Stealing Part of a Production Language Model

Nicholas Carlini [1] Daniel Paleka [2] Krishnamurthy (Dj) Dvijotham [1] Thomas Steinke [1] Jonathan Hayase [3] A. Feder Cooper [1] Katherine Lee [1] Matthew Jagielski [1] Milad Nasr [1] Arthur Conmy [1] Eric Wallace [4] David Rolnick [5] Florian Tramèr [2]

Abstract

We introduce the first model-stealing attack that extracts precise, nontrivial information from black-box production language models like OpenAI's ChatGPT or Google's PaLM-2. Specifically, our attack recovers the *embedding projection layer* (up to symmetries) of a transformer model, given typical API access. For under \$20 USD, our attack extracts the entire projection matrix of OpenAI's `ada` and `babbage` language models. We thereby confirm, for the first time, that these black-box models have a hidden dimension of 1024 and 2048, respectively. We also recover the exact hidden dimension size of the `gpt-3.5-turbo` model, and estimate it would cost under \$2,000 in queries to recover the entire projection matrix. We conclude with potential defenses and mitigations, and discuss the implications of possible future work that could extend our attack.

1. Introduction

Little is publicly known about the inner workings of today's most popular large language models, such as GPT-4, Claude 2, or Gemini. The GPT-4 technical report states it "contains no [...] details about the architecture (including model size), hardware, training compute, dataset construction, training method, or similar" (OpenAI et al., 2023). Similarly, the PaLM-2 paper states that "details of [the] model size and architecture are withheld from external publication" (Anil et al., 2023). This secrecy is often ascribed to "the competitive landscape" (because these models are expensive to train) and the "safety implications of large-scale models" (OpenAI et al., 2023) (because it is easier to attack models when more information is available). Nevertheless, while these models' weights and internal details are not publicly accessible, the models themselves are exposed via APIs.

In this paper we ask: *how much information can an adversary learn about a production language model by making queries to its API?* This is the question studied by the field of *model stealing* (Tramèr et al., 2016): the ability of an adversary to extract model weights by making queries its API.

Contributions. We introduce an attack that can be applied to black-box language models, and allows us to recover the complete *embedding projection layer* of a transformer language model. Our attack departs from prior approaches that reconstruct a model in a *bottom-up* fashion, starting from the input layer. Instead, our attack operates *top-down* and directly extracts the model's last layer. Specifically, we exploit the fact that the final layer of a language model projects from the hidden dimension to a (higher dimensional) logit vector. This final layer is thus low-rank, and by making targeted queries to a model's API, we can extract its embedding dimension or its final weight matrix.

Stealing this layer is useful for several reasons. First, it reveals the *width* of the transformer model, which is often correlated with its total parameter count. Second, it slightly reduces the degree to which the model is a complete "black-box", which so might be useful for future attacks. Third, while our attack recovers only a (relatively small) part of the entire model, the fact that it is at all possible to steal *any* parameters of a production model is surprising, and raises concerns that extensions of this attack might be able to recover more information. Finally, recovering the model's last layer (and thus hidden dimension) may reveal more global information about the model, such as relative size differences between different models.

Our attack is effective and efficient, and is applicable to production models whose APIs expose full logprobs, or a "logit bias". This included Google's PaLM-2 and OpenAI's GPT-4 (Anil et al., 2023; OpenAI et al., 2023); after responsible disclosure, both APIs have implemented defenses to prevent our attack or make it more expensive. We extract the embedding layer of several OpenAI models with a mean squared error of 10^{-4} (up to unavoidable symmetries). We apply a limited form of our attack to `gpt-3.5` at a cost of under \$200 USD and, instead of recovering the full embedding layer, recover just the size of the embedding dimension.

[1]Google DeepMind [2]ETH Zurich [3]University of Washington [4]OpenAI [5]McGill University.

Responsible disclosure. We shared our attack with all services we are aware of that are vulnerable to this attack. We also shared our attack with several other popular services, even if they were not vulnerable to our specific attack, because variants of our attack may be possible in other settings. We received approval from OpenAI prior to extracting the parameters of the last layers of their models, worked with OpenAI to confirm our approach's efficacy, and then deleted all data associated with the attack. In response to our attack, OpenAI and Google have both modified their APIs to introduce mitigiations and defenses (like those that we suggest in Section 8) to make it more difficult for adversaries to perform this attack.

2. Related Work

Model stealing attacks (Tramèr et al., 2016) aim to recover the functionality of a black-box model, and optimize for one of two objectives (Jagielski et al., 2020):

1. *Accuracy*: the stolen model $\hat{f}$ should match the performance of the target model f on some particular data domain. For example, if the target is an image classifier, we might want the stolen model to match the target's overall accuracy on ImageNet.

2. *Fidelity*: the stolen model $\hat{f}$ should be functionally equivalent to the target model f on all inputs. That is, for any valid input p, we want $\hat{f}(p) \approx f(p)$.

In this paper, we focus on high-fidelity attacks. Most prior high-fidelity attacks exploit specific properties of deep neural networks with ReLU activations. Milli et al. (2019) first showed that if an attacker can compute *gradients* of a target two-layer ReLU model, then they can steal a nearly bit-for-bit equivalent model. Jagielski et al. (2020) observed that if the attacker only has query access to model outputs, they can approximate gradients with finite differences. Subsequent work extended these attacks to efficiently extract deeper ReLU models (Carlini et al., 2020; Rolnick & Kording, 2020; Shamir et al., 2023). Unfortunately, none of these approaches scale to production language models, because they (1) accept tokens as inputs (and so performing finite differences is intractable); (2) use activations other than ReLUs; (3) contain architectural components such as attention, layer normalization, residual connections, etc. that current attacks cannot handle; (4) are orders-of-magnitude larger than prior extracted models; and (5) expose only limited-precision outputs.

Other attacks aim to recover more limited information. Wei et al. (2020) show that an adversary co-located on the same server as the LLM can recover the sizes of all hidden layers. Others have attempted to recover model sizes by correlating performance on published benchmarks with model sizes in academic papers (Gao, 2021).

3. Problem Formulation

We study models that take a sequence of tokens drawn from a vocabulary $\mathcal{X}$ as input. Let $\mathcal{P}\left(\mathcal{X}\right)$ denote the space of probability distributions over $\mathcal{X}$. We study parameterized models $f_\theta : \mathcal{X}^N \to \mathcal{P}\left(\mathcal{X}\right)$ that produce a probability distribution over the next output token, given an input sequence of N tokens. The model has the following structure:

$$f_\theta(p) = \mathsf{softmax}(\mathbf{W} \cdot g_\theta(p)), \tag{1}$$

where $g_\theta : \mathcal{X}^N \to \mathbb{R}^h$ is another parameterized model that computes hidden states, $\mathbf{W}$ is an $l \times h$ dimensional matrix (the *embedding projection matrix*), and $\mathsf{softmax} : \mathbb{R}^l \to [0,1]^l$ is the softmax function applied to the resulting *logits*:

$$\mathsf{softmax}(\mathbf{z}) = \left[\frac{e^{\mathbf{z}_1}}{\sum_{i=1}^{l} e^{\mathbf{z}_i}}, \ldots, \frac{e^{\mathbf{z}_l}}{\sum_{i=1}^{l} e^{\mathbf{z}_i}}\right] .$$

Note that the hidden dimension size is much smaller than the size of the token dictionary, i.e., $h \ll l$. For example, LLaMA (Touvron et al., 2023) chooses $h \in \{4096, 5120, 6656, 8192\}$ and $l = 32{,}000$, and there is a recent trend towards increasingly large token sizes; GPT-4, for example, has a ≈100,000 token vocabulary.

Threat model. Throughout the paper, we assume that the adversary does not have any additional knowledge about the model parameters. We assume access to a model f_θ, hosted by a service provider and made available to users through a query interface (API) $\mathcal{O}$. We assume that $\mathcal{O}$ is a perfect oracle: given an input sequence p, it produces $y = \mathcal{O}\left(p\right)$ without leaking any other information about f_θ than what can be inferred from (p, y). For example, the adversary cannot infer anything about f_θ via timing side-channels or other details of the implementation of the query interface.

Different open-source and proprietary LLMs offer APIs with varying capabilities, which impact the ability to perform model extraction attacks and the choice of attack algorithm. A summary of the different APIs we study, and our motivation for doing so, is presented in Table 1. The logits API is a strawman threat model where the API provides logits for all tokens in the response to a given prompt. We begin with this toy setting, as the attack techniques we develop here can be reused in subsequent sections, where we will first reconstruct the logits from more limited information (e.g., log-probabilities for only the top few tokens) and then run the attack.

4. Extraction Attack for Logit-Vector APIs

In this section, we assume the adversary can directly view the logits that feed into the softmax function for every token in the vocabulary (we will later relax this assumption), i.e.,

$$\mathcal{O}\left(p\right) \leftarrow \mathbf{W} \cdot g_\theta\left(p\right) .$$

Table 1. Summary of APIs

API	Motivation
All Logits §4	Pedagogy & basis for next attacks
Top Logprobs, Logit-bias §5	Current LLM APIs (e.g., OpenAI)
No logprobs, Logit-bias §6	Potential future constrained APIs

We develop new attack techniques that allow us to perform high-fidelity extraction of (a small part of) a transformer. Section 4.1 demonstrates how we can identify the hidden dimension h using the logits API and Section 4.2 presents an algorithm that can recover the matrix $\mathbf{W}$.

4.1. Warm-up: Recovering Hidden Dimensionality

We begin with a simple attack that allows an adversary to recover the size of the hidden dimension of a language model by making queries to the oracle $\mathcal{O}$ (Algorithm 1). The techniques we use to perform this attack will be the foundation for attacks that we further develop to perform complete extraction of the final embedding projection matrix.

Algorithm 1 Hidden-Dimension Extraction Attack

```
Require: Oracle LLM $\mathcal{O}$
1: Initialize $n$ to an appropriate value greater than $h$
2: Initialize an empty matrix $\mathbf{Q} = \mathbf{0}^{n \times l}$
3: for $i = 1$ to $n$ do
4:     $p_i \leftarrow$ RandPrefix()    ▷ Choose a random prompt
5:     $\mathbf{Q}_i \leftarrow \mathcal{O}(p_i)$
6: end for
7: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \leftarrow$ SingularValues($\mathbf{Q}$)
8: count $\leftarrow \arg\max_i \log\|\lambda_i\| - \log\|\lambda_{i+1}\|$
9: return count
```

Intuition. Suppose we query a language model on a large number of different random prefixes. Even though each output logit vector is an l-dimensional vector, they all actually lie in a h-dimensional subspace because the embedding projection layer up-projects from h-dimensions. Therefore, by querying the model "enough" (more than h times) we will eventually observe new queries are linearly dependent of past queries. We can then compute the dimensionality of this subspace (e.g., with SVD) and report this as the hidden dimensionality of the model.

Formalization. The attack is based on the following straightforward mathematical result:

Lemma 4.1. *Let $\mathbf{Q}(p_1, \ldots p_n) \in \mathbb{R}^{l \times n}$ denote the matrix with columns $\mathcal{O}(p_1), \ldots, \mathcal{O}(p_n)$ of query responses from the logit-vector API. Then*

$$h \geq \mathit{rank}\left(\mathbf{Q}(p_1, \ldots p_n)\right).$$

Further, if the matrix with columns $g_\theta(p_i)$ $(i = 1, \ldots, n)$ has rank h and $\mathbf{W}$ has rank h, then

$$h = \mathit{rank}\left(\mathbf{Q}(p_1, \ldots p_n)\right).$$

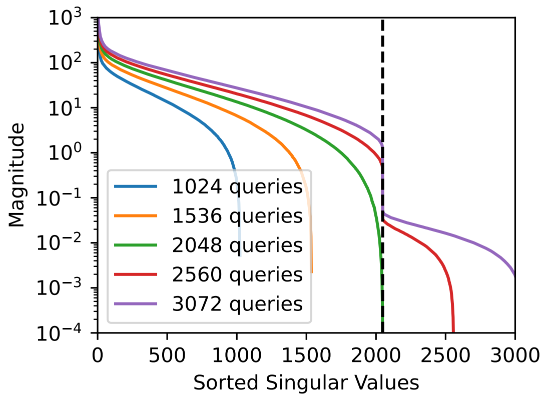


Figure 1. SVD can recover the hidden dimensionality of a model when the final output layer dimension is greater than the hidden dimension. Here we extract the hidden dimension (2048) of the Pythia 1.4B model. We can precisely identify the size by obtaining slightly over 2048 full logit vectors.

Proof. We have $\mathbf{Q} = \mathbf{W} \cdot \mathbf{H}$, where $\mathbf{H}$ is a $h \times n$ matrix whose columns are $g_\theta(p_i)$ $(i = 1, \ldots, n)$. Thus, $h \geq \text{rank}(\mathbf{Q})$. Further, if $\mathbf{H}$ has rank h (with the second assumption), then $h = \text{rank}(\mathbf{Q})$. □

Assumptions. In Lemma 4.1, we assume that both the matrix with columns $g_\theta(p_i)$ and the matrix $\mathbf{W}$ have rank h. These matrices have either h rows or h columns, so both have rank at most h. Moreover, it is very unlikely that they have rank $< h$: this would require the distribution of $g_\theta(p)$ to be fully supported on a subspace of dimension $< h$ across all p_i we query, or all $h \ll l$ columns of $\mathbf{W}$ to lie in the same $(h-1)$ dimensional subspace of $\mathbb{R}^l$ (the output space of logits). In practice we find this assumption holds for all larger models (Table 2) and when different normalization layers are used (Appendix B.1).

Practical considerations. Since the matrix $\mathbf{Q}$ is not computed over the reals, but over floating-point numbers (possibly with precision as low as 16-bits or 8-bits for production neural networks), we cannot naively take the rank to be the number of linearly independent rows. Instead, we use a practical *numerical rank* of $\mathbf{Q}$, where we order the singular values $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, and identify the largest *multiplicative* gap $\frac{\lambda_i}{\lambda_{i+1}}$ between consecutive singular values. A large multiplicative gap arises when we switch from large "actual" singular values to small singular values that arise from numerical imprecision. Figure 2 shows these gaps. Algorithm 1 describes this attack.

Experiments. In order to visualize the intuition behind this attack, Figure 1 illustrates an attack against the Pythia-1.4b LLM. Here, we plot the magnitude of the singular values of $\mathbf{Q}$ as we send an increasing number n of queries to the model. When we send fewer than 2048 queries it

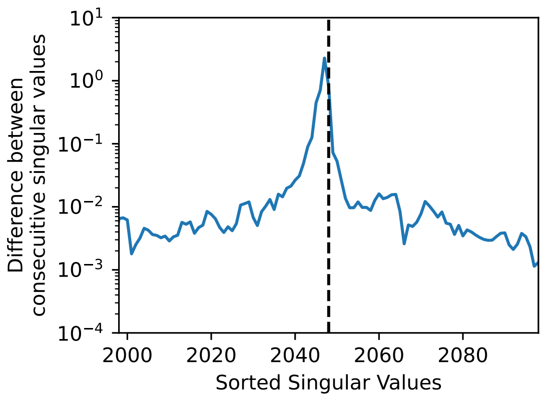


Figure 2. Our extraction attack recovers the hidden dimension size by looking for a sharp drop in singular values, visualized as a spike in the difference between consecutive singular values. On Pythia-1.4B, a 2048 dimensional model, the spike occurs at 2047 singular values.

is impossible to identify the dimensionality of the hidden space. This is because $n < h$, and so the $n \times l$ dimensional matrix $\mathbf{Q}$ has full rank and n nontrivial singular values. But once we make more than 2048 queries to the model, and thus $n > h$, the number of numerically significant singular values does not increase further; it is capped at exactly 2048.

In Figure 2 we plot the difference (in log-space) between subsequent singular values. As we can see, the largest difference occurs at (almost exactly) the 2048th singular value—the true hidden dimensionality of this model.

We now analyze the efficacy of this attack across a wider range of models: GPT-2 (Radford et al., 2019) Small and XL, Pythia (Biderman et al., 2023) 1.4B and 6.9B, and LLaMA (Touvron et al., 2023) 7B and 65B. The results are in Table 2: our attack recovers the embedding size nearly perfectly, with an error of 0 or 1 in five out of six cases.

Our near perfect extraction has one exception: GPT-2 Small. On this 768 dimensional model our attack reports a hidden dimension of 757. In Appendix A we show that this "failure" is caused by GPT-2 actually having an effective hidden dimensionality of 757 despite having 768 dimensions.

4.2. Full Layer Extraction (Up to Symmetries)

We extend the attack from the prior section to recover the final output projection matrix $\mathbf{W}$ that maps from the final hidden layer to the output logits.

Method: Let $\mathbf{Q}$ be as defined in Algorithm 1. Now rewrite $\mathbf{Q} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^\top$ with SVD. Previously we saw that the number of large enough singular values corresponded to the dimension of the model. But it turns out that the matrix $\mathbf{U}$ actually directly represents (a rotation of) the final layer! Specifically, we can show that $\mathbf{U} \cdot \mathbf{\Sigma} = \mathbf{W} \cdot \mathbf{G}$ for some $h \times h$ matrix $\mathbf{G}$ in the following lemma.

Table 2. Our attack succeeds across a range of open-source models, at both stealing the model size, and also at reconstructing the output projection matrix (up to invariances; we show the root MSE).

Model	Hidden Dim	Stolen Size	$\mathbf{W}$ RMS
GPT-2 Small (fp32)	768	757 ± 1	$4 \cdot 10^{-4}$
GPT-2 XL (fp32)	1600	1599 ± 1	$6 \cdot 10^{-4}$
Pythia-1.4 (fp16)	2048	2047 ± 1	$3 \cdot 10^{-5}$
Pythia-6.9 (fp16)	4096	4096 ± 1	$4 \cdot 10^{-5}$
LLaMA 7B (fp16)	4096	4096 ± 2	$8 \cdot 10^{-5}$
LLaMA 65B (fp16)	8192	8192 ± 2	$5 \cdot 10^{-5}$

Lemma 4.2. *In the logit-API threat model, under the assumptions of Lemma 4.1:* (i) *The method above recovers* $\tilde{\mathbf{W}} = \mathbf{W} \cdot \mathbf{G}$ *for some* $\mathbf{G} \in \mathbb{R}^{h \times h}$*;* (ii) *With the additional assumption that* $g_\theta(p)$ *is a transformer with residual connections, it is impossible to extract* $\mathbf{W}$ *exactly.*

Proof. See Appendix C. □

Experiments. For the six models considered previously, we evaluate the attack success rate by comparing the root mean square (RMS) between our extracted matrix $\tilde{\mathbf{W}} = \mathbf{U} \cdot \mathbf{\Sigma}$ and the actual weight matrix, after allowing for a $h \times h$ affine transformation. Concretely, we solve the least squares system $\tilde{\mathbf{W}} \cdot \mathbf{G} \approx \mathbf{W}$ for $\mathbf{G}$, which reduces to h linear least squares problems, each with l equations and h unknowns. Then, we report the RMS of $\mathbf{W}$ and $\tilde{\mathbf{W}} \cdot \mathbf{G}$.

The results are in Table 2. As a point of reference, the RMS between a randomly initialized model and the actual weights is $2 \cdot 10^{-2}$, over 100–500$\times$ higher than the error of our reconstruction.

In Appendices C and G, we show that reconstruction is possible up an *orthogonal* transformation (approximately $h^2/2$ missing parameters, as opposed to h^2 for reconstruction up to an affine transformation), and that this is tight under some formal assumptions. However, we only have an efficient algorithm for reconstruction up to affine transformations.

5. Extraction Attack for Logit-Bias APIs

The above attack makes a significant assumption: that the adversary can directly observe the complete logit vector for each input. In practice, this is not true: no production model we are aware of provides such an API. Instead, for example, they provide a way for users to get the top-K (by logit) token log probabilities. In this section we address this challenge.

5.1. Description of the API

In this section we develop attacks for APIs that return log probabilities for the top K tokens (sorted by logits), and where the user can specify a real-valued bias $b \in \mathbb{R}^{|\mathcal{X}|}$ (the "logit bias") to be added to the logits for specified tokens before the softmax, i.e.,

$$\begin{aligned}\mathcal{O}(p, b) &\leftarrow \mathsf{TopK}\left(\text{logsoftmax}\left(\mathbf{W} g_\theta(p) + b\right)\right)\\ &= \mathsf{TopK}\left(\mathbf{W} g_\theta(p) + b - \log\left(\sum_i \exp(\mathbf{W} g_\theta(p) + b)_i\right) \cdot \mathbf{1}\right).\end{aligned}$$

where $\mathsf{TopK}\left(\mathbf{z}\right)$ returns the K highest entries of $\mathbf{z} \in \mathbb{R}^l$ and their indices. Many APIs (prior to this paper) provided such an option for their state-of-the-art models (OpenAI, 2024; Google, 2024). In particular, the OpenAI API supports modifying logits for at most 300 tokens, and the logit bias for each token is restricted to the range $[-100, 100]$ (OpenAI, 2023).

All that remains is to show that we can uncover the full logit vector for distinct prompt queries through this API. In this section, we develop techniques for this purpose. Once we have recovered multiple complete logit vector, we can run the attack from Section 4.2 without modification.

5.2. Evaluation Methodology

Practical attacks must be *efficient*, both to keep the cost of extraction manageable and to bypass any rate limiters or other filters in the APIs. We thus begin with two cost definitions that we use to measure the efficacy of our attack.

Token cost: the number of tokens the adversary sends to (or receives from) the model during the attack. Most APIs charge users per-token, so this metric represents the monetary cost of an attack (after scaling by the token cost).

Query cost: the total duration of the attack. Most APIs place a limit on the number of queries an adversary can make in any given interval, and so some attacks may be faster but cost more (by sending more tokens per query).

In the remainder of this section we develop several attacks under varying attack assumptions and optimizing for either *token cost*, *query cost*, or both.

5.3. Extraction Attack for Top-5 Logit Bias APIs

We develop a technique to compute the logit vector for any prefix p via a sequence of queries with varying logit biases.

To begin, **suppose that the API returned the top K logits**. Then we could recover the complete logit vector for an arbitrary prompt p by cycling through different choices for the logit bias and measuring the top-k logits each time. In particular, for an API with top-5 logits we can send a sequence of queries

$$\mathcal{O}(p, b_k = b_{k+1} = \ldots = b_{k+4} = B), \text{ for } k \in \{0, 5, 10, \ldots, |\mathcal{X}|\}$$

with a large enough B. Each query thus promotes five different tokens $\{k, k+1, \ldots, k+4\}$ into the top-5, which allows us to observe their logits. By subtracting the bias B and merging answers from all of these queries, we recover the entire logit vector.

Unfortunately, we cannot use this attack directly because all production APIs we are aware of return *logprobs* (the log of the softmax output of the model) instead of logits. The problem now is that when we apply a logit bias B to the i-th token and observe that token's logprob, we get the value

$$y_i^B = z_i + B - \log\left(\sum_{j \neq i} \exp(z_j) + \exp(z_i + B)\right)$$

We thus get an additional bias-dependent term which we need to deal with. We propose two approaches.

Our first approach relies on a common "reference" token that lets us learn the relative difference between all logits (this is the best we can hope for, since the softmax is invariant under additive shifts to the logits). Suppose the top token for a prompt is R, and we want to learn the relative difference between the logits of tokens i and R. We add a large bias B to token i to push it to the top-5, and then observe the logprobs of both token i and R. We have:

$$y_R^B - y_i^B - B = z_R + z_i\,.$$

Since we can observe 5 logprobs, we can compare the reference token R to four tokens per query, by adding a large bias that pushes all four tokens into the top 5 (along with the reference token). We thus issue a sequence of queries

$$\mathcal{O}(p, b_i = b_{i+1} = b_{i+2} = b_{i+3} = B)$$

for $i \in \{0, 4, 8, \cdots, |\mathcal{X}|\}$. This recovers the logits up to the free parameter z_R that we set to 0.

Query cost. This attack reveals the value of 4 logits with each query to the model (the fifth being used as a reference point), for a cost of $1/4$ queries per logit.

In Appendix E we present a second, more sophisticated method that allows us to recover 5 logits per query, i.e., a cost of $1/5$, by viewing each logprob we receive as a linear constraint on the original logits.

Token cost. Recall that our attack requires that we learn the logits for several distinct prompts; and so each prompt must be at least one token long. Therefore, this attack costs at least two tokens per query (one input and one output), or a cost of $1/2$ for each token of output. But, in practice, many models (like `gpt-3.5-turbo`) include a few tokens of overhead along with every single query. This increases the token cost per logit to $\frac{2+\Delta}{4}$ where Δ is the number of overhead tokens; for `gpt-3.5-turbo` we report $\Delta = 7$.

An improved cost-optimal attack. It is possible to generalize the above attack to improve *both* the query cost and token cost. Instead of issuing queries to the model that reveal 4 or 5 logit values for a single generated token, we might instead hope to be able to send a multi-token query $[p_0 \quad p_1 \quad p_2 \dots p_n]$ and then ask for the logprob vector for each prefix of the prompt $[p_0]$, $[p_0 \quad p_1]$, $[p_0 \quad p_1 \quad p_2]$ etc. OpenAI's API did allow for queries of this form in the past, by providing logprobs for *prompt* tokens as well as generated tokens by combining the `logprob` and `echo` parameters; this option has since been removed.

Now, it is only possible to view logprobs of *generated* tokens. And since only the very last token is generated, we can only view four logprobs for this single longer query. This, however, presents a potential approach to reduce the query and token cost: if there were some way to cause the model to emit a specific sequence of tokens $[p_{n+1} \quad p_{n+2} \quad \dots \quad p_{n+m}]$, then we could inspect the logprob vector of each generated token.

We achieve this as follows: we fix a token x and four other tokens, and force the model to emit $[x \quad x \quad \dots \quad x]$. Instead of supplying a logit bias of B for each of the five tokens, we supply a logit bias of B for token x, and $B' < B$ for the other four tokens. If B' is large enough so that the other tokens will be brought into the top-5 outputs, we will still be able to learn the logits for those tokens. As long as B' is small enough so that the model will always complete the initial prompt p_0 with token x (and not any other), then we will be able to collect the logits on several prompts of the form $[p_0 \quad x \quad x \quad \dots \quad x]$.

Analysis. It is easy to see that the query cost of this attack is $\frac{1}{4m}$, where m is the expansion factor. Further, since each query requires $1+m$ tokens, the token cost is $\frac{1+m}{4m}$. (Or, $1+m+\Delta$ if the API has an overhead of Δ tokens.) Note that if $m = 1$, i.e., there is no expansion, this attack reduces to our first attack and the analysis similarly gives a query cost of $1/4$ and a token cost of $1/2$.

5.4. Extraction Attack for top-1 Binary Logit Bias APIs

In light of our attacks, it is conceivable that model providers introduce restrictions on the above API. We now demonstrate that an attack is possible even if the API only returns the top logprob ($K = 1$ in the API from Section 5.1), and the logit bias is constrained to only take one of two values.

API. We place two following further restrictions on the logit bias API (Section 5.1): first, we set $K = 1$, and only see the most likely token's logprob; and second, each logit bias entry b is constrained to be in $\{-1, 0\}$. These constraints would completely prevent the attacks from the prior section. We believe this constraint is significantly tighter than any practical implementation would define.

Method. At first it may seem impossible to be able to learn any information about a token t if it is not already the most likely token. However, note that if we query the model twice, once without any logit bias, and once with a logit bias of -1 for token t, then the top token will be *slightly* more likely with a bias of -1, with exactly how slight depending on the *value* of token t's logprob. Specifically, in Appendix D we show the logprob equals $(1/e - 1)^{-1}(\exp(y_{\text{top}} - y'_{\text{top}}) - 1)$ where y_{top} and y'_{top} are the logprobs of the most likely token when querying with logit bias of 0 and -1.

Analysis. This attack requires 1 query and token per logprob extracted. However, as we will show in the evaluation, this attack is much less numerically stable than the previously-discussed attacks, and so may require more queries to reach the same level of accuracy.

6. Extraction From Logprob-free APIs

A more conservative API provider may remove access to the combination of logit bias and logprobs entirely. Indeed, after disclosing our attack to OpenAI, they removed the ability for logit bias to impact the top logprobs—thus preventing the attacks from the prior sections. To exploit situations such at this, we further develop several logprob-free attacks that recover the complete logit vector by performing binary search on the logit bias vector, albeit at increased cost.

API: Some APIs provide access to a logit bias term, but do not provide any information about the logprobs. Thus, we have,

$$\mathcal{O}(p, b) = \mathsf{ArgMax}\left(\mathsf{logsoftmax}\left(\mathbf{W} \cdot g_\theta(p) + b\right)\right).$$

where $\mathsf{ArgMax}(\mathbf{z})$ returns the index of the highest coordinate in the vector $\mathbf{z} \in \mathbb{R}^l$. In this section, we will use the notation $b = \{i : z\}$ to denote that the bias is set to z for token i and 0 for every other token. We also use $b = \{\}$ to denote that no logit bias is used. Finally, we assume that the bias is restricted to fall within the range $[-B, B]$.

What can be extracted? The attacks developed in this Section reconstruct the logit vector up to an additive (∞-norm) error of ε.

6.1. Warm-up: Basic Logprob-free Attack

Method. We make one simple insight for our logprob-free attacks: sampling with temperature 0 produces the token with the largest logit value. By adjusting the logit bias for each token accordingly, we can therefore recover every token's logit value through binary search. Formally, let p be the prompt, and relabel tokens so that the token with index 0 is the most likely token in the response to p, given by $\mathcal{O}(p, b = \{\})$. For each token $i \neq 0$, we run a binary search over the logit bias term to find the minimal value $x_i \geq 0$ such that the model emits token i with probability 1. This recovers all logits (like all prior attacks, we lose one free variable due to the softmax).

Algorithm 2 Learning logit differences

```
$\alpha_i \leftarrow -B, \beta_i \leftarrow 0$
**while** $\beta_i - \alpha_i > \varepsilon$ **do**
    **if** $\mathcal{O}\left(p, b = \{i : -\frac{\alpha_i+\beta_i}{2}\}\right) = 0$ **then**
        $\beta_i \leftarrow \frac{\alpha_i+\beta_i}{2}$
    **else**
        $\alpha_i \leftarrow \frac{\alpha_i+\beta_i}{2}$
    **end if**
    Return $\frac{\alpha_i+\beta_i}{2}$
**end while**
```

Analysis. This attack, while inefficient, correctly extracts the logit vector.

Lemma 6.1. *For every token i such that* $\mathsf{logit}_i - \mathsf{logit}_0 \geq -B$*, Algorithm 2 outputs a value that is at most* ε *away from the* $\mathsf{logit}_i - \mathsf{logit}_0$ *in at most* $\log\left(\frac{B}{\varepsilon}\right)$ *API queries.*

Proof. The API returns the (re-ordered) token 0 as long as the logit bias added is smaller than $\mathsf{logit}_i - \mathsf{logit}_0$. By the assumption, we know that $\mathsf{logit}_i - \mathsf{logit}_0 \in [-B, 0]$. The algorithm ensures that $\beta_i \geq \mathsf{logit}_i - \mathsf{logit}_0 \geq \alpha_i$ at each iteration, as can be seen easily by an inductive argument. Further, $\beta_i - \alpha_i$ decreases by a factor of 2 in each iteration, and hence at termination, we can see that the true value of $\mathsf{logit}_i - \mathsf{logit}_0$ is sandwiched in an interval of length ε. Furthermore, it is clear that the number of iterations is at most $\log_2\left(\frac{B}{\varepsilon}\right)$ and hence so is the query cost of this algorithm. □

Limitations of the approach. If $\mathsf{logit}_i - \mathsf{logit}_0 < -2B$ it is easy to see there is no efficient way to sample the token i, hence no way to find information about logit_i without logprob access. There is a way to slightly increase the range for $-2B \leq \mathsf{logit}_i - \mathsf{logit}_0 \leq -B$ by adding negative logit biases to the tokens with the largest logit values, but we skip the details since for most models, for the prompts we use, the every token satisfies $\mathsf{logit}_i - \mathsf{logit}_0 > -B$.

Related work. Concurrent work (Morris et al., 2023) has discussed this method of extracting logits.

6.2. Improved Logprob-free Attack: Hyperrectangle Relaxation Center

We can improve the previous attack by modifying the logit bias of multiple tokens at once.

API: We use the same API as in the previous section, with the additional constraint that the $\mathcal{O}$ accepts at most $N+1$ tokens in the logit bias dictionary. We again first run a query $\mathcal{O}(p, b = \{\})$ to identify the most likely token and set its index to 0. Our goal is to approximate $\mathsf{logit}_i - \mathsf{logit}_0$ for N different tokens. If $N < l - 1$, we simply repeat the same algorithm for different batches of N tokens $\frac{l-1}{N}$ times.

Algorithm 3 Learning logit differences with multi-token calls

```
$\alpha_i \leftarrow -B, \beta_i \leftarrow 0 \quad \forall i = 1, \ldots, N$
$\mathcal{C} = \{\mathsf{logit} : \mathsf{logit}_i - \mathsf{logit}_0 \leq B \quad \forall i = 1, \ldots, N\}$
**for** $T$ rounds **do**
    $b_i \leftarrow -\frac{\alpha_i+\beta_i}{2}$ for $i = 0, \ldots, N$
    $k \leftarrow \mathcal{O}(p, b = \{0 : b_0, 1 : b_1, \ldots, N : b_N\})$
    **for** $j \neq k$ **do**
        $\mathcal{C} \leftarrow \mathcal{C} \cap \{\mathsf{logit} : \mathsf{logit}_k + b_k \geq \mathsf{logit}_j + b_j\}$
    **end for**
    **for** $i = 0, \ldots, N$ **do**
        $\alpha_i \leftarrow \min_{\mathsf{logit} \in \mathcal{C}} \mathsf{logit}_i - \mathsf{logit}_0$
        $\beta_i \leftarrow \min_{\mathsf{logit} \in \mathcal{C}} \mathsf{logit}_i - \mathsf{logit}_0$
    **end for**
**end for**
Return $[\alpha_i, \beta_i] \quad \forall i \in \{0, \ldots, N\}$
```

Method. Our approach queries the API with the logit bias set for several tokens in parallel. The algorithm proceeds in *rounds*, where each round involves querying the API with the logit bias set for several tokens.

Suppose that the query returns token k as output when the logit bias was set to $\{i : b_i\}$ for $i = 1, \ldots, l$ and the prompt is p. Then, we know that $\mathsf{logit}_k + b_k \geq \mathsf{logit}_j + b_j$ for all $j \neq k$ by the definition of the API.

This imposes a system of linear constraints on the logits. By querying the model many times, and accumulating many such systems of equations, we can recover the logit values more efficiently. To do this, we accumulate all such linear constraints in the set $\mathcal{C}$, and at the end of each round, compute the smaller and largest possible values for $\mathsf{logit}_i - \mathsf{logit}_0$ by solving a linear program that maximizes/minimizes this value over the constraint set $\mathcal{C}$. Thus, at each round, we can maintain an interval that encloses $\mathsf{logit}_i - \mathsf{logit}_0$, and refine the interval at each round given additional information from the queries made in that round. After T rounds (where T is chosen based on the total query budget for the attack), we return the tightest known bounds on each logit.

Lemma 6.2. *Suppose that* $\mathsf{logit}_i - \mathsf{logit}_0 \in [-B, 0]$ *for all* $i = 1, \ldots, l$*. Then, Algorithm 3 returns an interval* $[\alpha_i, \beta_i]$ *such that* $\mathsf{logit}_i - \mathsf{logit}_0 \in [\alpha_i, \beta_i]$ *for each i such that* $\mathsf{logit}_i - \mathsf{logit}_0 \in [-B, 0]$*. Furthermore, each round* *in the algorithm can be implemented in computation time $O(N^3)$ (excluding the computation required for the API call).*

Proof. Algorithm 3 maintains the invariant that $\mathsf{logit}_i - \mathsf{logit}_0 \in [\alpha_i, \beta_i]$ in each round. We will prove by induction that this is true and that the true vector of logits always lies in $\mathcal{C}$. Note that by the assumption stated in the Lemma, this is clearly true at the beginning of the first round. Suppose that this is true after $K < T$ rounds. Then, in the $K+1$-th round, the constraints added are all valid constraints for the true logit vector, since the API returning token k guarantees that $\mathsf{logit}_k + b_k \geq \mathsf{logit}_j + b_j$ for all $j \neq k$. Hence, by induction, the algorithm always ensures that $\mathsf{logit}_i - \mathsf{logit}_0 \in [\alpha_i, \beta_i]$.

In Section 6.2.1, we show the LP to compute α_i, β_i for all i can be seen as an all-pairs shortest paths problem on graph with edge weights $c_{jk} = \min_{\text{rounds}} b_j - b_k$ where the minimum is taken over all rounds where the token returned was k. This ensures the computation complexity of maintaining the logit difference intervals is $O(N^3)$. □

6.2.1. Shortest-path Formulation of the Logprob-free Attack LP

It is actiually possible to improve the computational efficiency of the hyperrectangle relaxation of the polytope $\mathcal{C}$. Here we show how to formulate this problem as a shortest path problem on a weighted graph. This enables us to quickly compute the exact $[\alpha_i, \beta_i]$ for all $i \in \{1, \dots, N\}$ after each query.

Lemma 6.3. *Let $G = (\{0, 1, \dots, N\}, E)$ be a weighted directed graph without negative cycles. Let $\mathcal{P} \subset \mathbb{R}^{n+1}$ be the solution set of a system of linear inequalities:*

$$\mathsf{logit}_i - \mathsf{logit}_j \leq c_{ji} \quad \forall\, j \xrightarrow{c_{ji}} i \quad \in E$$

Then if $\mathsf{logit}_0 = 0$*, we have*

$$\max_{x \in \mathcal{C}} \mathsf{logit}_i = \textit{distance in } G \textit{ from } 0 \textit{ to } i.$$

Proof. Let $e_{0j_1}, e_{j_1 j_2}, \dots, e_{j_{m-1} i}$ be the edges of the minimum distance path from 0 to i in G. We have

$$\begin{aligned}\mathsf{logit}_i &\leq \mathsf{logit}_{j_{m-1}} + c_{j_{m-1} i} \leq \dots \\ &\leq \mathsf{logit}_0 + \sum_{t=1}^{m-1} c_{j_{t+1} j_t} = \sum_{t=1}^{m-1} c_{j_{t+1} j_t},\end{aligned}$$

hence the shortest path is an upper bound on logit_i. To prove feasibility, we claim that setting logit_i to be the distance from 0 to i satisfies all the inequalities. Assume some inequality $\mathsf{logit}_i - \mathsf{logit}_j \leq c_{ji}$ is violated. Then we can go from $0 \to j \to i$ in G with a total weight of $\mathsf{logit}_j + c_{ji} < \mathsf{logit}_i$, which contradicts the assumption that logit_i is the distance from 0 to i. □

To apply this to our setting, note that (1) all constraints, even the initial $\alpha_i \leq \mathsf{logit}_i \leq \beta_i$, are of the required form; (2) the graph has no negative cycles because the true logits give a feasible solution. (3) we can get the lower bounds by applying the same procedure to the graph induced by inequalities on $-\mathsf{logit}_i$.

We can find the distances from 0 to all other vertices using the Bellman-Ford algorithm in $O(N^3)$ time. If $N = 300$, this is at most comparable to the latency of $\mathcal{O}$. Since only N edges of the graph update at each step, we note that the heuristic of just updating and doing a few incremental iterations of Bellman-Ford gets $[\alpha_i, \beta_i]$ to high precision in practice. The number of API queries and the token cost, of course, remains the same.

6.3. Improved Logprob-free Attack: Better Queries on Hyperrectangles

The main issue of the previous approach is that some tokens are sampled more often than others, even in the case our prior for the logit vector is uniform over $[-B, 0]$. This is because the "centering of the hyperrectangle" logit bias does not partition the hyperrectangle into equally-sized parts labeled by the argmax coordinate. For example, if $\beta_i - \alpha_i \ll \beta_j - \alpha_j$, under an uniform prior over $[\alpha_i, \beta_i] \times [\alpha_j, \beta_j]$, j will be much more likely to be the output token than i. Hence, in Algorithm 3 we rarely get constraints lower-bounding logit_i in terms of other logits, which makes for weaker relaxations of $\mathcal{C}$.

Our solution is to bias tokens so that the output token distribution is closer to uniform; in particular, biasing the token with the smallest $\beta_t - \alpha_t$ (the 0 token) to have probability exactly $1/(N+1)$ given an uniform prior over the hyperrectangle. One logit bias that satisfies this is:

$$\begin{aligned} b_i &= -(1-c)\alpha_i - c\beta_i \quad \forall i = 0, \dots, N \\ \text{where} \quad c &= \exp(-\log(N+1)/N). \end{aligned} \tag{2}$$

We now run Algorithm 3, with one simple modification: we replace $b_i = -\frac{\alpha+\beta}{2}$ with $b = -(1-c)\alpha - c\beta$. As can be seen in Table 3, the modified algorithm outperforms the method in 6.2 significantly.

The goal of balanced sampling of all output tokens can be approached in many ways. For example, we could tune c in the above expression; bias tokens which $\mathcal{O}$ hasn't returned previously to be more likely; or solve for the exact logit bias that separates $\mathcal{C}$ (or some relaxation) into equal parts. the queries/logit metric of this method in Table 3 is surprisingly close to optimal.

Table 3. Average error at recovering the logit vector for each of the logit-estimation attacks we develop. Our highest precision, and most efficient attack, recovers logits nearly perfectly; other attacks approach this level of precision but at a higher query cost.

Attack	Logprobs	Bits of precision	Queries per logit
logprob-4 (§5.3)	top-5	23.0	0.25
logprob-5 (§E)	top-5	11.5	0.64
logprob-1 (§5.4)	top-1	6.1	1.0
binary search (§6.1)	✗	7.2	10.0
hyperrectangle (§6.2)	✗	15.7	5.4
one-of-n (§6.3)	✗	18.0	3.7

7. Evaluation

We now study the efficacy of our practical stealing attack.

7.1. Logit Validation

We begin by validating that the attacks developed in the prior sections can effectively recover the full logit vector given a limited query interface. In Table 3 we report the average number of bits of agreement between the true logit vector and the recovered logit vector, as well as the (amortized) number of queries required to recover one full logit vector.

Generally, attacks that operate under stronger threat models have higher precision. But theoretical improvements are not always practical: the theoretically stronger attack from §E that learns 5 logprobs per query in practice requires more queries and recovers logits with lower fidelity. This is because this attack is numerically unstable: it requires a potentially ill-conditioned matrix, and therefore can require re-querying the API after adjusting the logit bias. Our strongest logprob-free attack is highly efficient, and recovers 18 bits of precision at just 3.7 queries per logit. In Appendix F we theoretically analyze how far this is from optimal, and find it is within a factor of two.

7.2. Stealing Parts of Production Models

We now investigate our ability to steal production language models, focusing on five of OpenAI's models available on 1 January 2024: `ada`, `babbage`, `babbage-002`, `gpt-3.5-turbo-instruct`, and `gpt-3.5-turbo-1106`. We selected these models because these were the only production models which were able to receive advance permission to attempt an extraction attack; we are exceptionally grateful to OpenAI for allowing us to perform this research using their models.

Given the results from the prior section, we chose to implement the improved 4-logprob attack (Section 5.3) because it is both the most query efficient attack and also the most precise attack. Switching to a different attack algorithm would increase our total experiment cost significantly, and so we do not perform these ablation studies.

Both our hidden-dimension-stealing and entire-layer-stealing attack worked for all five of these models. The size we recover from the model perfectly matches the actual size of the original model, as confirmed by OpenAI. For the first three models, we report in Table 4 the size we recover because (1) the sizes of these models was never previously confirmed, but (2) they have now been deprecated and so disclosing the size is not harmful. In discussions with OpenAI, we decided to withhold disclosure of the size of `gpt-3.5-turbo` models, but we confirmed with them that the number our attack reported was accurate.

When running the full layer-stealing attack, we confirmed that our extracted weights are nearly identical to the actual weights, with error $< 7 \cdot 10^{-4}$, up to an $h \times h$ matrix product as discussed previously. Table 4 reports the RMS between our extracted weight matrix and the actual model weights, after "aligning" the two by an $h \times h$ transform.

8. Defenses

It would be possible to prevent or mitigate this attack in a number of different ways, albeit with loss of functionality.

8.1. Prevention

Remove logit bias. Perhaps the simplest defense would be to outright remove the logit bias parameter from the API. Unfortunately, there are several legitimate use cases of this parameter. For example, several works use logit bias in order to perform controlled or constrained generation (Jiang et al., 2023; Yang & Klein, 2021), to shift generation and mimic fine-tuning the model (Liu et al., 2024; Mitchell et al., 2024), or other reasons (Ren et al., 2023; Lee et al., 2022).

Replace logit bias with a block-list. Instead of offering a logit bias, model developers could replace it with a block-list of tokens the model is prohibited from emitting. This would support (some) of the functionality discussed in the prior section, but would still prevent our attack.

Architectural changes. Instead of modifying the API, we could instead make changes to the model. Our attack only works because the hidden dimension h is less than the output dimension l. This suggests a natural architectural defense: split the final layer into two layers, one that goes from $h \rightarrow t$ and then $t \rightarrow l$ where $t > l$ and a nonlinearity was placed in between. This is not very efficient though, as the last linear layer is large (quadratic in the vocabulary size).

Post-hoc altering the architecture. We can also modify the hidden dimension h for the final layer after the model is trained. In particular, we can expand the dimensionality of $\mathbf{W}$ by concatenating extra weight vectors that are orthogonal to the original matrix. We set the singular values for

Table 4. Attack success rate on five different black-box models

Model	Dimension Extraction			Weight Matrix Extraction		
	Size	# Queries	Cost (USD)	RMS	# Queries	Cost (USD)
OpenAI `ada`	1024 ✓	$< 2 \cdot 10^6$	\$1	$5 \cdot 10^{-4}$	$< 2 \cdot 10^7$	\$4
OpenAI `babbage`	2048 ✓	$< 4 \cdot 10^6$	\$2	$7 \cdot 10^{-4}$	$< 4 \cdot 10^7$	\$12
OpenAI `babbage-002`	1536 ✓	$< 4 \cdot 10^6$	\$2	†	$< 4 \cdot 10^6$ †+	\$12
OpenAI `gpt-3.5-turbo-instruct`	* ✓	$< 4 \cdot 10^7$	\$200	†	$< 4 \cdot 10^8$ †+	\$2,000†+
OpenAI `gpt-3.5-turbo-1106`	* ✓	$< 4 \cdot 10^7$	\$800	†	$< 4 \cdot 10^8$ †+	\$8,000†+

✓ Extracted attack size was exactly correct; confirmed in discussion with OpenAI.
* As part of our responsible disclosure, OpenAI has asked that we do not publish this number.
† Attack not implemented to preserve security of the weights.
+ Estimated cost of attack given the size of the model and estimated scaling ratio.

these weights to be small enough to not materially affect the model's predictions, while also being large enough to look realistic. Then, during the model's forward pass, we concatenate a vector of random Gaussian noise to the final hidden vector $g_\theta(p)$ before multiplying by $\mathbf{W}$. Figure 7 shows an example of this, where we expand GPT-2 small to appear as if it was 1024 dimensional instead of 768 dimensions. This misleads the adversary into thinking that the model is wider than it actually is.

8.2. Mitigations

Logit bias XOR logprobs. Our attack is $10\times$ cheaper when an adversary can supply both a logit bias and also view output logprobs. This suggests a natural mitigation: prohibit queries to the API that make use of *both* logit bias and logprobs at the same time. This type of defense is common in both the security and machine learning community: for example, in 2023 OpenAI removed the ability to combine both `echo` and `logprobs`, but with either alone being allowed; this defense would behave similarly.

Noise addition. By adding a sufficient amount of noise to the output logits of any given query, it would be possible to prevent our attack. However, logit-noise has the potential to make models less useful. We perform some preliminary experiments on this direction in Appendix H.

Rate limits on logit bias. Our attack requires that we are able to learn at least h logit values for each prompt p. One defense would be to allow logit-bias queries to the model, but only allow $T = \tilde{h}/5$ logit bias queries for any given prompt p to prevent an adversary from learning if a model has hidden dimension $\tilde{h}$ or smaller.

Unfortunately this has several significant drawbacks: the threshold has to be independent of h (or learning the threshold would reveal h); the system would need to maintain state of all user queries to the API; and preventing Sybil attacks requires a global pool of user queries, which can present significant privacy risks (Debenedetti et al., 2023).

Detect malicious queries. Instead of preventing any queries that might leak model weights, an alternate strategy could be to implement standard anti-abuse tools to *detect* any patterns of malicious behavior. Several proposals of this form exist for prior machine learning attacks, including model stealing (Juuti et al., 2019; Pal et al., 2021) and adversarial examples (Chen et al., 2020).

9. Future Work

We are motivated to study this problem not because we expect to be able to steal an entire production transformer model bit-for-bit, but because we hope to conclusively demonstrate that model stealing attacks are not just of academic concern but can be practically applied to the largest production models deployed today. We see a number of potential directions for improving on this attack.

Breaking symmetry with quantized weights. Large production models are typically stored "quantized", where each weight is represented in just 4 or 8 bits. In principle, this quantization could allow an adversary to recover a nearly bit-for-bit copy of the matrix $\mathbf{W}$: while there exist an infinite number of matrices $\mathbf{W} \cdot \mathbf{G}$, only one will be discretized properly. Unfortunately, this integer-constrained problem is NP-hard in general (similar problems are the foundation for an entire class of public key cryptosystems). But this need not imply that the problem is hard on all instances.

Extending this attack beyond a single layer. Our attack recovers a single layer of a transformer. We see no obvious methodology to extend it beyond just a single layer, due to the non-linearity of the models. But we invite further research in this area.

Removing the logit bias assumption. All our attacks require the ability to pass a logit bias. Model providers including Google and OpenAI provided this capability when we began the writing of this paper, but this could change. (Indeed, it already has, as model providers begin implementing defenses to prevent this attack.) Other API parameters could give alternative avenues for learning logit information. For example, unconstrained `temperature` and `top-k` parameters could also leak logit values through a series of queries. In the long run, completely hiding the logit information might be challenging due both to public demand for the feature, and ability of adversaries to infer this information through other means.

Exploiting the stolen weights. Recovering a model's embedding projection layer might improve other attacks against that model, e.g., for designing stronger prompt injection attacks or jailbreaks. Alternatively, an attacker could infer details about a provider's *finetuning* API by observing changes (or the absence thereof) in the last layer. In this paper, we focus primarily on the model extraction problem and leave exploring downstream attacks to future work.

Practical stealing of other model information. Existing high-fidelity model stealing attacks are "all-or-nothing" attacks that recover entire models, but only apply to small ReLU networks. We show that stealing partial information can be much more practical, even for state-of-the-art models. Future work may find that practical attacks can steal many more bits of information about current proprietary models.

10. Conclusion

As the field of machine learning matures, and models transition from research artifacts to production tools used by millions, the field of adversarial machine learning must also adapt. While it is certainly useful to understand the potential applicability of model stealing to three-layer 100-neuron ReLU-only fully-connected networks, at some point it becomes important to understand to what extent attacks can be actually applied to the largest production models.

This paper takes one step in that direction. We give an existence proof that it is possible to steal one layer of a production language model. While there appear to be no immediate practical consequences of learning this layer, it represents the first time that *any* precise information about a deployed transformer model has been stolen.

Our attack also highlights how small design decisions influence the overall security of a system. Our attack works because of the seemingly-innocuous *logit-bias* and *logprobs* parameters made available by the largest machine learning service providers, including OpenAI and Google—although both have now implemented mitigations to prevent this attack or make it more expensive. Practitioners should strive to understand how system-level design decisions impact the safety and security of the full product.

Overall, we hope our paper serves to further motivate the study of practical attacks on machine learning models, in order to ultimately develop safer and more reliable systems.

References

Anil, R. et al. PaLM 2 Technical Report, 2023.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Biderman, S. Common LLM settings, 2024. URL `https://rb.gy/2afqlw`. Accessed February 1, 2024.

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, 2023.

Cancedda, N. Spectral filters, dark signals, and attention sinks, 2024.

Carlini, N., Jagielski, M., and Mironov, I. Cryptanalytic extraction of neural network models. In *Annual International Cryptology Conference*, 2020.

Chen, S., Carlini, N., and Wagner, D. Stateful detection of black-box adversarial attacks. In *Proceedings of the 1st ACM Workshop on Security and Privacy on Artificial Intelligence*, 2020.

Chiu, J. openlogprobs, 2024. URL `https://github.com/justinchiu/openlogprobs`. Accessed February 1, 2024.

Debenedetti, E., Severi, G., Carlini, N., Choquette-Choo, C. A., Jagielski, M., Nasr, M., Wallace, E., and Tramèr, F. Privacy side channels in machine learning systems. *arXiv preprint arXiv:2309.05610*, 2023.

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. 8-bit Optimizers via Block-wise Quantization. *ICLR*, 2022.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., DasSarma, N., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., Amodei, D., Brown, T., Clark, J., Kaplan, J., McCandlish, S., and Olah, C. A mathematical framework for transformer circuits. 2021. URL `https://transformer-circuits.pub/2021/framework/index.html`.

Gao, L. On the sizes of OpenAI API models. `https://blog.eleuther.ai/gpt3-model-sizes/`, 2021.

Google. Changelog 1.38.0, 2024. URL `https://cloud.google.com/python/docs/reference/aiplatform/1.38.0/changelog`. Accessed January 30, 2024.

Gurnee, W., Horsley, T., Guo, Z. C., Kheirkhah, T. R., Sun, Q., Hathaway, W., Nanda, N., and Bertsimas, D. Universal neurons in gpt2 language models, 2024.

Hayase, J., Borevkovic, E., Carlini, N., Tramèr, F., and Nasr, M. Query-based adversarial prompt generation. *arXiv preprint arXiv:2402.12329*, 2024.

Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., and Papernot, N. High accuracy and high fidelity extraction of neural networks. In *USENIX Security Symposium*, 2020.

Jiang, Z., Xu, F., Gao, L., Sun, Z., Liu, Q., Dwivedi-Yu, J., Yang, Y., Callan, J., and Neubig, G. Active retrieval augmented generation. In *EMNLP*, 2023.

Juuti, M., Szyller, S., Marchal, S., and Asokan, N. PRADA: protecting against DNN model stealing attacks. In *EuroS&P*, 2019.

Lee, K.-H., Nachum, O., Yang, M. S., Lee, L., Freeman, D., Guadarrama, S., Fischer, I., Xu, W., Jang, E., Michalewski, H., and Mordatch, I. Multi-game decision transformers. In *Advances in Neural Information Processing Systems*, 2022.

Liu, A., Han, X., Wang, Y., Tsvetkov, Y., Choi, Y., and Smith, N. A. Tuning language models by proxy, 2024.

Milli, S., Schmidt, L., Dragan, A. D., and Hardt, M. Model reconstruction from model explanations. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 2019.

Mitchell, E., Rafailov, R., Sharma, A., Finn, C., and Manning, C. D. An emulator for fine-tuning large language models using small language models. In *ICLR*, 2024.

Morris, J. X., Zhao, W., Chiu, J. T., Shmatikov, V., and Rush, A. M. Language model inversion. *arXiv preprint arXiv:2311.13647*, 2023.

OpenAI. Using logit bias to define token probability, 2023. URL https://help.openai.com/en/articles/5247780-using-logit-bias-to-define-token-probability. Accessed Febraury 1, 2024.

OpenAI. Create chat completion, 2024. URL https://platform.openai.com/docs/api-reference/chat/create. Accessed January 30, 2024.

OpenAI et al. GPT-4 Technical Report, 2023.

Pal, S., Gupta, Y., Kanade, A., and Shevade, S. Stateful detection of model extraction attacks. *arXiv preprint arXiv:2107.05166*, 2021.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language Models are Unsupervised Multitask Learners. Technical report, OpenAI, 2019. URL https://rb.gy/tm8qh.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L. A., Rauh, M., Huang, P.-S., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X. L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., Donato, D., Lazaridou, A., Mensch, A., Lespiau, J.-B., Tsimpoukelli, M., Grigorev, N., Fritz, D., Sottiaux, T., Pajarskas, M., Pohlen, T., Gong, Z., Toyama, D., de Masson d'Autume, C., Li, Y., Terzi, T., Mikulik, V., Babuschkin, I., Clark, A., de Las Casas, D., Guy, A., Jones, C., Bradbury, J., Johnson, M., Hechtman, B., Weidinger, L., Gabriel, I., Isaac, W., Lockhart, E., Osindero, S., Rimell, L., Dyer, C., Vinyals, O., Ayoub, K., Stanway, J., Bennett, L., Hassabis, D., Kavukcuoglu, K., and Irving, G. Scaling language models: Methods, analysis and insights from training gopher, 2022.

Ren, J., Zhao, Y., Vu, T., Liu, P. J., and Lakshminarayanan, B. Self-evaluation improves selective generation in large language models. *arXiv preprint arXiv:2312.09300*, 2023.

Rolnick, D. and Kording, K. Reverse-engineering deep relu networks. In *International Conference on Machine Learning*, 2020.

Shamir, A., Canales-Martinez, I., Hambitzer, A., Chavez-Saab, J., Rodrigez-Henriquez, F., and Satpute, N. Polynomial time cryptanalytic extraction of neural network models. *arXiv preprint arXiv:2310.08708*, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. Stealing machine learning models via prediction APIs. In *USENIX Security Symposium*, 2016.

Veit, A., Wilber, M. J., and Belongie, S. J. Residual networks behave like ensembles of relatively shallow networks. In *Advances in Neural Information Processing Systems*, pp. 550–558, 2016.

Wei, J., Zhang, Y., Zhou, Z., Li, Z., and Al Faruque, M. A. Leaky DNN: Stealing deep-learning model secret with GPU context-switching side-channel. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

Yang, K. and Klein, D. FUDGE: Controlled text generation with future discriminators. In Toutanova, K., Rumshisky, A., Zettlemoyer, L., Hakkani-Tur, D., Beltagy, I., Bethard, S., Cotterell, R., Chakraborty, T., and Zhou, Y. (eds.), *ACL*, 2021.

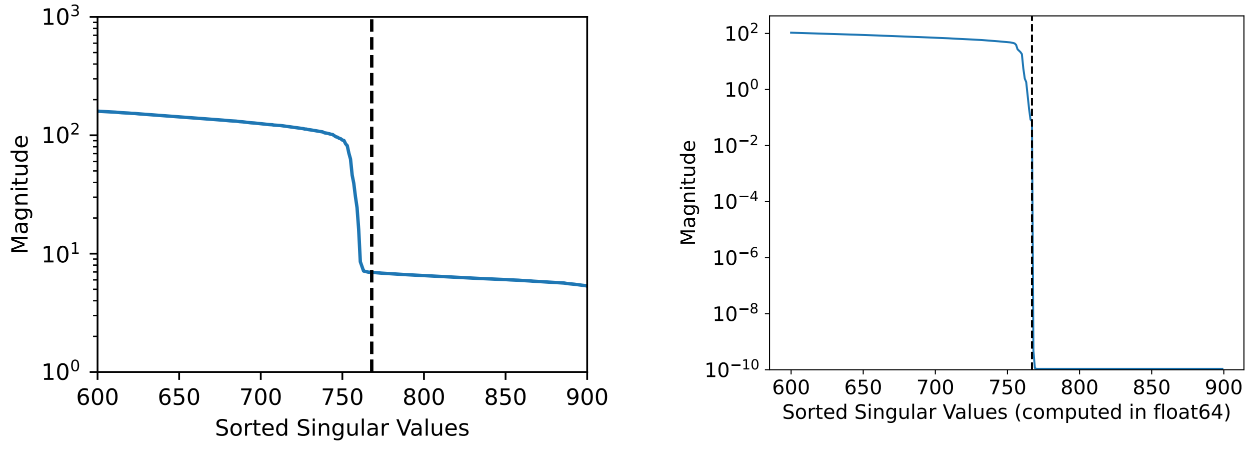
Zhang, B. and Sennrich, R. Root mean square layer normalization. *NeurIPS*, 2019.

A. What's Going On With GPT-2 Small?

Our attack nearly perfectly extracts the model size of all models—except for GPT-2 Small where our extracted size of 757 is off by 11 from the correct 768. Why is this?

In Figure 3 we directly inspect this model's final hidden activation vector across $10,000$ different model queries and perform SVD of the resulting activation matrix. We see that despite GPT-2 actually having 768 potential hidden neurons, there are only 757 different activation directions. Thus, while this model is *technically* a 768 dimensional model, in practice it behaves as if it was a 757 (i.e, the rank of the embedding matrix is 757) dimensional model, and our attack has recovered this effective size.

However, when running the model in higher float64 precision, we find that indeed all dimensions are used, but that the smallest dozen or so singular values are much smaller than the other singular values, an observation made by concurrent work (Cancedda, 2024).



(a). Singular values of GPT-2 Small (default bfloat16 precision)

(b). Singular values of GPT-2 Small (higher float64 precision)

Figure 3. Singular values of final hidden activations of GPT-2 Small.

B. Accounting for Normalization Layers

B.1. LayerNorm Does Not Affect Our Rank h Assumption

Almost all LLMs that have publicly available architecture details use LayerNorm (Ba et al., 2016) or RMSNorm (Zhang & Sennrich, 2019) just before applying the output projection $\mathbf{W}$ (Biderman, 2024). LayerNorm begins with a centering step, which projects its input onto a $(h-1)$-dimensional subspace (and RMSNorm does not). In theory, this could break our assumption that the rank of the matrix with columns $g_\theta(p_i)$ $(i = 1, ..., n)$ has rank h (Lemma 4.1). In practice, all LLMs we surveyed (Biderman, 2024) enabled the LayerNorm bias, which means the matrices had full rank h (besides GPT-2 Small: see Appendix A).

B.2. Stealing Architectural Details About Normalization Layers

B.2.1. Theory

The difference between LayerNorm and RMSNorm (Appendix B.1) could enable attackers to deduce whether models used LayerNorm or RMSNorm. If an attacker recovered an initial logit-vector API query response $\mathcal{O}(p_0)$, then they could apply Lemma 4.1 to $\mathcal{O}(p_1) - \mathcal{O}(p_0), \ldots, \mathcal{O}(p_n) - \mathcal{O}(p_0)$.[1] From the description of the API at the top of Section 4.1, it follows that $\mathcal{O}(p_i) - \mathcal{O}(p_0) = \mathbf{W}(g_\theta(p_i) - g_\theta(p_0))$. This subtraction of g terms occurs immediately after LayerNorm,

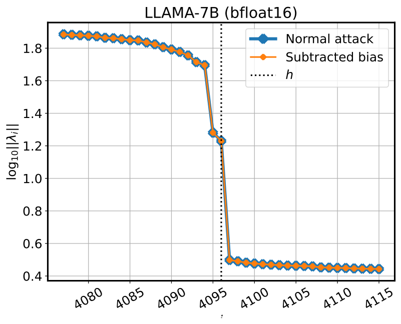
[1]Throughout this appendix section, we assume the sum of logit outputs is always 0. We can calculate centered logits from logprobs by subtracting the mean logits across the vocab dimension.

so cancels the LayerNorm bias term. Hence, if we apply the Lemma 4.1 attack with this subtraction modification to a model using LayerNorm, then the resultant 'h' output will be smaller by 1 (due to Appendix B.1). This would imply the model used LayerNorm rather than RMSNorm, because RMSNorm does not project onto a smaller subspace and so would not have a decrease in 'h' value if we were to use this subtraction trick.

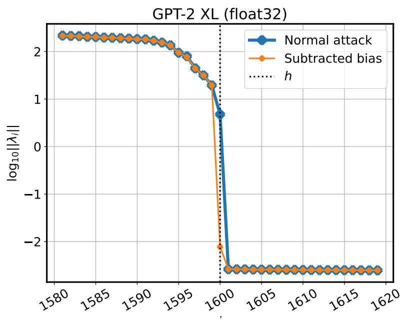
B.2.2. Results

To confirm that the method from Appendix B.2.1 works, we test whether we can detect whether the GPT-2, Pythia and LLAMA architectures use LayerNorm or RMSNorm from their logit outputs alone. We found that the technique required two adjustments before it worked on models with lower than 32-bit precision (it always worked with 32-bit precision). i) We do not subtract $\mathcal{O}(p_0)$ from logits queries, but instead subtract the mean logits over all queries, i.e. $\frac{1}{n}\sum_{i=1}^{n}\mathcal{O}(p_i)$. Since the average of several points in a common affine subspace still lie on that affine subspace, this doesn't change the conclusions from Appendix B.2.1. ii) We additionally found it helped to calculate this mean in lower precision, before casting to 64-bit precision to calculate the compact SVD.

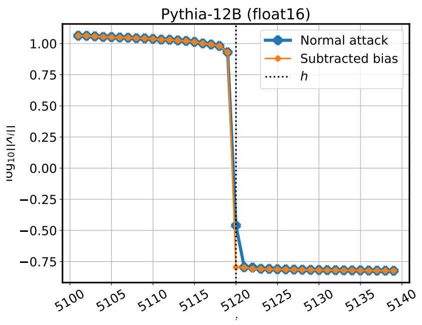
The results are in Figure 4. We plot the singular value magnitudes (as in Figure 1) and show that **there is a drop in the hth singular value for the architectures using LayerNorm, but not for architecture using RMSNorm**:



(a) LLAMA-7B (**RMSNorm**).



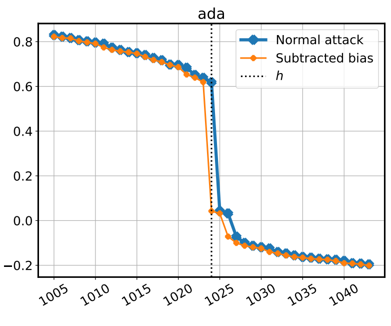
(b) GPT-2 XL (**LayerNorm**).



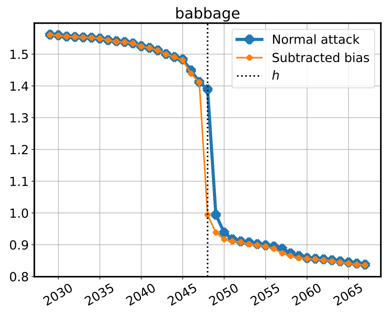
(c) Pythia-12B (**LayerNorm**).

Figure 4. Detecting whether models use **LayerNorm** or **RMSNorm** by singular value magnitudes.

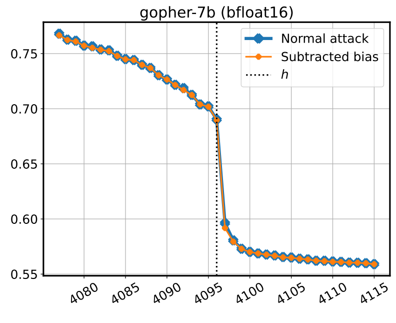
Is this attack practical for real models? We perform the same attack on the logprobs we obtained for `ada` and `babbage`.[2] We see in Figure 5a-b that indeed the drop in the hth singular values occurs for these two models that use LayerNorm (GPT-3's architecture was almost entirely inherited from GPT-2):



(a) `ada` uses **LayerNorm**.



(b) `babbage` uses **LayerNorm**.



(c) Gopher-7B uses **RMSNorm**.

Figure 5. Stress-testing the LayerNorm extraction attack on models behind an API (a-b), and models using both RMSNorm and biases (c).

As a final stress test, we found that all open language models that use RMSNorm do not use any bias terms (Biderman, 2024). Therefore, we checked that our attack would not give a false positive when applied to a model with RMSNorm but with biases. We chose Gopher-7B (Rae et al., 2022), a model with public architectural details but no public weight access,

[2]Unfortunately, we deleted the logprobs for GPT-3.5 models before we created this attack due to security constraints.

that uses RMSNorm but also biases (e.g. on the output logits). In Figure 5c we show that indeed the hth singular value does not decrease for this model that uses RMSNorm.

C. Proof of Lemma 4.2

Restating the lemma from Section 4.2:

Lemma 4.2 *In the logit-API threat model, under the assumptions of Lemma 4.1:* (i) *The method from Section 4.2 recovers* $\tilde{\mathbf{W}} = \mathbf{W} \cdot \mathbf{G}$ *for some* $\mathbf{G} \in \mathbb{R}^{h \times h}$*;* (ii) *With the additional assumption that $g_\theta(p)$ is a transformer with residual connections, it is impossible to extract $\mathbf{W}$ exactly.*

We first give a short proof of (i):

Proof. (i) To show we can recover $\tilde{\mathbf{W}} = \mathbf{W} \cdot \mathbf{G}$, recall Lemma 4.1: we have access to $\mathbf{Q}^\top = \mathbf{W} \cdot \mathbf{H}$ for some $\mathbf{H} \in \mathbb{R}^{h \times n}$. Using the compact SVD of $\mathbf{Q}$ from the method in Section 4.2, $\mathbf{W} \cdot \mathbf{H} \cdot \mathbf{V} = \mathbf{U} \cdot \mathbf{\Sigma}$. We know $\mathbf{G} := \mathbf{H} \cdot \mathbf{V} \in \mathbb{R}^{h \times h}$, hence if we take $\tilde{\mathbf{W}} = \mathbf{U} \cdot \mathbf{\Sigma}$, we have $\tilde{\mathbf{W}} = \mathbf{W} \cdot \mathbf{G}$. □

Proving Lemma 4.2(ii) requires several steps due to the complexity of the transformer architecture: we progressively strengthen the proof to apply to models with no residual connections (C.1), models with residual connections (C.2), models with RMSNorm (C.4), LayerNorm (C.5) and normalization with an ε term (C.6).

C.1. Proof of Lemma 4.2(ii) in Models With Fully-connected Layers

Proof of Lemma 4.2(ii). As a gentle warmup, we prove (ii) under the additional assumption that the model does not use normalization layers (LayerNorm or RMSNorm) in its architecture. To prove (ii) we show it is possible to find a two distinct sets of model parameters θ, θ' with different embedding projection matrices that result in identical API outputs.

We begin with a simpler case where g_θ does not have residual connections but a fully connected (FC) final layer. In this case, for any invertible $h \times h$ matrix $\mathbf{S}$, we have that $g_\theta(p) = \mathbf{S} g_{\theta'}(p)$ where θ' is the same as θ except that the weights of the final FC layer are pre-multiplied by $\mathbf{S}^{-1}$. Hence, if g_θ has a final FC layer, it is impossible to distinguish between the embedding projection layer $\mathbf{W}$ acting on g_θ and the embedding projection layer $\mathbf{W} \cdot \mathbf{S}$ acting on $g_{\theta'}$, given access to the output of the API $\mathcal{O}$ only.

C.2. Proof of Lemma 4.2(ii) With Residual Layers

More generally, if g_θ is composed of residual layers but no normalization layers then $g_\theta(p) = \sum_i L_i(p)$, where $L_i(p)$ is the output of the ith residual layer in the model, ignoring the skip connection (Elhage et al., 2021; Veit et al., 2016). Assume also that each L_i has a final layer that is a fully connected linear layer and a linear input layer (this assumption is true for both attention and MLP modules in transformers without normalization layers). Constructing θ' such that each L_i has input weights pre-multiplied by $\mathbf{S}^{-1}$ and output FC weights multiplied by $\mathbf{S}$, we have $g_{\theta'}(p) = \sum_i \mathbf{S} L_i(p) = \mathbf{S} \cdot g_\theta(p)$ by linearity. Finally, by using a new embedding projection matrix $(\mathbf{S}^{-1})^\top \cdot \mathbf{W}^\top$ and calculating

$$((\mathbf{S}^{-1})^\top \cdot \mathbf{W}^\top)^\top \cdot g_{\theta'}(p) = \mathbf{W} \cdot g_\theta(p), \tag{3}$$

we have shown that logit outputs are identical and so again we cannot distinguish these transformers by querying $\mathcal{O}$ and $\mathcal{O}'$ alone.

C.3. Normalization Layers and Orthogonal Matrices

In Sections C.3-C.6 we can no longer use general invertible matrices $\mathbf{S}$ in our arguments, and must instead use orthogonal matrices, matrices $\mathbf{U}$ such that $\mathbf{U}^\top \mathbf{U} = I$. In models with LayerNorm, we specialise further, too (Appendix C.5).

Lemma C.1. *The RMSNorm operation is equal to $x \mapsto \mathbf{W} n(x) + b$ where $\mathbf{W}$ is a diagonal matrix.*

Proof. RMSNorm is conventionally written as

$$x \mapsto \frac{w \cdot x}{\sqrt{\frac{1}{h} \sum_i x_i^2}} + b \tag{4}$$

where w is multiplied elementwise by normalized x. Clearly this can be written as a diagonal matrix. Further, we can multiply this diagonal matrix by $\sqrt{h}$ to cancel that factor in the denominator of Equation (4). Since $n(x) = x/||x|| = x \Big/ \sum_i \sqrt{x_i^2}$ we get the result. $\square$

Intuitively, the proof in Appendix C.2 relied on pre-multiplying the input projection weight of layers by a matrix $\mathbf{S}^{-1}$, so that this cancelled the rotation $\mathbf{S}$ applied to the model's hidden state (called the 'residual stream' in mechanistic interpretability literature (Elhage et al., 2021)). Formally, if we let the input projection layer be $\mathbf{M}$, we were using the fact that $\left(\mathbf{M}\mathbf{S}^{-1}\right)(\mathbf{S}x) = \mathbf{M}x$. However, since models with normalization layers use these before the linear input projection, the result of applying $\mathbf{S}$ to the hidden state, if we apply the same procedure, produces the activation

$$(\mathbf{M}\mathbf{S}^{-1})(\mathbf{W}n(\mathbf{S}x) + b) \tag{5}$$

but since in general n and $\mathbf{S}$ do not commute, we cannot conclude that the $\mathbf{S}$ transformations preserve the transformer's outputs. We will show that if we take $\mathbf{S} = \mathbf{U}$ an orthogonal matrix, then we still get a general impossibility result.

To do this, we will need a simple result from linear algebra:

Lemma C.2. *Let $x \in \mathbb{R}^h$. Then the normalization map $n(x) := \frac{x}{||x||}$ commutes with orthogonal matrices* $\mathbf{U}$.

Proof of Lemma C.2. We need to show that $\frac{\mathbf{U}x}{||x||} = \frac{\mathbf{U}x}{||\mathbf{U}x||}$. This is true since $x^\top \mathbf{U}^\top \mathbf{U}x = x^T x$, so $||\mathbf{U}x|| = ||x||$. $\square$

C.4. Proof of Lemma 4.2(ii) in Models With RMSNorm

In Lemma C.2, we showed that orthogonal matrices $\mathbf{U}$ commute with normalization. Hence if we multiply all layer output weights by $\mathbf{U}$, but pre-multiply all layer input projection weights by $\mathbf{W}\mathbf{U}^\top\mathbf{W}^{-1}$, then the effect of the linear projection layer is

$$(\mathbf{M}\mathbf{W}\mathbf{U}^\top\mathbf{W}^{-1})(\mathbf{W}n(\mathbf{U}x) + b) = (\mathbf{M}\mathbf{W}\mathbf{U}^\top\mathbf{W}^{-1})(\mathbf{W}\mathbf{U}n(x) + b) = \mathbf{M}(\mathbf{W}n(x) + b) \tag{6}$$

which is identical to the original model. Applying this procedure to all layers added to the hidden state (using the different W diagonal matrices each time) gives us a model $g_{\theta'}(p)$ such that $g_{\theta'}(p) = \mathbf{U}g_{\theta'}(p)$ so a different embedding projection matrix $\mathbf{W}\mathbf{U}^\top$ will give identical outputs to the original model $g_\theta(p)$ (with embedding projection $\mathbf{W}$).

Note that we ignore what happens to b in the above arguments, since any sequence of affine maps applied to a constant $b \in \mathbb{R}^h$ yields a constant $b' \in \mathbb{R}^h$, and we can just use b' instead of b in $g_{\theta'}$.

C.5. Proof of Lemma 4.2(ii) in Models With LayerNorm

The LayerNorm operation is the composition of a centering operation $x \mapsto x - \bar{x}$ with RMSNorm (i.e. first centering is applied, then RMSNorm). Therefore the identical argument to Appendix C.4 goes through, besides the fact that we need $\mathbf{U}$ to also commute with the centering operation. Since the centering operation fixes a $(h-1)$ dimensional subspace defined by $\mathbf{1}^T x = 0$ where $\mathbf{1} \in \mathbb{R}^h$ is the vector of ones, it is enough to impose an additional condition that $\mathbf{U}\,\mathbf{1} \in \{-\mathbf{1}, \mathbf{1}\}$.

C.6. Proof of Lemma 4.2(ii) in Models With Normalization $\varepsilon \neq 0$

We now extend to realistic models where the ε in the denominator of LayerNorm is not 0. We can do this because the only fact we used about $x \mapsto n(x)$ was that $x \mapsto n(\mathbf{U}x)$ was identical to $x \mapsto \mathbf{U}n(x)$. In turn Lemma C.2 relied on $||\mathbf{U}x|| = ||x||$ due to orthogonality. But adjusting $n(x)$ to $n'(x) := x \Big/ \sqrt{\frac{1}{h}||x||^2 + \varepsilon}$ (i.e. normalization with an epsilon), since $||x|| = ||\mathbf{U}x||$, n' commutes with $\mathbf{U}$, and so the proofs in Appendix C.4 and Appendix C.5 still work when using n' instead of n.

Therefore finally, we have proven the impossibility result Lemma 4.2(ii) in all common model architectures (all non-residual networks that end with dense layers, and all transformers from Biderman (2024)). $\square$

D. Derivation of Binarized Logprob Extraction (Section 5.4)

To begin, observe that we can write

$$y_{\text{top}} = \mathsf{logit}_{\text{top}} - \log \sum_i \exp(\mathsf{logit}_i)$$
$$y'_{\text{top}} = \mathsf{logit}_{\text{top}} - \log \big(\exp(\mathsf{logit}_t - 1) + \sum_{i \neq t} \exp(\mathsf{logit}_i) \big)$$

Let $\mathbf{N} = \sum_i \exp(\mathsf{logit}_i)$ and $p = \exp(\mathsf{logit}_t) / \mathbf{N}$. Then, we can rewrite

$$y_{\text{top}} = \mathsf{logit}_{\text{top}} - \log \mathbf{N}$$
$$y_{\text{top}} = \mathsf{logit}_{\text{top}} - \log(\mathbf{N} + (1/e - 1)p\mathbf{N})$$

Subtracting the two, we get

$$y_{\text{top}} - y'_{\text{top}} = \log(1 + (1/e - 1)p)$$
$$\implies p = \frac{\exp(y_{\text{top}} - y'_{\text{top}}) - 1}{1/e - 1}.$$

Related work. Concurrent work (Morris et al., 2023) discusses a similar but weaker two-query logprob extraction. Their attack requires a logit bias larger than $\mathsf{logit}_{\text{top}} - \mathsf{logit}_i$ and top-2 logprob access; our attack works as soon the logit bias is allowed to be nonzero, and with top-1 logprob access.

E. Efficient Recovery of Logits From Top k Logprobs APIs

In Section 5.3 of the main body, we presented a simple and practical method for extracting the entire logits vector via multiple queries to an API that only provides the top few logprobs and accepts a logit bias with each query. In this section we present more efficient methods.

The method we presented earlier uses a reference token. We set this to some arbitrary value (e.g., 0) and then compare the logits for all other tokens to this one. This approach is numerically stable, but is slightly wasteful: of the top K logprobs returned by the API, one is always the reference token. Hence, we only recover $K - 1$ logits per query with this method.

In this appendix, we present linear algebraic methods that are able to recover K logits per query to the top-K logprobs API.

Setting: Recall that there is an unknown vector $z = \mathbf{W} \cdot g_\theta(p) \in \mathbb{R}^\ell$ (i.e., the logits for a given prompt p) that we want to recover. We can make multiple queries to the API with the same prompt $\mathcal{O}(p, b)$. Each query is specified by a vector $b \in \mathbb{R}^\ell$ (a.k.a. the logit bias). We receive answers of the form $(i, a_i(z, b)) \in \mathbb{N} \times \mathbb{R}$, where i is a token index and $a_i(z, b)$ is a logprob:

$$a_i(z, b) = \log \left(\frac{\exp(z_i + b_i)}{\sum_j^\ell \exp(z_j + b_j)} \right) = z_i + b_i - \log \left(\sum_j^\ell \exp(z_j + b_j) \right). \quad (7)$$

Each query may receive multiple answers (namely, the K largest $a_i(z, b)$ values). For notational simplicity, we denote multiple answers to one query the same way as multiple queries each returning one answer. Suppose queries $b^1, \cdots, b^m$ were asked and we received m answers $(i_1, a_{i_1}(z, b^1)) \leftarrow \mathcal{O}(p, b^1), \cdots, (i_m, a_{i_m}(z, b^m)) \leftarrow \mathcal{O}(p, b^m)$.

Our goal is to compute z from the answers $a_i(z, b)$.

E.1. Warmup: Single Logprob API ($K = 1$)

As a starting point, suppose the API only returns the single largest logprob (i.e., $K = 1$). The approach from Section 5.3 cannot work in this setting because we cannot obtain the logprob of both the reference token and another token at the same time, meaning we can recover less than 1 logit per query.

The high-level idea to overcome this problem is that, instead of normalizing logits relative to a reference token, we shall normalize the logits to be logprobs. That is, we recover the logits with the normalization $\sum_j \exp(z_j) = 1$. With this normalization it is no longer necessary to include a reference token in every query.

Fix a token index i and let $b_i = B$ and $b_j = 0$ for all $j \neq i$. We query the API with this logit bias and assume that B is large enough that token i is returned:

$$(i, a_i(z, b)) \leftarrow \mathcal{O}(p, b).$$

From Equation 7,

$$\begin{aligned}
a_i(z,b) &= z_i + b_i - \log\left(\sum_j^{\ell} \exp(z_j + b_j)\right)\\
&= z_i + B - \log\left(\exp(z_i + B) + \sum_{j\neq i} \exp(z_j)\right)\\
&= z_i + B - \log\left(\exp(z_i + B) - \exp(z_i) + \sum_j^{\ell} \exp(z_j)\right),\\
\implies z_i + B - a_i(z,b) &= \log\left(\exp(z_i + B) - \exp(z_i) + \sum_j^{\ell} \exp(z_j)\right),\\
\implies \exp(z_i + B - a_i(z,b)) &= \exp(z_i + B) - \exp(z_i) + \sum_j^{\ell} \exp(z_j),\\
\implies \exp(z_i + B - a_i(z,b)) - \exp(z_i + B) + \exp(z_i) &= \sum_j^{\ell} \exp(z_j),\\
\implies \exp(z_i) \cdot (\exp(B - a_i(z,b)) - \exp(B) + 1) &= \sum_j^{\ell} \exp(z_j),\\
\implies \exp(z_i) &= \frac{\sum_j^{\ell} \exp(z_j)}{\exp(B - a_i(z,b)) - \exp(B) + 1},\\
\implies z_i &= \log\left(\sum_j^{\ell} \exp(z_j)\right) - \log\left(\exp(B - a_i(z,b)) - \exp(B) + 1\right).
\end{aligned}$$

Thus if we normalize $\sum_j^{\ell} \exp(z_j) = 1$, we have

$$z_i = -\log\left(\exp(B - a_i(z,b)) - \exp(B) + 1\right). \tag{8}$$

E.2. Recovering K Logits From K Logprobs

The approach from the previous subsection extends to the setting where each API query returns the top K logprobs. In practice we work with $K = 5$. We are able to recover K logits. Again, instead of using a reference token to normalize the logits, we will normalize $\sum_j \exp(z_j) = 1$. However, in this setting we will need to solve a K-by-K system of linear equations.

Fix K token indices $i_1, \cdots, i_K$ and let $b_{i_k} = B$ for $k \in \{1, \cdots, K\}$ and $b_j = 0$ for all $j \notin \{i_1, \cdots, i_K\}$. We query the API with this logit bias and assume that B is large enough that the logprobs for $i_1, \cdots, i_K$ are returned as the top K logprobs:

$$(i_1, a_{i_1}(z,b)), (i_2, a_{i_2}(z,b)), \cdots, (i_K, a_{i_K}(z,b)) \leftarrow \mathcal{O}(p, b).$$

Let $z \in \mathbb{R}^{\ell}$ be the (unknown) logits and let $\mathbf{N} = \sum_i \exp(z_i)$ be the normalizing constant. For each $k \in \{1, \cdots, K\}$, we have

$$\begin{aligned}
a_{i_k}(z,b) &= z_{i_k} + B - \log\left(\sum_{i\in\{i_1,\cdots,i_K\}} \exp(z_i+B) + \sum_{i\notin\{i_1,\cdots,i_K\}} \exp(z_i)\right)\\
&= z_{i_k} + B - \log\left((e^B-1)\sum_{i\in\{i_1,\cdots,i_K\}} \exp(z_i) + \sum_{i}^{\ell} \exp(z_i)\right)\\
&= z_{i_k} + B - \log\left((e^B-1)\sum_{i\in\{i_1,\cdots,i_K\}} \exp(z_i) + \mathbf{N}\right),\\
\implies z_{i_k} + B - a_{i_k}(z,b) &= \log\left((e^B-1)\sum_{i\in\{i_1,\cdots,i_K\}} \exp(z_i) + \mathbf{N}\right),\\
\implies \exp(z_{i_k} + B - a_{i_k}(z,b)) &= (e^B-1)\sum_{i\in\{i_1,\cdots,i_K\}} \exp(z_i) + \mathbf{N},
\end{aligned}$$

And therefore we can conclude

$$\exp(B - a_{i_k}(z,b))\cdot\exp(z_k) - (e^B-1)\sum_{i\in\{i_1,\cdots,i_K\}} \exp(z_i) = \mathbf{N}.$$

This linear system of equations can be expressed in matrix form:

$$A\cdot\begin{pmatrix}\exp(z_{i_1})\\ \exp(z_{i_2})\\ \vdots\\ \exp(z_{i_K})\end{pmatrix} = \begin{pmatrix}\mathbf{N}\\ \mathbf{N}\\ \vdots\\ \mathbf{N}\end{pmatrix},$$

where A is a $K\times K$ matrix with entries

$$A_{k,j} = \begin{cases}\exp(B - a_{i_k}(z,b)) - (e^B-1) & \text{if } j=k\\ -(e^B-1) & \text{if } j\neq k.\end{cases}$$

Note that A is a rank-one perturbation of a diagonal matrix, that is, if $\mathbf{1}$ is the all-ones vector, then

$$A = \mathrm{diag}_{1\le k\le K}(\exp(B - a_{i_k}(z,b))) - (e^B-1)\mathbf{1}\mathbf{1}^T,$$

where $\mathrm{diag}_{1\le k\le K}(\exp(B - a_{i_k}(z,b)))$ denotes a diagonal matrix with the k-th diagonal entry being $\exp(B - a_{i_k}(z,b))$. Inverting a diagonal matrix is easy and thus we can use the Sherman-Morrison formula to compute the inverse of A:

$$\begin{aligned}
A^{-1} &= \mathrm{diag}_{1\le k\le K}(\exp(a_{i_k}(z,b) - B))) + (e^B-1)\frac{\mathrm{diag}_{1\le k\le K}(\exp(a_{i_k}(z,b)-B)))\mathbf{1}\mathbf{1}^T\mathrm{diag}_{1\le k\le 5}(\exp(a_{i_k}(b)-B)))}{1-(e^B-1)\mathbf{1}^T\mathrm{diag}_{1\le k\le 5}(\exp(a_{i_k}(b)-B)))\mathbf{1}}\\
&= \mathrm{diag}(v) + (e^B-1)\frac{vv^T}{1-(e^B-1)\mathbf{1}^T v},
\end{aligned}$$

where $v \in \mathbb{R}^K$ is the vector with entries $v_k = \exp(a_{i_k}(z,b) - B)$. Hence

$$\begin{aligned}
\begin{pmatrix} \exp(z_{i_1}) \\ \exp(z_{i_2}) \\ \vdots \\ \exp(z_{i_K}) \end{pmatrix} &= A^{-1} \cdot \begin{pmatrix} \mathbf{N} \\ \mathbf{N} \\ \vdots \\ \mathbf{N} \end{pmatrix} \\
&= \left(\mathrm{diag}(v) + (e^B - 1) \frac{vv^T}{1 - (e^B - 1)\mathbf{1}^T v} \right) \cdot \mathbf{1} \cdot \mathbf{N} \\
&= \left(v + \frac{(e^B - 1)vv^T\mathbf{1}}{1 - (e^B - 1)\mathbf{1}^T v} \right) \cdot \mathbf{N} \\
&= \left(1 + \frac{(e^B - 1)\mathbf{1}^T v}{1 - (e^B - 1)\mathbf{1}^T v} \right) \cdot \mathbf{N} \cdot v \\
&= \frac{\mathbf{N}}{1 - (e^B - 1)\sum_j v_j} \cdot v, \\
\implies z_{i_k} &= \log\left(A^{-1}\mathbf{1}\mathbf{N}\right)_k \\
&= \log\left(\frac{\mathbf{N} v_k}{1 - (e^B - 1)\sum_j^K v_j} \right) \\
&= \log\left(\frac{\mathbf{N}\exp(a_{i_k}(z,b) - B)}{1 - (e^B - 1)\sum_j^K \exp(a_{i_j}(z,b) - B)} \right) \\
&= \log \mathbf{N} + a_{i_k}(z,b) - B - \log\left(1 - (e^B - 1)\sum_j^K \exp(a_{i_j}(z,b) - B) \right) \\
&= \log \mathbf{N} + a_{i_k}(z,b) - B - \log\left(1 - (1 - e^{-B})\sum_j^K \exp(a_{i_j}(z,b)) \right).
\end{aligned}$$

If we normalize $\mathbf{N} = 1$, this gives us a formula for computing the logits:

$$z_{i_k} = a_{i_k}(z,b) - B - \log\left(1 - (1 - e^{-B})\sum_j^K \exp(a_{i_j}(z,b)) \right). \tag{9}$$

Note that setting $K = 1$ yields the same result as in Equation 8.

Recovery using Equation 9 is more efficient than the method in Section 5.3, as we recover K logits $z_{i_1}, z_{i_2}, \cdots, z_{i_K}$ rather than just $K - 1$ logits. However, if B is large, numerical stability may be an issue. (And, if B is small, the logit bias may be insufficient to force the API to output the desired tokens by placing them in the top K.) Specifically, as $B \to \infty$, we have $(1 - e^{-B})\sum_j^K \exp(a_{i_j}(z,b)) \to 1$ and so the logarithm in Equation 9 tends to $\log(1 - 1) = -\infty$; this means we may have catastrophic cancellation.

Related work. Two works published during the responsible disclosure period use a similar procedure, and deal with numerical issues in different ways. (Chiu, 2024) start with a low B for the whole vocabulary, then increase B and ask for all tokens that haven't appeared before, and repeat until all tokens are covered. (Hayase et al., 2024) use the method in Appendix E.1, and set $B = -\hat{z}_i$, where $\hat{z}_i$ is an estimate of z_i inherent to their application. It is possible variants of this method have been discussed before our or these works, but we are not aware of further references.

E.3. General Method

In general, we may not have have full control over which logprobs the API returns or which logit bias is provided to the API. Thus we generalize the linear algebraic approach above to reconstruct the logits from arbitrary logit biases and tokens.

Suppose queries $b^1, \cdots, b^m$ were asked and we received m answers $(i_1, a_{i_1}(z, b^1)) \leftarrow \mathcal{O}(p, b^1), \ldots, (i_m, a_{i_m}(z, b^m)) \leftarrow \mathcal{O}(p, b^m)$. (If a query returns multiple answers, we can treat this the same as multiple queries each returning one answer.)

As before, rearranging Equation 7 gives the following equations.

$$\forall k \in [m] \quad \exp(a_{i_k}(z, b^k_{i_k})) = \frac{\exp(z_{i_k} + b^k_{i_k})}{\sum_j^\ell \exp(z_j + b^k_j)}.$$

$$\forall k \in [m] \quad \sum_j^\ell \exp(z_j + b^k_j) = \exp(z_{i_k} + b^k_{i_k} - a_{i_k}(z, b^k)).$$

$$\forall k \in [m] \quad \sum_j^\ell \exp(z_j) \cdot \exp(b^k_j) = \exp(z_{i_k}) \cdot \exp(b^k_{i_k} - a_{i_k}(z, b^k)).$$

$$\forall k \in [m] \quad \sum_j^\ell \Big(\exp(b^k_j) - \mathbb{I}[j = i_k] \cdot \exp(b^k_{i_k} - a_{i_k}(z, b^k))\Big) \cdot \exp(z_j) = 0.$$

$$A \cdot \begin{pmatrix} \exp(z_1) \\ \exp(z_2) \\ \vdots \\ \exp(z_\ell) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

$$\text{where} \quad \forall k \in [m] \, \forall j \in [\ell] \quad A_{k,j} = \exp(b^k_j) \cdot \Big(1 - \mathbb{I}[j = i_k] \cdot \exp(-a_{i_k}(z, b^k))\Big).$$

Here $\mathbb{I}[j = i_k]$ is 1 if $j = i_k$ and 0 otherwise. If A is invertible, then this linear system can be solved to recover the logits z. Unfortunately, A is not invertible: Indeed, we know that the solution cannot be unique because shifting all the logits by the same amount yields the exact same answers $a_i(z, b) = a_i(z + \mathbf{1}, b)$. That is, we expect a one-dimensional space of valid solutions to $A \cdot \exp(z) = \mathbf{0}$. To deal with this we simply add the constraint that $z_1 = 0$ or, equivalently, $\exp(z_1) = 1$. This corresponds to the system

$$\widehat{A} \cdot \exp(z) = \begin{pmatrix} 1 \; 0 \; \cdots \; 0 \\ A \end{pmatrix} \cdot \begin{pmatrix} \exp(z_1) \\ \exp(z_2) \\ \vdots \\ \exp(z_\ell) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

(We could also normalize $\sum_i^\ell \exp(z_i) = 1$. This corresponds to the first row of $\widehat{A}$ being all 1s instead of one 1.) This is solvable as long as the augmented matrix has a nonzero determinant

$$\det\left(\widehat{A}\right) = \det \begin{pmatrix} 1 \; 0 \; \cdots \; 0 \\ A \end{pmatrix} = \det(A_{1:m,2:\ell}). \tag{10}$$

Here $A_{1:m,2:d}$ denotes A with the first column removed. Note that we are setting $m = \ell - 1$. This is the minimum number of query-answer pairs that we need. If we have more (i.e., $m \geq \ell$), then the system is overdetermined. Having the system be overdetermined is a good thing; the extra answers can help us recover the logprobs with greater accuracy. The least squares solution to the overdetermined system is given by

$$\widehat{A}^T \widehat{A} \cdot \begin{pmatrix} \exp(z_1) \\ \exp(z_2) \\ \vdots \\ \exp(z_\ell) \end{pmatrix} = \widehat{A}^T \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \tag{11}$$

This provides a general method for recovering the (normalized) logits from the logprobs API.

F. How Far Are Our Logprob-Free Attacks From Optimal?

In the logprob-free API, we have produced attacks capable of recovering logits and ultimately the embedding hidden dimension and embedding matrix up to a similarity transform. We now provide lower bounds on the minimum number of queries required by *any* attacker attempting model stealing under the logprob-free API threat model.

Lemma F.1. *Assume the entries of* $\mathsf{logit} \in \mathbb{R}^l$ *are i.i.d. uniform over* $[-B, 0]$. *To recover the vector* logit *up to* ∞*-norm error* ε, *the number of queries to* $\mathcal{O}(p, \cdot)$ *we need is at least:*

$$\frac{l \log_2(B/\varepsilon)}{\log_2(l)}.$$

Proof. The information content of a single logit value in $[-B, 0]$ up to ∞-norm error ε is $\log_2(B/\varepsilon)$, assuming a uniform prior over ε-spaced points in the interval. Since the logits are independent, the information encoded in l logit values up to ∞-norm error ε is $l \log_2(100/\varepsilon)$.

Any single query to $\mathcal{O}$, no matter how well-crafted, yields at most $\log_2(l)$ bits, because the output is one of l distinct values. The minimum number of queries required is at least the total information content divided by the information per query, yielding the lower bound $l \log_2(B/\varepsilon) / \log_2(l)$. □

The restriction of biasing at most N tokens at a time gives us a lower bound of

$$\frac{l \log_2(B/\varepsilon)}{\log_2(N)}$$

queries, which is a factor of $\log_2(l) / \log_2(N)$ worse. For $N = 300$ and $l \approx$ 100,000, this is only a factor of 2.

For $B = 100$ and $N = 300$, we thus need at least

$$\frac{\log_2(B/\varepsilon)}{\log_2(N)} \approx 0.81 + 0.12 \log_2(1/\varepsilon)$$

queries per logit. If we want between 6 and 23 digits of precision, the lower bound corresponds to 1.53 to 3.57 queries per logit. We see that the best logprob-free attack in Table 3 is only about 1 query per logit worse than the lower bound.

The main unrealistic assumption in Lemma F.1 is that the prior over the logit values is i.i.d. uniform over an interval. A better assumption might be that most of the logit values come from a light-tailed unimodal distribution. We leave more realistic lower bounds and attacks that make use of this better prior to future work.

G. Recovering W up to an orthogonal matrix

In this section, we outline an algorithm for extracting $\mathbf{W}$ up to an orthogonal $h \times h$ matrix, instead of merely up to a nonsingular $h \times h$ matrix as in Appendix C. We do **not** carry out this attack in practice for models considered in this paper, and leave improving this algorithm as an open problem for future work.

We make a few simplifying assumptions:

1. We merge the final LayerNorm weights γ into $\mathbf{W}$ by linearity.[3]
2. We assume the numerical precision is high enough that after the final LayerNorm, the hidden states are on a sphere.
3. There is no degenerate lower-dimensional subspace containing all $g_\theta(p)$ for all our queries p.
4. We assume the ε in RMSNorm/LayerNorm is 0. This is not a critical assumption.

Again, we use the compact SVD on the query output matrix $\mathbf{Q} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^\top$. Here $\mathbf{Q} \in \mathbb{R}^{l \times n}$, $\mathbf{U} \in \mathbb{R}^{l \times h}$, $\mathbf{\Sigma} \in \mathbb{R}^{h \times h}$, and $\mathbf{V}^\top \in \mathbb{R}^{h \times n}$. Note that the points $g_\theta(p)$ lie on a sphere in $\mathbb{R}^h$, and $\mathbf{U}^\top \cdot \mathbf{W} \in \mathbb{R}^{h \times h}$, hence $\mathbf{U}^\top \cdot \mathbf{W} \cdot g_\theta(p)$ lie on an ellipsoid in $\mathbb{R}^h$. From now on, it is convenient to work with the points $x_i = \mathbf{U}^\top \cdot \mathbf{W} \cdot g_\theta(p_i)$; note that we can compute these points directly, because we know both $\mathbf{U}$ and $\mathbf{W} \cdot g_\theta(p_i)$. As ellipsoids are equivalently defined by $x^\top \mathbf{A} x = 1$ for some positive semidefinite (symmetric) matrix $\mathbf{A} \in \mathbb{R}^{h \times h}$, this implies the following statement:

Lemma G.1. *There exists a positive semidefinite* $\mathbf{A} \in \mathbb{R}^{h \times h}$ *such that* $x_i^\top \mathbf{A} x_i = 1$ *for all* i.

[3]For a full explanation of this method of rewriting the unembedding matrix, see Appendix A.1, 'Folding LayerNorm' in Gurnee et al. (2024).

Because $\mathbf{A}$ is positive semidefinite, we can write $\mathbf{A} = \mathbf{M}^\top \cdot \mathbf{M}$ for some $\mathbf{M}$. Here is the key observation:

Lemma G.2. $\mathbf{W} = \mathbf{U} \cdot \mathbf{M}^{-1} \cdot \mathbf{O}$ *for some orthogonal matrix* $\mathbf{O}$.

Proof. We know that $g_\theta(p_i)$ lie on a sphere. The equation $x_i^\top \mathbf{A} x_i = 1$ is equivalent to $x_i^\top \mathbf{M}^\top \mathbf{M} x_i = 1$, which is equivalent to $\|\mathbf{M}x_i\| = 1$. This means that $\mathbf{M}x_i$ lie on a sphere. Because $\mathbf{M}x_i = \mathbf{M} \cdot \mathbf{U}^\top \cdot \mathbf{W} \cdot g_\theta(p_i)$, we have that $\mathbf{M} \cdot \mathbf{U}^\top \cdot \mathbf{W}$ is a norm-preserving transformation on the points $g_\theta(p_i)$. By the assumption that $g_\theta(p_i)$ are not in a degenerate lower-dimensional subspace, we have that $\mathbf{M} \cdot \mathbf{U}^\top \cdot \mathbf{W} =: \mathbf{O}$ is a norm-preserving endomorphism of $\mathbb{R}^h$, hence an orthogonal matrix. This directly implies $\mathbf{W} = \mathbf{U} \cdot \mathbf{M}^{-1} \cdot \mathbf{O}$ as claimed. □

This means that, to extract $\mathbf{W}$ up to an orthogonal matrix, it's enough to recover some $\mathbf{M}$ satisfying $\|\mathbf{M}x_i\| = 1$ for all i. To compute $\mathbf{M}$, we can actually recover the positive semi-definite $\mathbf{A}$ satisfying Lemma G.1, which would give us a feasible $\mathbf{M}$ by SVD or Cholesky decomposition.

The final observation is that the system in Lemma G.1 ($x_i^\top \mathbf{A} x_i = 1$ for all i) is linear in the $h(h+1)/2$ distinct entries of $\mathbf{A}$. By generating more than $h(h+1)/2$ values for x_i (and again assuming no degeneracy), we can therefore solve a large system of linear equations, and in principle recover $\mathbf{A}$, and hence $\mathbf{M}$, and hence $\mathbf{W}$. Note that any solution $\mathbf{A}$ will be positive semidefinite because an overdetermined linear system has at most one solution. However, we do not know how to solve these systems of linear equations in h^2 variables efficiently ($h > 750$ in all our experiments); so in practice we resort to reconstructing weights up to an arbitrary $h \times h$ matrix, as described in Appendix C.

H. Quantization and Noise

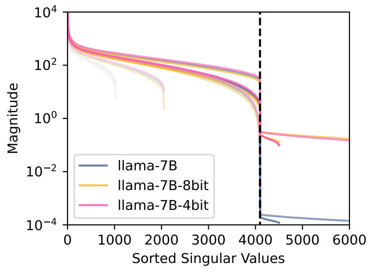
H.1. Quantization

Quantization is a popular strategy for decreasing a model's memory footprint and speeding up inference. In addition to these benefits, using lower-precision number representations also effectively adds noise. As noted in Section 8.2, adding noise to the output logits could prevent our attack. A natural question that follows is, does quantization add sufficient noise to make our attack ineffective or more difficult to carry out?

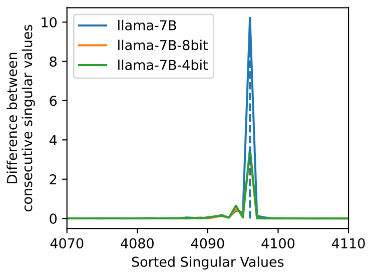
For a simple test, we quantize Llama-7B at both 8-bits and 4-bits, and compare our baseline attack (Section 4.1) to the default 16-bit implementation. We quantize using `bitsandbytes` (Dettmers et al., 2022), which HuggingFace supports for out-of-the-box quantization of model weights and lower-precision inference (Figure 6). We observe no meaningful differences at different levels of quantization; querying each model results in recovering the same same embedding matrix dimension h in the same number of queries. Given that 8-bit and 4-bit quantization are generally observed to not have a large impact on performance, this is perhaps an unsurprising result; any noise from quanitization does not seem to have a meaningful impact on the logits (in the context of our attack).

H.2. Noise

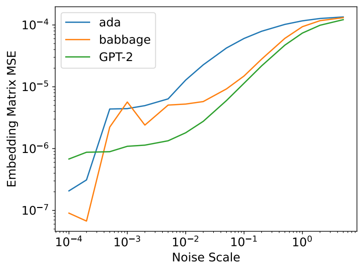
One natural defense to our attacks is to obfuscate the logits by adding noise. This will naturally induce a tradeoff between utility and vulnerability—more noise will result in less useful outputs, but increase extraction difficulty. We empirically measure this tradeoff in Figure 5(c). We consider noise added directly to the logits, that is consistent between different queries of the same prompt. To simulate this, we directly add noise to our recovered logits, and recompute the extracted embedding matrix. For GPT-2, we measure the RMSE between the true embedding matrix and the embedding matrix extracted with a specific noise level; for `ada` and `babbage`, we measure the RMSE between the noisy extracted weights and the weights we extracted in the absence of noise. We normalize all embedding matrices (to have ℓ_2 norm 1) before measuring RMSE.



(a). Sorted singular values for $\{1024, 2048, 4096, 8192\}$ queries.



(b). Differences between consecutive sorted singular values.



(c). RMSE of extracted embeddings at various noise variances.

Figure 6. In (a, b), recovering the embedding matrix dimension h for Llama-7B at different levels of precision: 16-bit (default), 8-bit, and 4-bit. We observe no meaningful differences, with respect to our attack, at different levels of quantization. In (c), the RMSE between extracted embeddings as a function of the standard deviation of Gaussian noise added to the logits.

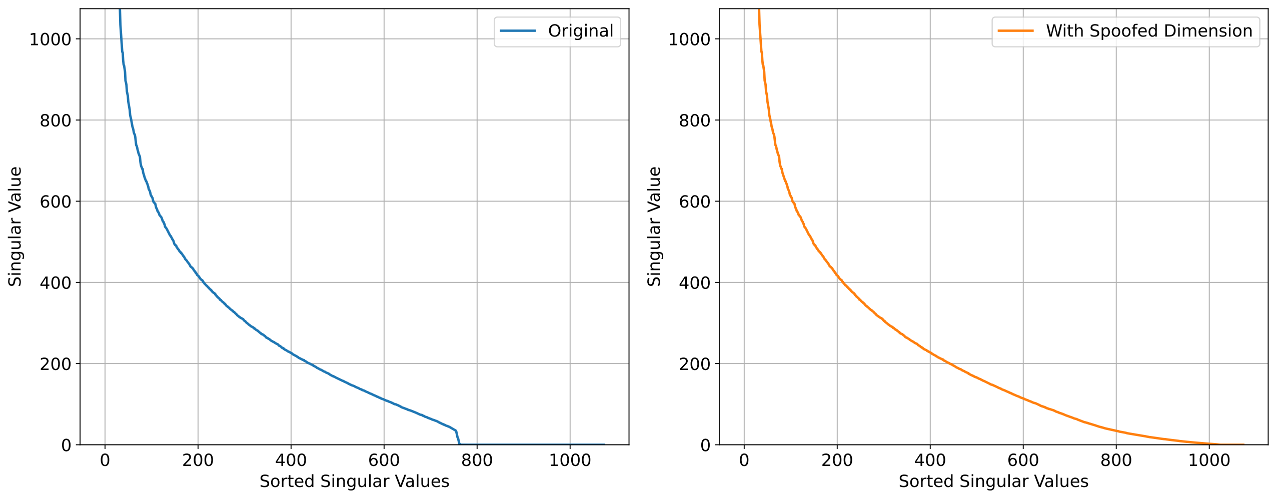


Figure 7. On the left, we plot the singular values that are extracted using our attack on GPT-2 small—the estimated hidden dimension is near 768. On the right, we post-hoc extend the dimensionality of the weight matrix to 1024, as described in Section 8. This misleads the adversary into thinking the model is wider than it actually is.