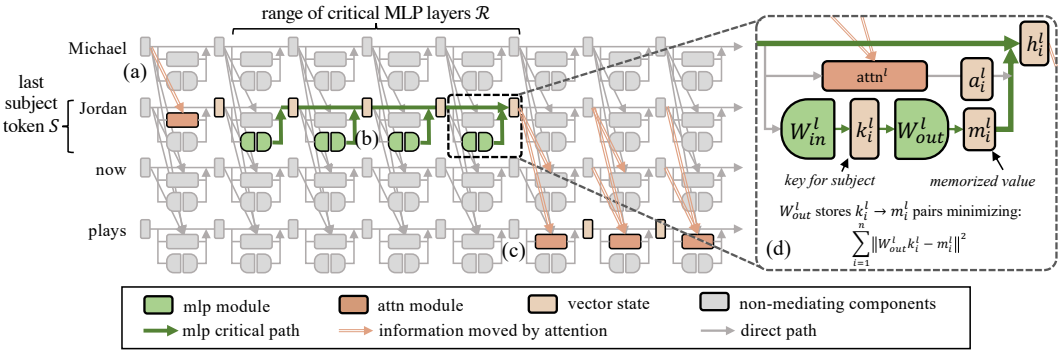

Figure 2: **MEMIT modifies transformer parameters on the critical path of MLP-mediated factual recall.** We edit stored associations based on observed patterns of causal mediation: (a) first, the early-layer attention modules gather subject names into vector representations at the last subject token S. (b) Then MLPs at layers $l \in \mathcal{R}$ read these encodings and add memories to the residual stream. (c) Those hidden states are read by attention to produce the output. (d) MEMIT edits memories by storing vector associations in the critical MLPs.

confirm that GPT-J has a concentration of mediating states h_i^l; moreover, they highlight a mediating causal role for a range of MLP modules, which can be seen as a large gap between the effect of single states (purple bars in Figure 3) and the effects with MLP severed (green bars); this gap diminishes after layer 8. Unlike Meng et al. (2022) who use this test to identify a single edit layer, we select the whole range of critical MLP layers $l \in \mathcal{R}$. For GPT-J, we have $\mathcal{R} = \{3, 4, 5, 6, 7, 8\}$.

Given that a *range* of MLPs play a joint mediating role in recalling facts, we ask: what is the role of *one* MLP in storing a memory? Each token state in a transformer is part of the residual stream that all attention and MLP modules read from and write to (Elhage et al., 2021). Unrolling Eqn. 2 for $h_i^L = h_{[S]}^L(p_i)$:

$$h_i^L = h_i^0 + \sum_{l=1}^{L} a_i^l + \sum_{l=1}^{L} m_i^l. \quad (6)$$

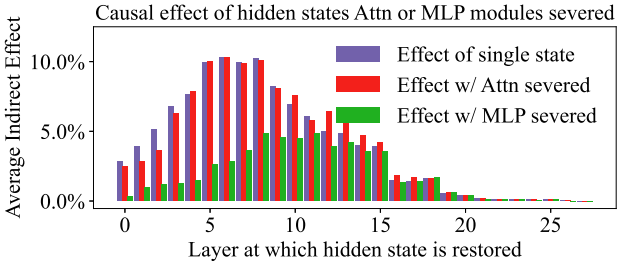

Figure 3: A critical mediating role for mid-layer MLPs.

Eqn. 6 highlights that each individual MLP contributes by *adding* to the memory at h_i^L (Figure 2b), which is later read by last-token attention modules (Figure 2c). Therefore, when writing new memories into G, we can spread the desired changes across all the critical layers m_i^l for $l \in \mathcal{R}$.

4.2 Batch Update for a Single Linear Associative Memory

In each individual layer l, we wish to store a large batch of $u \gg 1$ memories. This section derives an optimal single-layer update that minimizes the squared error of memorized associations, assuming that the layer contains previously-stored memories that should be preserved. We denote $W_0 \triangleq W_{out}^l$ (Eqn. 4, Figure 2) and analyze it as a linear associative memory (Kohonen, 1972; Anderson, 1972) that associates a set of input keys $k_i \triangleq k_i^l$ (encoding subjects) to corresponding memory values $m_i \triangleq m_i^l$ (encoding memorized properties) with minimal squared error:

$$W_0 \triangleq \underset{\hat{W}}{\operatorname{argmin}} \sum_{i=1}^{n} \left\| \hat{W} k_i - m_i \right\|^2. \quad (7)$$

If we stack keys and memories as matrices $K_0 = [k_1 \mid k_2 \mid \cdots \mid k_n]$ and $M_0 = [m_1 \mid m_2 \mid \cdots \mid m_n]$, then Eqn. 7 can be optimized by solving the normal equation (Strang, 1993, Chapter 4):

$$W_0 K_0 K_0^T = M_0 K_0^T. \quad (8)$$

Suppose that pre-training sets a transformer MLP's weights to the optimal solution W_0 as defined in Eqn. 8. Our goal is to update W_0 with some small change Δ that produces a new matrix W_1 with

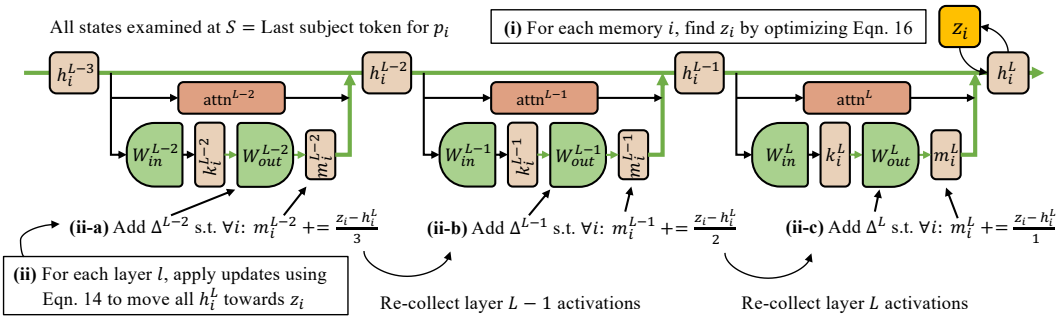

Figure 4: **The MEMIT update**. We first (i) replace h_i^l with the vector z_i and optimize Eqn. 16 so that it conveys the new memory. Then, after all z_i are calculated we (ii) iteratively insert a fraction of the residuals for all z_i over the range of critical MLP modules, executing each layer's update by applying Eqn. 14. Because changing one layer will affect activations of downstream modules, we recollect activations after each iteration.

a set of additional associations. Unlike Meng et al. (2022), we cannot solve our problem with a constraint that adds only a single new association, so we define an expanded objective:

$$W_1 \triangleq \underset{\hat{W}}{\operatorname{argmin}} \left(\sum_{i=1}^{n} \left\| \hat{W} k_i - m_i \right\|^2 + \sum_{i=n+1}^{n+u} \left\| \hat{W} k_i - m_i \right\|^2 \right). \tag{9}$$

We can solve Eqn. 9 by again applying the normal equation, now written in block form:

$$W_1 \begin{bmatrix} K_0 & K_1 \end{bmatrix} \begin{bmatrix} K_0 & K_1 \end{bmatrix}^T = \begin{bmatrix} M_0 & M_1 \end{bmatrix} \begin{bmatrix} K_0 & K_1 \end{bmatrix}^T \tag{10}$$

$$\text{which expands to:} \quad (W_0 + \Delta)(K_0 K_0^T + K_1 K_1^T) = M_0 K_0^T + M_1 K_1^T \tag{11}$$

$$W_0 K_0 K_0^T + W_0 K_1 K_1^T + \Delta K_0 K_0^T + \Delta K_1 K_1^T = M_0 K_0^T + M_1 K_1^T \tag{12}$$

$$\text{subtracting Eqn. 8 from Eqn. 12:} \quad \Delta(K_0 K_0^T + K_1 K_1^T) = M_1 K_1^T - W_0 K_1 K_1^T. \tag{13}$$

A succinct solution can be written by defining two additional quantities: $C_0 \triangleq K_0 K_0^T$, a constant proportional to the uncentered covariance of the pre-existing keys, and $R \triangleq M_1 - W_0 K_1$, the residual error of the new associations when evaluated on old weights W_0. Then Eqn. 13 can be simplified as:

$$\Delta = R K_1^T (C_0 + K_1 K_1^T)^{-1}. \tag{14}$$

Since pretraining is opaque, we do not have access to K_0 or M_0. Fortunately, computing Eqn. 14 only requires an aggregate statistic C_0 over the previously stored keys. We assume that the set of previously memorized keys can be modeled as a random sample of inputs, so that we can compute

$$C_0 = \lambda \cdot \mathbb{E}_k \left[k k^T \right] \tag{15}$$

by estimating $\mathbb{E}_k \left[k k^T \right]$, an uncentered covariance statistic collected using an empirical sample of vector inputs to the layer. We must also select λ, a hyperparameter that balances the weighting of new v.s. old associations; a typical value is $\lambda = 1.5 \times 10^4$.

4.3 Updating multiple layers

We now define the overall update algorithm (Figure 4). Inspired by the observation that robustness is improved when parameter change magnitudes are minimized (Zhu et al., 2020), we spread updates evenly over the range of mediating layers $\mathcal{R}$. We define a target layer $L \triangleq \max(\mathcal{R})$ at the end of the mediating layers, at which the new memories should be fully represented. Then, for each edit $(s_i, r_i, o_i) \in \mathcal{E}$, we (i) compute a hidden vector z_i to replace h_i^L such that adding $\delta_i \triangleq z_i - h_i^L$ to the hidden state at layer L and token T will completely convey the new memory. Finally, one layer at a time, we (ii) modify the MLP at layer l, so that it contributes an approximately-equal portion of the change δ_i for each memory i.

(i) Computing z_i. For the ith memory, we first compute a vector z_i that would encode the association (s_i, r_i, o_i) if it were to replace h_i^L at layer L at token S. We find $z_i = h_i^L + \delta_i$ by optimizing the residual vector δ_i using gradient descent:

$$z_i = h_i^L + \underset{\delta_i}{\operatorname{argmin}} \frac{1}{P} \sum_{j=1}^{P} -\log \mathbb{P}_{G(h_i^L \mathrel{+}= \delta_i)} \left[o_i \mid x_j \oplus p(s_i, r_i) \right]. \tag{16}$$

In words, we optimize δ_i to maximize the model's prediction of the desired object o_i, given a set of factual prompts $\{x_j \oplus p(s_i, r_i)\}$ that concatenate random prefixes x_j to a templated prompt to aid generalization across contexts. $G(h_i^L \mathrel{+}= \delta_i)$ indicates that we modify the transformer execution by substituting the modified hidden state z_i for h_i^L; this is called "hooking" in popular ML libraries.

(ii) Spreading $z_i - h_i^L$ over layers. We seek delta matrices Δ^l such that:

$$\text{setting } \hat{W}_{out}^l := W_{out}^l + \Delta^l \text{ for all } l \in \mathcal{R} \text{ optimizes } \min_{\{\Delta^l\}} \sum_i \left\| z_i - \hat{h}_i^L \right\|^2, \tag{17}$$

$$\text{where } \hat{h}_i^L = h_i^0 + \sum_{l=1}^{L} a_i^l + \sum_{l=1}^{L} \hat{W}_{out}^l \, \sigma\left(W_{in}^l \gamma\left(h_t^{l-1}\right)\right). \tag{18}$$

Because edits to any layer will influence all following layers' activations, we calculate Δ^l iteratively in ascending layer order (Figure 4ii-a,b,c). To compute each individual Δ^l, we need the corresponding keys $K^l = [k_1^l \mid \cdots \mid k_n^l]$ and memories $M^l = [m_1^l \mid \cdots \mid m_n^l]$ to insert using Eqn. 14. Each key k_i^l is computed as the input to W_{out}^l at each layer l (Figure 2d):

$$k_i^l = \frac{1}{P} \sum_{j=1}^{P} k(x_j + s_i), \text{ where } k(x) = \sigma\left(W_{in}^l \, \gamma\left(h_i^{l-1}(x)\right)\right). \tag{19}$$

m_i^l is then computed as the sum of its current value and a fraction of the remaining top-level residual:

$$m_i^l = W_{out} k_i^l + r_i^l \text{ where } r_i^l \text{ is the residual given by } \frac{z_i - h_i^L}{L - l + 1}, \tag{20}$$

where the denominator of r_i spreads the residual out evenly. Algorithm 1 summarizes MEMIT, and additional implementation details are offered in Appendix B.

Algorithm 1: The MEMIT Algorithm

Data: Requested edits $\mathcal{E} = \{(s_i, r_i, o_i)\}$, generator G, layers to edit $\mathcal{S}$, covariances C^l
Result: Modified generator containing edits from $\mathcal{E}$

for $s_i, r_i, o_i \in \mathcal{E}$ **do** // Compute target z_i vectors for every memory i
  optimize $\delta_i \leftarrow \text{argmin}_{\delta_i} \frac{1}{P} \sum_{j=1}^{P} -\log \mathbb{P}_{G(h_i^L \mathrel{+}= \delta_i)}\left[o_i \mid x_j \oplus p(s_i, r_i)\right]$ (Eqn. 16)
  $z_i \leftarrow h_i^L + \delta_i$
end
for $l \in \mathcal{R}$ **do** // Perform update: spread changes over layers
  $h_i^l \leftarrow h_i^{l-1} + a_i^l + m_i^l$ (Eqn. 2) // Run layer l with updated weights
  for $s_i, r_i, o_i \in \mathcal{E}$ **do**
    $k_i^l \leftarrow k_i^l = \frac{1}{P} \sum_{j=1}^{P} k(x_j + s_i)$ (Eqn. 19)
    $r_i^l \leftarrow \frac{z_i - h_i^L}{L-l+1}$ (Eqn. 20) // Distribute residual over remaining layers
  end
  $K^l \leftarrow [k_i^{l_1}, ..., k_i^L]$
  $R^l \leftarrow [r_i^{l_1}, ..., r_i^L]$
  $\Delta^l \leftarrow R^l {K^l}^T (C^l + K^l {K^l}^T)^{-1}$ (Eqn. 14)
  $W^l \leftarrow W^l + \Delta^l$ // Update layer l MLP weights in model
end

5 Experiments

5.1 Models and Baselines

We run experiments on two autoregressive LLMs: GPT-J (6B) and GPT-NeoX (20B). For baselines, we first compare with a naive fine-tuning approach that uses weight decay to prevent forgetfulness (**FT-W**). Next, we experiment with **MEND**, a hypernetwork-based model editing approach that edits multiple facts at the same time (Mitchell et al., 2021). Finally, we run a sequential version of **ROME** (Meng et al., 2022): a direct model editing method that iteratively updates one fact at a time. The recent SERAC model editor (Mitchell et al., 2022) does not yet have public code, so we cannot compare with it at this time. See Appendix B for implementation details.