FlexModel: A Framework for Interpretability of Distributed Large Language Models

Matthew Choi,* **Muhammad Adil Asif,*** **John Willes, David B. Emerson**
Vector Institute, Toronto, ON, Canada

Abstract

With the growth of large language models, now incorporating billions of parameters, the hardware prerequisites for their training and deployment have seen a corresponding increase. Although existing tools facilitate model parallelization and distributed training, deeper model interactions, crucial for interpretability and responsible AI techniques, still demand thorough knowledge of distributed computing. This often hinders contributions from researchers with machine learning expertise but limited distributed computing background. Addressing this challenge, we present `FlexModel`, a software package providing a streamlined interface for engaging with models distributed across multi-GPU and multi-node configurations. The library is compatible with existing model distribution libraries and encapsulates PyTorch models. It exposes user-registerable `HookFunctions` to facilitate straightforward interaction with distributed model internals, bridging the gap between distributed and single-device model paradigms. Primarily, `FlexModel` enhances accessibility by democratizing model interactions and promotes more inclusive research in the domain of large-scale neural networks. The package is found at https://github.com/VectorInstitute/flex_model.

1 Introduction

Driven by recent advances in language modelling, neural network sizes have increased dramatically [14, 28]. Large language models (LLMs), with billions of parameters, have demonstrated impressive and surprising abilities, often attributed to their generalization and in-context learning capacity [24, 11, 6]. Increased model size has resulted in a commensurate increase in hardware prerequisites for training and deployment of these models, frequently requiring distributed infrastructure. To meet this need, many technologies have emerged [28, 33, 26]. While these tools abstract numerous challenges associated with model parallelization and distributed training, non-standard interactions and deeper model manipulations, such as intermediate activation retrieval or model editing, still necessitate a comprehensive understanding of distributed computing. However, such interactions are important elements of many techniques in interpretability and responsible AI [8, 21, 3, 9]. For researchers lacking foundational experience in distributed computing, this represents a significant barrier. As such, a large cohort of researchers find themselves at a disadvantage or unable to contribute, despite possessing valuable machine learning expertise, due to the technical complexities inherent in working with distributed models.

In response to this challenge, this paper introduces `FlexModel`, a software package engineered to deliver a lightweight and user-friendly interface for interacting with large-scale models deployed in multi-GPU and multi-node settings. The library is designed to seamlessly integrate with existing technologies employed by researchers and practitioners, ensuring compatibility and convenience. `FlexModel` wraps PyTorch models deployed using any of the major libraries, including Accelerate, Fully Sharded Data Parallel (FSDP), and DeepSpeed, and supports user-defined `HookFunctions` to

*Correspondence to {matthew.choi, adil.asif}@vectorinstitute.ai

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

enable straightforward interaction with distributed model internals during both forward and backward passes. These `HookFunctions` abstract the required distributed communication and allow researchers to interact with distributed models as if they were running on a single device.

The contributions of this work are summarized as:

- `FlexModel`: A software package which provides an infrastructure-agnostic model interface to enable researchers to perform interpretability and responsible AI research at scale without a deep understanding of distributed systems. The package aligns distributed model interaction with the simpler paradigm of single-device model manipulation.
- Validation of `FlexModel` for use at scale in two experimental settings: Transformer Induction Head Isolation [24] and a TunedLens [3] implementation.

2 Background and Related Work

2.1 Interpretability

With the release of highly performant, closed-source LLMs, such as ChatGPT and BARD, interest in LLMs has rapidly increased. Open-source models, including LLaMA [30, 31], have also gained widespread adoption. The rise of LLMs has also attracted attention from researchers interested in model interpretability and explainability. Characterizing the emergence of capabilities in LLMs during training is crucial to understanding their formation and function in downstream applications [4]. Tracing inference and investigating the influence of context are also key to providing insights into model behaviour. However, such investigations are challenging due to the massive number of parameters and layers of non-linearity present in LLMs.

Typical interpretability and explainability work investigates LLMs on a macroscopic level, where qualitative behaviours are explored. Many such works leverage prompts for such investigation [17, 25, 15, 13, 32]. A more limited set of works are beginning to delve into the internal mechanics of LLMs through probing mechanisms like activation or prompt tuning [2, 29]. Alternatively, mechanistic interpretability develops hypotheses about basic transformer behaviour, derived from first principles. Such approaches aim to more deeply investigate the fundamental components driving LLM generation and inject or train modules to examine the internal representations of these models.

2.2 Distributed Models and Parallelism

Running inference on models with tens or hundreds of billions of parameters often requires multi-GPU coordination to fit the model parameters in GPU DRAM. Training such models requires multi-node techniques to compute and store activations, gradients and optimizer states. Typically these techniques involve splitting or replicating a model along three "dimensions," namely the data parallel (DP), tensor parallel (TP) and pipeline parallel (PP) dimensions (see Figure 5a). DP methods, like PyTorch's `DistributedDataParallel` [16], replicate a model on each GPU, effectively increasing the global batch size. Note that the model memory footprint remains the same on each GPU in this case. Frameworks such as `FullyShardedDataParallel` [33] implement sharded DP to address this limitation, where each GPU only holds a piece of the model states. Computation is performed by sequentially communicating these pieces across all GPUs. Alternatively, TP and PP schemes directly seek to reduce the memory footprint of model states. These schemes shard a model instance across multiple GPUs. TP breaks layer operations into smaller parallel operations distributed over multiple GPUs, but requires expensive all-to-all communication [28]. PP splits the model into sequential stages where forward and backward passes iterate through each stage [12, 28, 22]. The sequential pipeline stages reduce the per-device memory footprint at the cost of increasing GPU idle time due to data-dependencies, known as pipeline bubbles.

2.3 Related Work

Several libraries facilitating model inspection, probing, and interpretability exist. However, each have drawbacks, especially in distributed settings, which `FlexModel` aims to address. TransformerLens [20] provides a variety of features for interpretability research on transformers, such as activation retrieval and editing. However, models must be re-engineered to fit specific requirements, and

features for experimentation in distributed settings are limited. CircuitsVis [7] is a dynamic library for visualizing transformer mechanics, but it assumes access to the target quantities rather than providing probing tools. Finally, Microscope [27] and Neuroscope [19] focus on studying the formation of features and tracing activations in deep learning models. Microscope solely considers vision models, is not publicly available, and only a fixed set of models are admissible. Neuroscope is capable of identifying maximally activating examples for each neuron in several language models. However, the set of supported models is limited with the largest having only 1.4B parameters.

3 FlexModel

Working with LLMs typically involves many painful lessons in distributed systems. Researchers often need to learn the PyTorch distributed back-end and familiarize themselves with many of the distributed model strategies described in Section 2.2. Performing interpretability research in such settings, where model surgery and inspection requires opening the black box, is even more challenging. `FlexModel` lowers these barriers to interpretability research in LLMs and beyond. The design goals of the `FlexModel` API are two-fold:

1. **Intuitive:** Applying the `FlexModel` wrapper to a PyTorch `nn.Module` should simply add features for model inspection to the target model. Unwrapping the model should produce the original model without side-effects. The `HookFunction`'s editing function should allow arbitrary code to be run on the activations.
2. **Scalable:** `FlexModel` should be agnostic to the number of GPUs or GPU nodes, the model architecture (e.g. LLaMA, Falcon, GPT), the model size, and the distribution strategy (e.g. DP, FSDP, TP, PP) or composition thereof.

3.1 API

3.1.1 `FlexModel`

The `FlexModel` class provides the main interface for user interactions. It inherits from `nn.Module`, allowing developers to easily interact with the wrapped model via the `nn.Module` API without any code changes. This also enables `FlexModel` to intercept API calls and inject additional logic if required. `FlexModel` requires a few initialization arguments: The `nn.Module` to wrap, a dictionary object where the `HookFunctions` send retrieved activations, and the distributed strategy used to launch the wrapped module, defined in terms of TP, PP and DP sizes. For more detail on the use of the distributed strategy information, see Appendix B. See Figure 1 for an example of instantiating a `FlexModel` using HuggingFace Accelerate for model distribution. For more detail, see Appendix A.

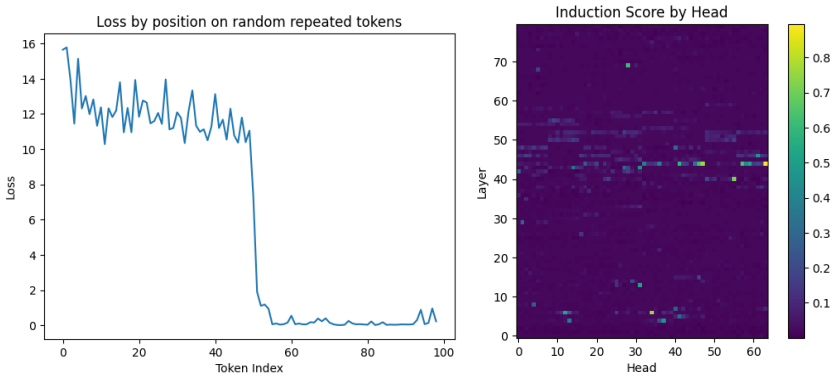
```
model = AutoModelForCausalLM.from_pretrained("model-name")
model = accelerator.prepare(model)

output_dict: Dict[str, Tensor] = {}
model = FlexModel(model, output_dict,
 data_parallel_size=accelerator.num_processes)
```

Figure 1: `FlexModel` initialization example.

3.1.2 `HookFunction`

Once a `FlexModel` has been instantiated, users may define a collection of `HookFunctions` to perform activation retrieval and/or editing. To create a `HookFunction`, the user provides the name of the target module/tensor, the expected shape of the activation tensor, and, optionally, an editing function to run, see Figure 4 in Appendix A. The expected shape is a tuple of integers representing the shape of the target activation tensor. The user need not fill in all of the dimensions; only the sharded dimensions are required. Other dimensions may simply be annotated as `None` and are inferred by the communication system. The editing function is defined by the user and must adhere to a specific signature. However, the contents of the function are completely up to the user. The guarantee within



(a) (b)

Figure 2: Empirical verification of the presence of induction heads within LLaMA-2-70B using a repeated randomly generated sequence as input. The measured loss per token is shown in (a) and the location of the induction heads is shown in (b), indicated by the high induction score values.

the editing function body is single-threaded behaviour and the full activation tensor as an input. There are also additional input options to support other `FlexModel` features like save contexts and trainable modules. See Appendix B, and Figure 5 therein, for details on how `FlexModel` and `HookFunction` implement these features.

4 Experiments

4.1 Induction Head Isolation

Introduced in [10], induction heads are a specific type of attention head within transformer models that emerge during pretraining. Olsson et al. [24] theorize that induction heads are the primary driver for in-context learning in LLMs. An attention head is classified as an induction head if it performs both prefix-matching and copying. Concretely, an induction head performs the operation `[A][B]...[A]`$\rightarrow$`[B]`, where `[A]` and `[B]` are tokens in a sequence. Prefix-matching means that the induction head looks back into the context and attends to the token `[B]` which was prefixed by `[A]`. The induction head then copies this pattern by increasing the output logit for token `[B]`. Induction heads are potentially powerful because they have also been observed performing this algorithm on higher-level concepts [24], rather than just token level behaviour.

To demonstrate the capabilities of `FlexModel`, an induction head search within LLaMA-2-70B, distributed using FSDP over four A100 GPUs, is conducted. Results for LLaMA-2-13B are shown in Figure 8 of Appendix E. The goal is to identify and isolate any induction heads in the model by retrieving the attention maps of each attention head and evaluating their induction head score. We first uniformly sample 50 tokens, with replacement, to produce a sequence which is repeated once to obtain a 100 token query. Because the first half of the sequence is randomly sampled, the model is expected to poorly predict that portion of the sequence. However, in the presence of induction heads, the model should be able to predict the latter half of the sequence, conditioned on the first half, nearly perfectly, as this aligns with induction head behaviour. The per-token-loss for such a sequence structure is visualized in Figure 2a. For a given model head, the induction score is computed by averaging the attention scores $50 - 1 = 49$ tokens back, since induction heads attend to the token *after* the last occurrence of the *current* token, for each sequence position. These scores, across all layers and heads, are shown in Figure 2b as a heat map. We observe candidates in the middle layers.

4.2 TunedLens

TunedLens [3] is an interpretability technique for investigating the residual stream of a transformer model using linear probes [1]. The probes map the transformer layer residual streams to a similar space as the transformer model output residual stream using an affine projection. Thereafter, the final normalization and output unembedding layers are applied, producing token distributions at

Table 1: Forward pass latency comparison on a simulated model, with alternating `ColumnParallelLinear` and `RowParallelLinear` layers from the Megatron-LM codebase, and a model dimension of 4096. The profiles are run on 4 A100-80G GPUs, and `HookFunctions` are applied on each linear layer. "No Hooks" denotes a regular forward pass. The `FlexModel` runs vary in distributed communication and local data movement parameters.

	Extra Comm.	Local Data Transfer	Pinned Mem.	Time Per Step (s)
No Hooks			N/A	0.1237
`FlexModel`	✓		N/A	0.4016
`FlexModel`	✓	✓		6.071
`FlexModel`	✓	✓	✓	2.632

every model layer. This allows researchers to observe how predictions are iteratively refined during a model's forward pass. To demonstrate the features of `FlexModel`, a toy experiment implementing TunedLens applied to LLaMA-2-13B is constructed. See Appendix D for details about the `FlexModel` implementation of TunedLens, as well as an inference example. Using `FlexModel` to implement TunedLens greatly reduces the amount of code and decouples TunedLens from the base model.

4.3 Communication Overhead

It is important to note that `HookFunctions` may introduce substantial overhead into model execution, the sources of which stem from editing function compute, added collective communication and device-to-host data movement. Forward-pass editing functions are fully-exposed on the critical path, so latency costs related to host-device synchronization and added compute cannot be hidden. `HookFunctions` applied to modules with GPU-sharded outputs also incur latency costs, primarily due to added collective communication used to materialize the output activation tensor. Finally, CPU offloading of activation tensors during the execution of many `HookFunctions` experience the heaviest overhead, due to slow device-to-host data transfers of activation tensors.

To investigate the impact of both the added communication and host memory allocation, we use the PyTorch profiler to measure the execution time of several different experiments. A simulated model is used, comprised of alternating Megatron-LM `ColumnParallelLinear` and `RowParallelLinear` layers with ReLU activation functions. In total, the model has 32 linear layers with a model dimension of 4096. The model is distributed along the tensor parallel dimension only. The profiles are run on four A100-80G GPUs, with NVLink-v4 intraconnect. The primary metric measured is the time per profiler step, which is averaged over 10 iterations, as we observed little variance. The time per profile step is calculated as the maximum of the reported `self_cuda_time_total` and `self_cpu_time_total`. Refer to Table 1 for the timing results, along with experiment parameters.

The results of the experiment profiling showed a reasonable increase in latency when extra collective communication operations are introduced. When opting for CPU offloading in favour of GPU offloading, there was a substantial increase in latency. Hence, it is extremely important for `FlexModel` to avoid frequent CPU offloads when necessary. Using CPU pinned memory mitigates a portion of this overhead. Further discussion on key optimization factors is found in Appendix C.

5 Conclusion

In this paper, we introduced a new unified library to advance alignment and interpretability research on LLMs and beyond. Prior works [20, 3] have successfully provided many of the features that we implement. However, existing works have limited or no support for distributed models, and most libraries only consider a restricted set of compatible models. As such, `FlexModel` provides a straightforward user interface and an infrastructure-agnostic wrapper around PyTorch models requiring minimal additional code. Additionally, we have shown how `FlexModel` provides utility for workloads across a selection of current research directions within LLM alignment and interpretability. These include induction head identification, linear probing, activating editing, activation caching, and simple insertion of generic trainable modules. We aim to provide on-going support for `FlexModel` and implement additional features in future work.

Social Impacts Statement

The `FlexModel` package is designed to make interpretability and explainability research simpler and more intuitive for large models that require distributed compute. Large models, especially in the form of LLMs, are becoming increasingly common and are widely deployed in important settings. Facilitating such research is paramount to ensuring that such models are safe, free from bias, and robust in the wider settings in which they are used.

References

[1] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. In *International Conference on Learning Representations (ICLR), Workshop Track Proceedings*, 2017.

[2] Amos Azaria and Tom Mitchell. The Internal State of an LLM Knows When its Lying. *arXiv preprint arXiv:2304.13734*, 2023.

[3] Nora Belrose, Zach Furman, Logan Smith, Danny Halawi, Igor Ostrovsky, Lev McKinney, Stella Biderman, and Jacob Steinhardt. Eliciting Latent Predictions from Transformers with the Tuned Lens. *arXiv preprint arXiv:2303.08112*, 2023.

[4] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2022.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[7] Alan Cooney. CircuitsVis, 2022.

[8] Nelson Elhage, Tristan Hume, Catherine Olsson, Neel Nanda, Tom Henighan, Scott Johnston, Sheer ElShowk, Nicholas Joseph, Nova DasSarma, Ben Mann, Danny Hernandez, Amanda Askell, Kamal Ndousse, Andy Jones, Dawn Drain, Anna Chen, Yuntao Bai, Deep Ganguli, Liane Lovitt, Zac Hatfield-Dodds, Jackson Kernion, Tom Conerly, Shauna Kravec, Stanislav

Fort, Saurav Kadavath, Josh Jacobson, Eli Tran-Johnson, Jared Kaplan, Jack Clark, Tom Brown, Sam McCandlish, Dario Amodei, and Christopher Olah. Softmax Linear Units. *Transformer Circuits Thread*, 2022.

[9] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy Models of Superposition. *Transformer Circuits Thread*, 2022.

[10] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A Mathematical Framework for Transformer Circuits. *Transformer Circuits Thread*, 2021.

[11] Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. What Can Transformers Learn In-Context? A Case Study of Simple Function Classes. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[13] Jaehun Jung, Lianhui Qin, Sean Welleck, Faeze Brahman, Chandra Bhagavatula, Ronan Le Bras, and Yejin Choi. Maieutic Prompting: Logically Consistent Reasoning with Recursive Explanations. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[14] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020.

[15] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[16] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. In *Proceedings of the VLDB Endowment*, 2020.

[17] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Alexander Cosgrove, Christopher D Manning, Christopher Re, Diana Acosta-Navas, Drew Arad Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue WANG, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Andrew Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic Evaluation of Language Models. *Transactions on Machine Learning Research*, 2023.

[18] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations (ICLR)*, 2017.

[19] Neel Nanda. 200 COP in MI: Studying Learned Features in Language Models.

[20] Neel Nanda and Joseph Bloom. TransformerLens, 2022.

[21] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations (ICLR)*, 2023.

[22] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[23] nostalgebraist. Interpreting GPT: The logit lens, 2020.

[24] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context Learning and Induction Heads. *Transformer Circuits Thread*, 2022.

[25] Alicia Parrish, Angelica Chen, Nikita Nangia, Vishakh Padmakumar, Jason Phang, Jana Thompson, Phu Mon Htut, and Samuel Bowman. BBQ: A Hand-Built Bias Benchmark for Question Answering. In *Association for Computational Linguistics (ACL)*, 2022.

[26] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[27] Ludwig Schubert, Michael Petrov, Shan Carter, Nick Cammarata, Gabriel Goh, and Chris Olah. OpenAI Microscope, 2020.

[28] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[29] Jacob-Junqi Tian, David Emerson, Sevil Zanjani Miyandoab, Deval Pandya, Laleh Seyyed-Kalantari, and Faiza Khan Khattak. Soft-prompt Tuning for Large Language Models to Evaluate Bias. *arXiv preprint arXiv: 2306.04735*, 2023.

[30] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*, 2023.

[31] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.

[32] Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, and Tengyu Ma. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*, 2023.

[33] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment*, 2023.

Appendix

A `FlexModel` Usage

Below, in Figures 3 and 4, are code examples demonstrating how to instantiate a `FlexModel` object and how to define a custom editing function in the proposed framework, respectively.

```
from flex_model import FlexModel, HookFunction

accelerator = Accelerator()
model =
 AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-13b-chat-hf")
tokenizer =
 AutoTokenizer.from_pretrained("meta-llama/Llama-2-13b-chat-hf")

model = accelerator.prepare(model)

activation_dict: Dist[str, Tensor] = {}
model = FlexModel(model, activation_dict,
 data_parallel_size=accelerator.num_processes)
```

Figure 3: `FlexModel` initialization example.

```
def my_edit_fn(current_module, inputs, save_ctx, global_modules) ->
 Tensor:
    # Cache data for later.
    _, s, _ = torch.svd(inputs)
    save_ctx.activation_singular_values = s
    # Edit the activation tensor.
    inputs = torch.where(inputs > 1.0, inputs, 0.0)
    # Apply a torch layer to the activation tensor.
    outputs = global_modules["linear_projection"](inputs)
    return outputs

my_hook_function = HookFunction(
    "model.layers.23",
    expected_shape=(4, 1024, 4096),
    editing_function=my_edit_fn)
flex_model.register_hook_function(my_hook_function)
```

Figure 4: `HookFunction` registration example.

B `FlexModel` Communication

The core design of the `HookFunction` editing function runtime is single-threaded execution and access to the target unsharded activation tensor for manipulation or storage. Single-threaded execution means user code is run once across all workers. Hence, they need not worry about multi-worker coordination. This is much easier for the user to develop and iterate on, as code running on their local python interpreter is directly transferable to the editing function. The user-defined code is run on the desired unsharded activation tensor. For example, consider a model that is duplicated across 4 GPUs (i.e. DP = 4). The communication system gathers the activations from each duplicated model instance and concatenates them, in the batch dimension, to form the relevant unsharded activation tensor. Additionally, the activation tensors, which are streamed to the `FlexModel`'s output dictionary, are only present on the rank 0 worker's CPU to prevent additional GPU memory usage.

To enable this functionality, the `FlexModel` and `HookFunction` instances coordinate the distributed communication. `FlexModel` initialize the communication system and provides a pointer to the output dictionary to each registered `HookFunction` instance. The `HookFunctions` handle the necessary distributed communication for unsharding of the activation tensor, running the editing function and dispersing the subsequent edited activation tensor.

Initialization of the communication system consists of constructing a 3D GPU device mesh, exactly representing the 3D layout of the wrapped model (i.e. DP, TP and PP; see Figure 5a). PyTorch distributed groups are created such that activation tensors can be gathered, scattered, and singleton accumulated on CPU. Note that gathering and scattering activation tensors only require a subset of the distributed groups and communication collectives used for accumulating activation tensors to the rank 0 CPU.

To run the single-threaded editing function on the unsharded activation tensor, each `HookFunction` runs prologue and epilogue functions to support the editing function. The prologue consists of extracting the local activation tensor from the layer outputs of the current module and gathering them along the necessary dimensions to form the full activation tensor. The epilogue is an exact inverse of the prologue wherein it scatters the edited activation tensor along the same dimensions in reverse-order and repacks the edited local activation tensor into the layer outputs. Gathering and scattering are performed along the DP and TP axes, but not the PP axis, as activations are never sharded between different PP stages. Refer to Figure 5b for the full workflow. Since the layer outputs can be arbitrary Python objects, `HookFunction` uses a tree-traversal strategy to unpack and repack the activation tensors with the layer outputs. These are similar to JAX `tree_util` functions [5]. Finally, activation tensors present on each pipeline stage are gathered to the global rank 0 GPU, and placed onto its CPU.

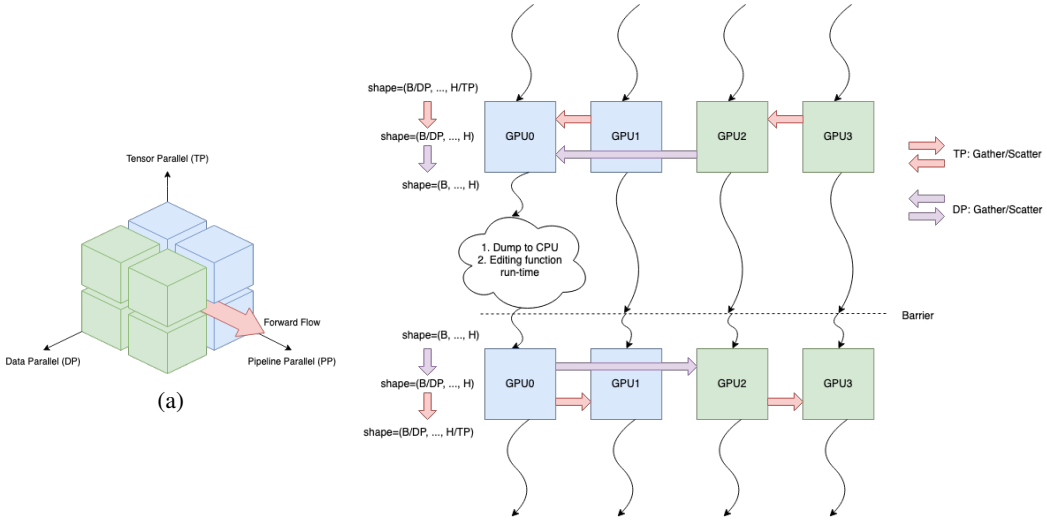


Figure 5: (a) Models distributed over 8 GPUs, depicted as a 3D mesh, have activation tensors distributed along a 2D slice in the TP and DP axes. The process facilitating single-threaded editing is shown in (b). Gather collectives are used to unshard the TP then DP axes. The local rank 0 GPU puts a copy of the full activation tensor in CPU and runs the user-provided editing function on it. Finally scatter collectives are used to redistribute shards of the modified tensor first across DP then TP dimensions.

C Communication Overhead

The communication overhead results in Table 1 show a significant increase in latency as collective communication and CPU offloading were added. The baseline "No Hooks" case involved running a standard forward pass through the simulated model. The rest of the cases use `FlexModel` and complexities are incrementally added, such as extra collective communication to materialize the

full activation tensors, local data transfers from device to host and the usage of pinned memory to stage the device-to-host transfers. Since half of the layers in the test model required collective communication (i.e. the `ColumnParallelLinear` layers), there are 16 additional `all-gather` collective communication operations performed. This results in ~3x additional latency. Note that the GPU offload itself does not result in any significant added overhead since the activation tensor is already resident in device memory. The bulk of the overhead stems from the CPU offloading operation, which occurs at every layer in the model. Offloading to pinned memory before pageable memory is much faster than naively offloading immediately to pageable memory, at 6.5x increased latency compared to 15x increased latency relative to GPU offloading.

After profiling many `HookFunction` scenarios, it is immediately obvious that mitigating overhead from copy operations is of the utmost importance. Currently, activation tensors are guaranteed to be offloaded to the global rank 0 GPU DRAM (or local CPU RAM). However, for frameworks such as TunedLens, this results in an indirection where `FlexModel` offloads tensors to rank 0, but the tensors must be moved again to where the TunedLens model lives. This is further exacerbated by the NCCL requirement for tensors to be resident in device memory, which incurs a host-to-device transfer if activation tensors were offloaded to host memory. Therefore an obvious optimization would be to achieve zero-copy activation tensor routing, where the user may specify the required offload location of tensors or groups of tensors. Such a strategy avoids unnecessary memory allocations and/or copying operations.

D TunedLens

TunedLens [3] is an extention of LogitLens [23], which projects the transformer layer residual streams into distributions over the model vocabulary. LogitLens applied to a hidden state, h, at layer l is defined as

$$\text{LogitLens}_l(h_l) = \text{LayerNorm}(h_l)W_U,$$

where LayerNorm is the output normalization layer of the transformer, and W_U is the transformer unembedding matrix. Using the final layer unembedding matrix, LogitLens generates a token prediction given the residual stream state at the current layer. Applied to each transformer layer, the evolution of predictions during the forward pass can be observed.

However, it has been shown that the LogitLens is a biased estimator of the final logit distribution, and that it is also susceptible to residual stream covariance drift [3]. Hence, the predictions it computes are not trustworthy approximations of the contents of the residual stream, and it cannot perform well across different transformer architectures. TunedLens improves on LogitLens by decreasing the effect of representation drift and bias using a learnable affine projection. This allows TunedLens to map the hidden state of a given layer to a similar space as the hidden state of the final transformer layer. Formally, the TunedLens at layer l is computed as:

$$\text{TunedLens}_l(h_l) = \text{LogitLens}_l(A_l h_l + b_l)$$

where A_l and b_l are the affine projection weights and biases for layer l. Each TunedLens probe is trained to minimize the KL divergence between current layer token distribution and the transformer output logits.

As a toy demonstration of TunedLens using `FlexModel`, a simplified version of the training procedure is run on LLaMA-2-13B. The model is wrapped with `FlexModel`, and `HookFunctions` are applied to each transformer layer residual stream. Additionally, `FlexModel` provides functions for retrieving unsharded parameters in a similar fashion to activation retrieval using `HookFunctions`. This is used to collect the unsharded RMSNorm and output embedding weights to be applied to each TunedLens probe. Using the `FlexModel` API, all of the interactions required between the TunedLens model and the wrapped LLaMA-2-13B model are condensed to a small code-block, see Figure 6. The experiment is run using the WikiText-103 dataset [18] and a short maximum sequence length of 128 tokens to reduce the resources and time required for the experiment. See Figure 7 for an example inference. Even in this toy example, we observe the predictions made by each layer being steadily refined during the forward pass.

```
class TunedLens(nn.Module):
    ...
    # Setup Flex Model wrapper.
    self.act_dict = {}
    self.flex_model = FlexModel(
        base_model,
        self.act_dict,
        tp_size, pp_size, dp_size,
    )
    # Hook into residual stream states.
    for layer_name in residual_stream_layers:
        self.flex_model.register_hook_function(
            HookFunction(
                layer_name,
                expected_shape=(None, None, hidden_dim),
            )
        )
    # Retrieve unsharded weights.
    unembed_weight = self.flex_model.get_module_parameter(
        "output.weight",
        (vocab_size, hidden_dim),
    )
    norm_weight = self.flex_model.get_module_parameter(
        "norm.weight",
        (hidden_dim,),
    )
    # Init Norm and Unembed TunedLens layers using weights.
    ...
```

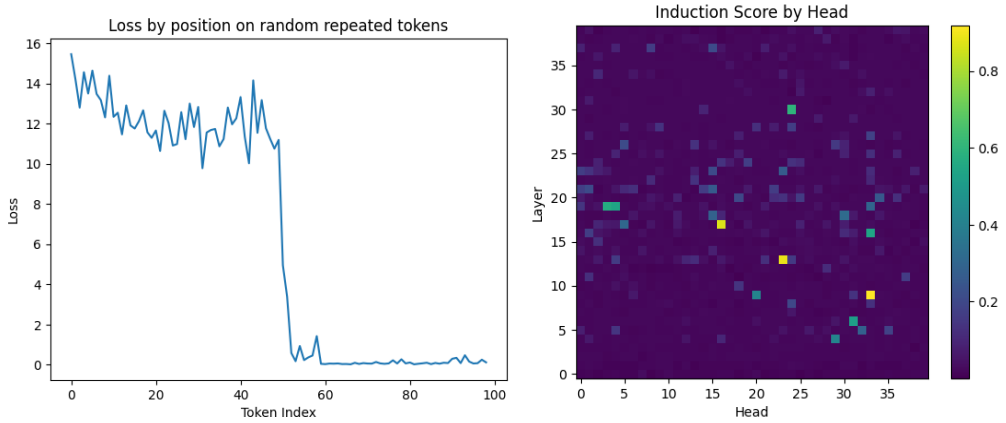
Figure 6: TunedLens initialization example using `FlexModel` and `HookFunctions`. `FlexModel` is used to wrap the base model, and `HookFunctions` are placed at each layer to retrieve the unsharded residual stream activations. The weights for the normalization and unembedding layers are also fetched and unsharded by `FlexModel` to generate the residual stream logit distributions.

E Additional Induction Head Search Results

In this section, the results of an induction head search on LLaMA-2-13B are reported. As done for LLaMA-2-70B, the model is distributed using FSDP over for A100 GPUs. Several induction heads are identified in the early layers of the model.

POS	118	119	120	121	122	123	124	125	126
TGT	692	475	4909	1919	322	15241	1283	575	3145
	oug	ain	ville	_,	_and	_launched	_off	ens	ives
L0	1038	284	303	1919	322	278	278	6270	322
	urr	al	st	_,	_and	_the	ensive	_and	
L1	1038	284	29892	1919	322	278	278	6270	304
	urr	al	,	_,	_and	_the	ensive	_to	
L2	1038	347	29892	29892	322	278	278	6270	297
	urr	ie	,	,	_and	_the	ensive	_in	
L3	1038	290	29892	29892	607	278	278	278	297
	urr	om	,	,	_which	_the	_the	_the	_in
L4	1038	347	423	29892	607	278	278	6270	2750
	urr	ie	ia	,	_which	_the	_the	ensive	_against
L5	1038	279	423	1919	322	372	278	6270	2750
	urr	ar	ia	_,	_and	_it	_the	ensive	_against
L6	1038	475	423	1919	322	278	278	6270	2750
	urr	ain	ia	_,	_and	_the	_the	ensive	_against
L7	29889	475	423	869	322	278	278	6270	2750
	.	ain	ia	_.	_and	_the	_the	ensive	_against
L8	29889	2497	423	1919	322	278	278	6270	2750
	.	ula	ia	_,	_and	_the	_the	ensive	_against
L9	1038	2497	1049	1919	322	278	278	6270	573
	urr	ula	land	_,	_and	_the	_the	ensive	ive
L10	1038	2497	1049	869	322	278	263	6270	573
	urr	ula	land	_.	_and	_the	_a	ensive	ive
L11	29889	423	423	869	322	278	263	6270	2750
	.	ia	ia	_.	_and	_the	_a	ensive	_against
L12	1038	2606	423	869	322	278	263	6270	2750
	urr	ali	ia	_.	_and	_the	_a	ensive	_against
L13	29889	29874	423	869	322	278	263	6270	2750
	.	a	ia	_.	_and	_the	_a	ensive	_against
L14	7935	29874	4909	869	322	278	263	6270	21881
	_Island	a	ville	_.	_and	_the	_a	ensive	_tropical
L15	29889	284	423	869	322	278	263	6270	583
	.	al	ia	_.	_and	_the	_a	ensive	ies
L16	29889	29874	423	1919	1550	896	263	6270	2750
	.	a	ia	_,	_while	_they	_a	ensive	_against
L17	29889	284	4909	1919	322	297	263	6270	2750
	.	al	ville	_,	_and	_in	_a	ensive	_against
L18	1049	284	4909	869	1550	892	278	6270	2750
	land	al	ville	_.	_while	_were	_the	ensive	_against
L19	1049	261	4909	869	322	278	263	6270	2750
	land	er	ville	_.	_and	_the	_a	ensive	_against
L20	17839	261	4909	869	322	278	263	6270	2750
	_Islands	er	ville	_.	_and	_the	_a	ensive	_against
L21	8579	261	4909	869	1550	297	263	6270	2750
	_Pap	er	ville	_.	_while	_in	_a	ensive	_against
L22	2620	261	4909	869	1550	297	263	6270	2750
	atter	er	ville	_.	_while	_in	_a	ensive	_against
L23	2620	261	4909	869	1550	297	263	6270	2750
	atter	er	ville	_.	_while	_in	_a	ensive	_against
L24	650	261	4909	869	1550	278	11531	6270	2750
	one	er	ville	_.	_while	_the	_campaign	ensive	_against
L25	8579	475	4909	869	1550	297	11531	6270	2750
	_Pap	ain	ville	_.	_while	_in	_campaign	ensive	_against
L26	8579	475	4909	869	1550	9870	263	6270	3145
	_Pap	ain	ville	_.	_while	_Australian	_a	ensive	ives
L27	8579	475	4909	869	1550	9870	278	6270	3145
	_Pap	ain	ville	_.	_while	_Australian	_the	ensive	ives
L28	8579	475	4909	869	1550	9870	263	6270	3145
	_Pap	ain	ville	_.	_while	_Australian	_a	ensive	ives
L29	8579	475	4909	869	322	9870	278	6270	3145
	_Pap	ain	ville	_.	_and	_Australian	_the	ensive	ives
L30	8579	475	4909	869	322	9870	278	6270	3145
	_Pap	ain	ville	_.	_and	_Australian	_the	ensive	ives
L31	8579	475	4909	869	1550	9870	278	6270	3145
	_Pap	ain	ville	_.	_while	_Australian	_the	ensive	ives
L32	692	475	4909	869	322	9870	263	6270	3145
	oug	ain	ville	_.	_and	_Australian	_a	ensive	ives
L33	692	475	4909	869	1550	297	263	6270	3145
	oug	ain	ville	_.	_while	_in	_a	ensive	ives
L34	692	475	4909	869	1550	297	385	6270	3145
	oug	ain	ville	_.	_while	_in	_an	ensive	ives
L35	692	475	4909	869	1550	297	385	6270	3145
	oug	ain	ville	_.	_while	_in	_an	ensive	ives
L36	692	475	4909	869	1550	297	263	575	3145
	oug	ain	ville	_.	_while	_in	_a	ens	ives
L37	692	475	4909	869	322	297	263	575	3145
	oug	ain	ville	_.	_and	_in	_a	ens	ives
L38	692	475	4909	869	322	297	263	575	3145
	oug	ain	ville	_.	_and	_in	_a	ens	ives
L39	692	475	4909	869	322	297	263	575	3145
	oug	ain	ville	_.	_and	_in	_a	ens	ives

Figure 7: **Toy TunedLens +** `FlexModel` example inference. The `POS` label specifies the token position in the context, the `TGT` label specifies the model output argmax token, and each row corresponds to the TunedLens probe argmax token at layer `L`.



(a) (b)

Figure 8: Empirical verification of the presence of induction heads within LLaMA-2-13B using a repeated randomly generated sequence as input. The measured loss per token is shown in (a) and the location of the induction heads is shown in (b), indicated by the high induction score values.