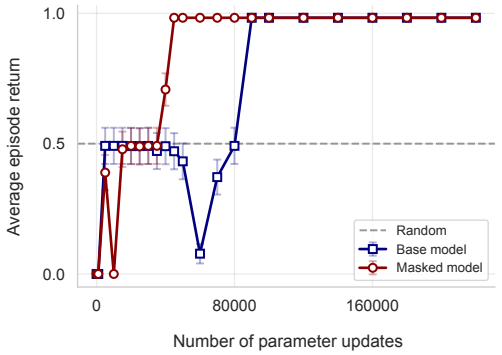

Figure 5: Circuit development in MiniGrid-Switching-Memory, showing similar performance gains on early checkpoints after masking. Averages over 50 episodes.

3.5 Circuit development

Studying the memory circuit's development reveals an interesting phenomenon. If we apply the binary weight mask (subsection 3.4) from a fully trained agent to its earlier, poorly performing agent checkpoints, their task performance jumps from near random to almost 100% success (Figures 4 and 5). This suggests that a functional, low-dimensional memory circuit emerges early in training, and much of the subsequent training is dedicated to "cleaning up" parameters outside this core circuit.

3.6 Model editing

Our findings suggest that behavior is governed by a small, identifiable circuit. This raises the possibility of targeted model editing to alter behavior, for example, to correct goal misgeneralization [13]. We develop a gradient-based method to do so. The core idea is to retrain only the discovered circuit parameters to encode the memory of an observed goal as if it had seen the opposite goal. We use a small, contrastive dataset (1–3 episode pairs) to create target hidden states where the key memory-encoding coordinates are patched as done in subsection 3.3. We then fine-tune the circuit's parameters with L2 loss to produce these patched representations. This one-time optimization of only a few parameters permanently alters the agent's goal-directed policy. The complete procedure is detailed in subsection A.6.

Table 2: Success rate of reaching the opposite goal after we apply the model editing to the original and masked agents. Evaluation on 100 testing episodes.

Model editing target	MEMORY	SWITCHING-MEMORY
Original agent	33.1%	86.8%
Masked agent	87.4%	92.1%

As shown in Table 2, this method is highly effective, achieving an 87.4% and 92.1% edit success rate on our two tasks. Crucially, attempting to edit the full model without first identifying the circuit is far less effective (e.g., 33.1% vs 87.4% on MiniGrid-Memory), demonstrating the importance of first locating the low-dimensional and causally robust target subnetwork for model editing.

4 Conclusion

In this work, we reverse-engineered the memory mechanisms of a state-of-the-art recurrent RL agent. Our findings provide both a processing and a developmental account of its behavior. We found that memory is not distributed but highly localized, implemented by sparse neural circuits that comprise just 0.16% of the model parameters. These functional circuits emerge remarkably early in training, well before the overall performance on the task stabilizes. This mechanistic insight is not just descriptive; it enables a practical application. By targeting the identified memory circuit, we developed a data-efficient model editing technique to reliably overwrite the agent's learned goals. Our results demonstrate that even complex deep RL agents can develop modular and interpretable cognitive strategies, opening a promising path for reverse-engineering and aligning intelligent systems.

Acknowledgments

Jan Sobotka was supported by the Bakala Foundation during his studies at EPFL.

References

[1] G. Alain and Y. Bengio. Understanding intermediate layers using linear classifier probes, 2017. URL https://openreview.net/forum?id=ryF7rTqgl.

[2] D. Bayazit, N. Foroutan, Z. Chen, G. Weiss, and A. Bosselut. Discovering knowledge-critical subnetworks in pretrained language models. In Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 6549–6583, Miami, Florida, USA, Nov. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.376. URL https://aclanthology.org/2024.emnlp-main.376/.

[3] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. URL https://arxiv.org/abs/1308.3432.

[4] S. Cao, V. Sanh, and A. Rush. Low-complexity probing via finding subnetworks. In K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 960–966, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.74. URL https://aclanthology.org/2021.naacl-main.74/.

[5] M. Chevalier-Boisvert, B. Dai, M. Towers, R. Perez-Vicente, L. Willems, S. Lahlou, S. Pal, P. S. Castro, and J. Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. In *Advances in Neural Information Processing Systems 36, New Orleans, LA, USA*, December 2023.

[6] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[7] Z. Dong, X. Shen, and R. Xia. Memit-merge: Addressing memit's key-value conflicts in same-subject batch editing for llms, 2025. URL https://arxiv.org/abs/2502.07322.

[8] A. Gupta, D. Sajnani, and G. Anumanchipalli. A unified framework for model editing. In Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 15403–15418, Miami, Florida, USA, Nov. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.903. URL https://aclanthology.org/2024.findings-emnlp.903/.

[9] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[10] S. Heimersheim and N. Nanda. How to use and interpret activation patching. *arXiv preprint arXiv:2404.15255*, 2024.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[12] Z. Juozapaitis, A. Koul, A. Fern, M. Erwig, and F. Doshi-Velez. Explainable reinforcement learning via reward decomposition. *in proceedings at the International Joint Conference on Artificial Intelligence. A Workshop on Explainable Artificial Intelligence.*, 2019.

[13] L. L. D. Langosco, J. Koch, L. D. Sharkey, J. Pfau, and D. Krueger. Goal misgeneralization in deep reinforcement learning. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 12004–12019. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/langosco22a.html.

[14] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. Emergent world representations: Exploring a sequence model trained on a synthetic task. In *The Eleventh*

International Conference on Learning Representations, 2023. URL https://openreview.net/forum?id=DeG07_TcZvT.

[15] K. Meng, D. Bau, A. Andonian, and Y. Belinkov. Locating and editing factual associations in gpt. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 17359–17372. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/6f1d43d5a82a37e89b0665b33bf3a182-Paper-Conference.pdf.

[16] K. Meng, A. S. Sharma, A. J. Andonian, Y. Belinkov, and D. Bau. Mass-editing memory in a transformer. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=MkbcAHIYgyS.

[17] E. K. Miller and J. D. Cohen. An integrative theory of prefrontal cortex function. *Annu Rev Neurosci*, 24:167–202, 2001.

[18] U. Mini, P. Grietzer, M. Sharma, A. Meek, M. MacDiarmid, and A. M. Turner. Understanding and controlling a maze-solving policy network. *arXiv preprint arXiv:2310.08043*, 2023.

[19] E. Mitchell, C. Lin, A. Bosselut, C. Finn, and C. D. Manning. Memory-based model editing at scale. In *Advances in Neural Information Processing Systems*, 2022.

[20] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. Zoom in: An introduction to circuits. *Distill*, 2020. doi: 10.23915/distill.00024.001. https://distill.pub/2020/circuits/zoom-in.

[21] M. Pleines, M. Pallasch, F. Zimmer, and M. Preuss. Memory gym: Partially observable challenges to memory-based agents. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=jHc8dCx6DDr.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[23] J. D. Wallis, K. C. Anderson, and E. K. Miller. Single neurons in prefrontal cortex encode abstract rules. *Nature*, 411(6840):953–956, Jun 2001. ISSN 1476-4687. doi: 10.1038/35082081. URL https://doi.org/10.1038/35082081.

[24] H. Wang, E. Miahi, M. White, M. C. Machado, Z. Abbas, R. Kumaraswamy, V. Liu, and A. White. Investigating the properties of neural network representations in reinforcement learning. *Artificial Intelligence*, 330:104100, 2024. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2024.104100. URL https://www.sciencedirect.com/science/article/pii/S0004370224000365.

[25] E. Wijmans, M. Savva, I. Essa, S. Lee, A. S. Morcos, and D. Batra. Emergence of maps in the memories of blind navigation agents. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=lTt4KjHSsyl.

[26] T. Zahavy, N. B. Zrihem, and S. Mannor. Graying the black box: understanding dqns. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1899–1908. JMLR.org, 2016.

A Technical Appendices and Supplementary Material

A.1 DreamerV3 agent

DreamerV3 [9] is a model-based RL approach that employs an RNN as a backbone of its internal world model. In this architecture, environment observations are encoded into latent states using an autoencoder, which is trained via an observation reconstruction (autoencoding) loss. This loss ensures that all observation information passes through the hidden states of the world model. The world model itself is trained to predict rewards, episode continuation signals, and subsequent latent states based on previous actions. Our analysis of DreamerV3 in section 3 focuses on the hidden states of the RNN backbone (GRU) [6] of this world model.

Policy optimization (actor-critic) is carried out in simulated trajectories, which are generated using replayed latent states and sampled actor actions. The critic network estimates the sum of expected future rewards, while the actor optimizes behavior by maximizing these predicted returns. Since learning occurs in abstract latent sequences, the method is highly generalizable to both low- and high-dimensional observations under full or partial observability. For additional algorithmic details and preprocessing, please refer to the original work [9].

For all our experiments except those reported in Figures 3 and 4, we used a hidden state size of 256. For Figure 3, we varied the size of the hidden state to investigate the relationship between model parameter count and required training steps. Note that while the hidden state size is not the only determinant of the number of model parameters, it is one of the main control knobs. This is because many other components that read or write to the hidden state, such as the encoder or the actor and value networks, adjust the sizes of their weight matrices accordingly.

The source code for all experiments, including the hyperparameters of the environments and the agents, is available at `https://github.com/Johnny1188/rl-memory.git`.

A.2 Baseline recurrent model-free agent

The baseline RL agent used for the comparison in subsection 3.3 is based on the proximal policy optimization (PPO) algorithm [22] with an LSTM backbone [11] that is trained using the truncated backpropagation through time. At each timestep, the current observation is first encoded by three convolutional layers, then passed through the LSTM, one fully connected layer, and finally split into two streams. One stream computes the value function, and the other computes the policy. Both streams consist of a single fully connected layer. In subsection 3.3, we analyze the hidden states of the LSTM backbone that are updated at each timestep.

We use the implementation and hyperparameters from [21], where the authors used this agent for evaluating memory-based architectures on RL tasks.

A.3 Environment details

MiniGrid-Memory. MiniGrid-Memory is a partially observable 9×9 grid-world environment designed to test an agent's ability to utilize memory. The task requires the agent to recall information seen in earlier parts of an episode, which makes it suitable for evaluating recurrent and memory-based agents. It was initially introduced by the Farama Foundation[4], but we use the modified version by [21].

In this environment, the agent begins close to a room where it observes a green object, a cue (a key or a circle), and must navigate through a narrow hallway leading to a split. At each end of the split, there is an object, one of which matches the object seen at the start. The agent's task is to recall the initial object and navigate to the matching object. The action space includes discrete actions to move forward and turn left and right. Observations are RGB images of a local 3×3 grid neighborhood of the agent. At the end of each episode, the agent receives a reward of $1 - 0.9 \cdot (\frac{\text{timestep}}{\text{max timesteps}})$ for correctly identifying the matching object, and no reward for failure (i.e., the agent reaches the wrong nonmatching object). Episodes end upon success (i.e., nonzero reward), failure, or timeout of 150 timesteps.

[4] `https://minigrid.farama.org/environments/minigrid/MemoryEnv/`

MiniGrid-Switching-Memory. MiniGrid-Switching-Memory is a more challenging version of MiniGrid-Memory, but with the same 9×9 grid-world structure. In this environment, the agent needs to remember the shape of the object (circle or key), as well as its color (green or yellow), and then has to recognize whether it should go to an object with a matching shape or to an object with a matching color based on another cue in the environment. This cue of the *goal type* has the form of a floor tile in the middle of the narrow corridor between the starting room and the final split (junction). If this tile is purple, the agent should go to the object of the same shape, but if the tile is blue, it should go to the object of the same color.

All object colors and shapes, as well as the goal types, are completely randomized between different episodes so as to leave minimal spurious correlation that the agent could learn. This also holds for MiniGrid-Memory, where the goal shapes are selected randomly in each episode.

A.4 Memory intervention procedure

To locate memory in the hidden state, we use the following procedure:

1. We collect hidden states from two episodes with two different (contrastive) goals. In the MiniGrid-Memory environment, this is an episode where the agent is cued with a circle and another where it is cued with a key.
2. We extract the hidden state at an important (critical) timestep, such as when the agent is in the narrow corridor before the decision split.
3. We identify the memorized cue in the hidden state by locating the top-k coordinates (indices) with the highest absolute difference between the two hidden states.
4. We patch the values at these top-k coordinates from the hidden state of the source episode into the hidden state of the target episode.
5. We find the minimal k for a successful intervention, where patching the top-k coordinates redirects the agent in the target episode to the opposite goal, but patching the top-$(k-1)$ does not.

A.5 Circuit discovery procedure

We approach circuit discovery using differentiable binary weight masking with the following steps:

1. Using the pre-trained agent, we collect an offline dataset of 50,000 environment steps.
2. For each parameter $\theta_i \in \mathbb{R}$ of the trained agent, we initialize a new mask parameter $m_i \in \mathbb{R}$.
3. We freeze the base parameters θ_i and optimize the masks m_i using a behavior cloning on the pre-collected dataset, along with a strong L1 regularization. We use the straight-through gradient estimator [3], where we binarize the mask in a way that preserves differentiability and run all forward passes of the agent with the (sparse) masked parameters $\hat{\theta}_i$:

$$\hat{\theta}_i = \theta_i \odot \big(\mathrm{sg}(\tilde{m}_i - \sigma(m_i)) + \sigma(m_i)\big). \tag{1}$$

   In Equation 1, σ is the sigmoid function, $\mathrm{sg}(\,\cdot\,)$ is the stop-gradient operator, and $\tilde{m}_i = \mathbb{1}[\sigma(m_i) > 0.5]$ is the binary mask.
4. After mask optimization is complete, we use only the masked agent's parameters $\hat{\theta}_i = \theta_i \odot \tilde{m}_i$, effectively sparsifying all the weight matrices of the agent.

A.6 Model editing procedure

Our gradient-based model editing method consists of these steps:

1. First, we apply the circuit discovery procedure (subsection A.5) to obtain the binary weight mask and the masked agent's parameters $\hat{\theta} \in \mathbb{R}^d$, where $d \in \mathbb{N}$ is the number of model's parameters.
2. We then apply the intervention procedure from Section 3.3 on this masked agent to locate the hidden-state coordinates $I_h = \big\{i_k \in \mathbb{N}\big\}_{k=1}^{K}$ where memory is stored by the memory

circuit. In our experiments, we use $K = 1$ (MiniGrid-Memory) and $K = 5$ (MiniGrid-Switching-Memory).

3. We collect M pairs of contrastive episodes as data for optimization. In MiniGrid-Memory, these are pairs of episodes in which one episode has a circle as the goal, and another episode has a key as the goal. In MiniGrid-Switching-Memory, the contrast between the two episodes is in terms of the goal type, i.e., whether the agent should go to an object with a matching shape or a matching color. We use $M = 1$ and $M = 3$ pairs of episodes for MiniGrid-Memory and MiniGrid-Switching-Memory, respectively. This balances data efficiency and the ability of the method to change the agent's behavior.
4. Finally, we perform the model editing optimization procedure. Intuitively, we want the model to encode the memory of an observed goal as it would if it had seen the opposite goal. For example, in the MiniGrid-Memory environment, we want the agent to encode the memory of a circle when it sees a key as the cue object, because if it did, the actor network of the agent that is reading the memory from the hidden state would steer the agent to a circle. We can express this intuition and objective more formally as follows. Let $o_t^{(m,l)}$ be the image observation at timestep t of l-th episode in the m-th contrastive episode pair, $h_{t-1}^{(m,l)}$ be the hidden state from the same episode but from the previous timestep, and let $\bar{h}_t^{(m,l)} \in \mathbb{R}^{d_h}$ be the target hidden state (target memory encoding) of dimension $d_h \in \mathbb{N}$, where we flip the memory coordinates between the two contrastive episodes as $\bar{h}_{t,i}^{(m,l)} := h_{t,i}^{(m,\,(l \bmod 2)+1)}$ if $i \in I_h$ else $h_{t,i}^{(m,l)}$. Using this setup, we minimize the following loss function by performing a gradient descent[5] on the masked agent's parameters $\hat{\theta}$:

$$\mathcal{L}(\hat{\theta}) := \frac{1}{M} \sum_{m=1}^{M} \sum_{l=1}^{2} ||f(o_t^{(m,l)}, h_{t-1}^{(m,l)}, \hat{\theta}) - \bar{h}_t^{(m,l)}||_2^2. \tag{2}$$

   In equation 2 above, f represents the agent's neural network performing the hidden state update, and t is chosen to be the timestep where the goal object (MiniGrid-Memory) or the goal type (MiniGrid-Switching-Memory) is observed by the agent.

Notice that as long as we can obtain a contrastive pair of episodes between which we want to flip the agent's behavior, this model editing method is applicable to any task where a certain memory is encoded at a single timestep, but could naturally be extended to multi-timestep encoding. Also note that the optimization is a one-time procedure, i.e., it does not need to be performed for each new episode as the memory intervention from Section 3.3 does. Furthermore, we can measure the *edit success rate* of the model editing by the average success rate of the agent in reaching the opposite (modified) goal than for which it was originally trained.

[5] 1,000 steps for MiniGrid-Memory and 10,000 steps for MiniGrid-Switching-Memory.