Sylva: Sparse Embedded Adapters via Hierarchical Approximate Second-Order Information

Baorun Mu
University of Toronto, Vector Institute, CentML
Toronto, Canada
baorun.mu@mail.utoronto.ca

Christina Giannoula
University of Toronto, CentML
Toronto, Canada
christina.giann@gmail.com

Shang Wang
University of Toronto, Vector Institute, CentML
Toronto, Canada
wangsh46@cs.toronto.edu

Gennady Pekhimenko
University of Toronto, Vector Institute, CentML
Toronto, Canada
pekhimenko@cs.toronto.edu

ABSTRACT

Fine-tuning is the gateway to transferring learned knowledge in a pre-trained Large Language Model (LLM) on *many* downstream applications. To make LLM fine-tuning more affordable, prior works follow two paths: i) *adapters* freeze the pre-trained LLM weights and inject a small number of trainable weights during fine-tuning, and ii) *pruners* remove the less important weights in pre-trained LLMs and train the remaining sparse weights during fine-tuning. We find that the former introduces computation overheads due to the injected trainable parameters, while the latter introduces an expensive pre-processing step to identify the important weights and degrades model quality. To get the best of both worlds, we propose **Sylva**, a novel LLM fine-tuning procedure that provides high system performance during fine-tuning and attains state-of-the-art model quality on downstream applications. Sylva identifies the most important LLM weights via second-order information in a pre-processing step, and significantly reduces the computation and storage costs of the pre-processing step via i) a hierarchical approximation of second-order information, and ii) an online projection and rediagonalization algorithm. Sylva trains only the sparse important weights and embeds these sparse weights into the pre-trained LLM during fine-tuning to provide high system performance. We show that end-to-end fine-tuning with Sylva is, on average, 5.1× faster than ZeRO and 1.2× faster than LoRA, the state-of-the-art adapter approach. Sylva's hierarchical approximation reduces the peak GPU memory in the pre-processing step by 2.3× compared to K-FAC, the most widely used approximation to second-order information. The source code of Sylva is publicly available at https://github.com/CentML/Sylva .

CCS CONCEPTS

• **Computing methodologies → Machine learning**; *Parallel computing methodologies*; • **Computer systems organization →** *Single instruction, multiple data.*

KEYWORDS

Fine-tuning, Large Language Models, GPUs

ACM Reference Format:
Baorun Mu, Christina Giannoula, Shang Wang, and Gennady Pekhimenko. 2024. Sylva: Sparse Embedded Adapters via Hierarchical Approximate Second-Order Information. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 4–7, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656619

1 INTRODUCTION

Large language models (LLMs) are deployed on various applications, e.g., machine translation [4, 54], code generation [6, 27], and text-to-image generation [2, 45]. Training an LLM from scratch for *each* application is prohibitively expensive since LLMs typically have billions of parameters [4, 53]. LLM training requires multiple accelerators with large memory capacities (e.g., NVIDIA A100 80 GB) and ultra-high bandwidth interconnects (e.g., InfiniBand EDR 100Gb/s [35]). A more cost-efficient approach is *fine-tuning* for each downstream task after obtaining a *pre-trained* LLM, i.e. a general-purpose LLM trained on large corpora. With fine-tuning, the learned knowledge of the pre-trained LLM is transferred to downstream tasks by continuing to train the LLM on the task-specific datasets [41, 58]. The naive full fine-tuning approach (Figure 1a) trains *all* weights of the pre-trained LLM and thus requires the *same* hardware capacities as pre-training, which are typically in high demand and expensive or difficult to access [16, 48]. Memory optimizations such as Zero Redundancy Optimizer (ZeRO) [42, 43] are proposed to reduce GPU memory usage for full fine-tuning.

To make LLM fine-tuning more affordable, two directions have been explored: adapters [8, 10, 19, 20, 31, 46, 59] and pruners [12, 13, 18, 23, 24, 26, 28, 49]. On the one hand, adapters (Figure 1b-c) keep the pre-trained LLM parameters as *frozen weights* and inject a small amount of *trainable weights* during fine-tuning. Low-Rank Adaptation (LoRA) [20], the state-of-the-art adapter approach, inserts trainable weights as low-rank decompositions during fine-tuning. Adapters attain state-of-the-art model quality by combining the frozen pre-trained weights and injected trainable weights. However, they introduce computation overheads during fine-tuning due to the injected trainable weights (See Section 2). For example, LoRA requires forward-backward passes on *both* the frozen weights and the injected trainable weights. On the other hand, pruners (Figure 1d) introduce a pre-processing step that identifies and removes the

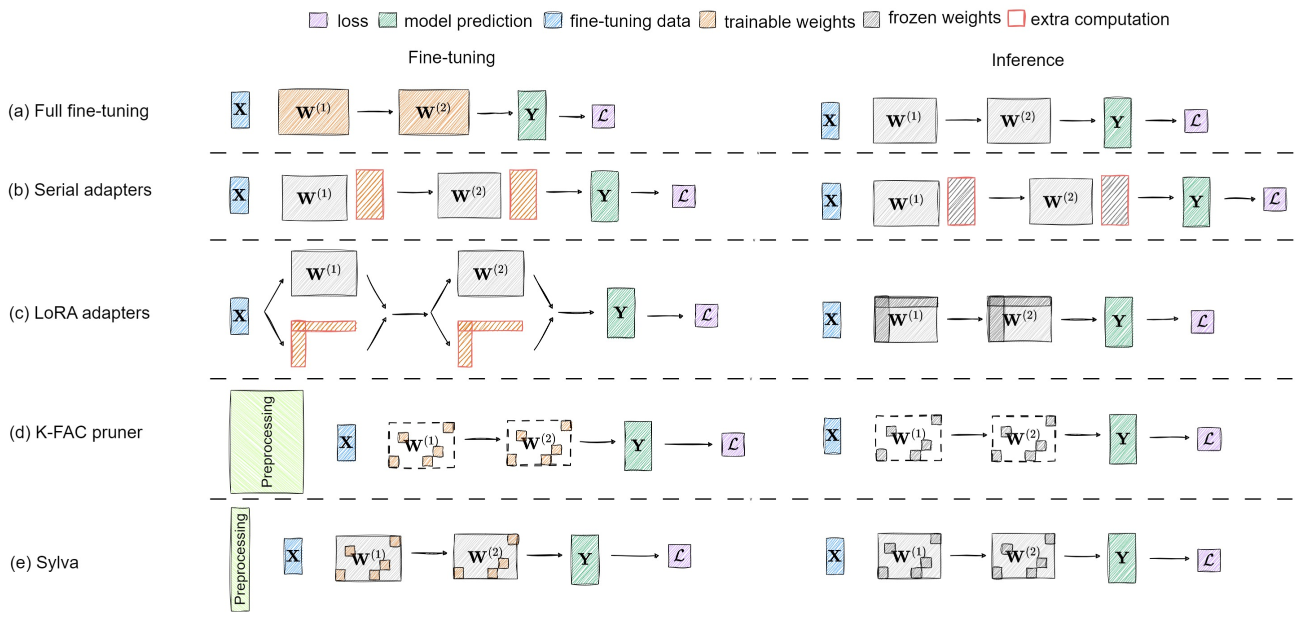

Figure 1: Fine-tuning and inference execution flow of various approaches.

less important weights of the pre-trained LLM. A few pioneering works [18, 26] propose to identify weight importance leveraging second-order information, specifically the Hessian [15, 29, 50]. However, the computation and storage cost of the Hessian is prohibitive for modern deep neural networks. Kronecker-Factorized Approximate Curvature (K-FAC) [17, 34] improves on them by using a block-diagonal approximation to the Fisher Information Matrix (FIM) and further approximates each block by the Kronecker products of two much smaller matrices. Indicatively, K-FAC pruner [56] is practical for convolutional neural networks, but it still requires excessive memory in the pre-processing step for multi-billion parameter LLMs (e.g., LLaMA 3B model incurs out-of-memory errors during pre-processing on 24GB GPUs). Moreover, although pruners provide high system performance in the fine-tuning process by leveraging weight sparsity, they typically degrade the model quality since they remove a large subset of the pre-trained LLM weights and incur high computation/storage costs in the pre-processing.

We propose a novel method named **Sylva**[1] (Figure 1e) which combines the strengths of adapters and pruners while mitigating their weaknesses. Sylva optimizes two goals: system performance during fine-tuning and model quality on downstream tasks. For the former goal, Sylva identifies a small number of important weights in the pre-trained LLMs leveraging second-order information and only trains these sparse weights during fine-tuning (similar to pruners). For the latter goal, Sylva keeps the pre-trained LLM weights frozen and employs these frozen weights in prediction (similar to adapters). Sylva addresses the following two shortcomings of prior works. First, it significantly reduces the computation and storage costs of the pre-processing step in pruners via (i) a hierarchical approximation of the second-order information and (ii) an online projection and rediagonalization algorithm (See Section 3). Second, it eliminates the computation overheads in adapters via embedding the sparse trainable weights into the frozen pre-trained LLM weights throughout the fine-tuning procedure.

We show that on average, the end-to-end fine-tuning with Sylva is 5.1× faster than full fine-tuning with ZeRO optimizer and 1.2× faster than LoRA, the state-of-the-art adapter approach. We study the sensitivity of Sylva's performance on various layer dimensions and sequence lengths. Sylva's forward and backward pass time is up to 2.3× faster on fully connected layers. Additionally, Sylva reduces GPU memory usage by up to 35% compared to LoRA during fine-tuning. In the pre-processing step, Sylva's hierarchical approximation reduces the peak GPU memory by 2.3× compared to K-FAC, the most widely-used approximation to second-order information.

To conclude, we make the following key contributions:

- We comprehensively analyze prior LLM fine-tuning approaches, and propose Sylva, an effective fine-tuning method for LLMs.
- We design a hierarchical approximation of the second-order information along with an online projection and rediagonalization algorithm to significantly reduce the computation and storage costs, when identifying the most important weights of a pre-trained LLM. We embed the most important weights as trainable parameters into the pre-trained LLM to eliminate the extra computation costs that would have been introduced by adapters (e.g. LoRA) injecting trainable weights.
- We extensively evaluate Sylva by fine-tuning state-of-the-art LLMs on multi-node systems and showing that Sylva outperforms prior works in performance and memory efficiency. Sylva also provides high model quality and low pre-processing costs.

[1]We open-source Sylva at https://github.com/CentML/Sylva .

2 BACKGROUND AND MOTIVATION

2.1 Overview of Prior Works

Figure 1 summarizes various approaches that accelerate fine-tuning.

The naive fine-tuning approach, known as **full fine-tuning** (Figure 1a), keeps *all* weights of the pre-trained large language model (LLM) and continues training them on task-specific datasets. Zero Redundancy Optimizer (ZeRO) [42, 43] improves the per-device memory footprint of full fine-tuning via sharding and offloading. However, ZeRO introduces communication overheads because they require gathering sharded weights at every iteration. Another line of prior works [21, 37] improves fine-tuning performance using pipeline or tensor parallelism. Existing implementations of these works are *only* tailored for a few LLM families and thus lack generality. For example, Megatron-LM [37] optimizes the GPT family [4] of models, and supporting the LLaMA family [53] requires significant engineering efforts and tuning to provide high system performance.

A group of prior works [19, 20, 31, 46], named as **adapters**, keeps all parameters of the pre-trained LLM as *frozen weights* and injects a small number of new parameters as *trainable weights*. Only the latter are trained during fine-tuning, and thus performance is improved. Houlsby et al., [19] propose the serial adapter (Figure 1b) that injects trainable weights as new layers into the pre-trained LLM. Serial adapter achieves state-of-the-art model quality; however, adding additional layers in LLM worsens inference performance compared to full fine-tuning. Some works [31, 46] improve inference performance via reducing the trainable weights in the serial adapter, however, at the cost of degrading model quality. Low-Rank Adaptation (LoRA) [20] (Figure 1c) injects trainable weights as low-rank decompositions to each existing fully connected (FC) layer of the pre-trained LLM. After fine-tuning, the injected trainable weights are merged with the frozen weights via summation. This way, LoRA does not introduce additional inference latency while attaining high model quality.

Another group of prior works [12, 13, 18, 23, 26, 49], named as **pruners**, identifies and removes the less important weights in pre-trained LLM in a pre-processing step (Figure 1d). In fine-tuning, only the non-zero elements of the induced sparse weight matrices are trained and updated using optimized sparse linear algebra libraries. LeCun et al. [26] and Hassibi et al. [18] propose to determine the importance of the pre-trained weights via second-order information, specifically the Hessian [15, 29]. Singh et al. [49] and Kurtic et al. [23] improve on computation and storage costs by using a block-diagonal approximation of the Fisher Information Matrix (FIM) [1, 22, 33] in place of the Hessian. Kurtic et al. [23] enables to scale up to medium-sized language models (e.g. BERT [9]). However, it faces a trade-off between computation/storage costs and the accuracy of the approximation. Specifically, the GPU memory capacity limits the block size to be small. However, the smaller the block size, the more off-diagonal information is disregarded. Frantar et al. [12, 13] enable to prune GPT-family models [4] using second-order derivatives; however, at the price of using a layer-wise reconstruction loss as the pruning objective instead of the model's prediction loss, and thus losing theoretical groundings of prior works [18, 26]. Kronecker-Factorized Approximate Curvature (K-FAC) [17, 34] is the state-of-the-art approximation to second-order information since it significantly reduces computation and storage costs compared to using the Hessian [18, 26]. As shown in Figure 2a (i), K-FAC employs a layer-wise approximation to the FIM and further approximates each block (corresponding to a layer in the neural network) using the Kronecker product of two much smaller matrices. Although K-FAC is practical for medium-sized LLMs (e.g., BERT), the memory footprints of the Kronecker factors are excessive for multi-billion parameter LLMs, making it prohibitive to prune large LLMs on most data-center GPUs.

2.2 Comparison of Prior Works

Table 1 qualitatively compares the aforementioned methods.

Memory Footprint. In ZeRO full fine-tuning, although the gradients and optimizer states are sharded and offloaded, each GPU still needs to temporarily store the gradients for all LLM weights in the backward pass. Serial and LoRA adapters only train the injected weights, thus reducing the peak memory footprints compared to naive full fine-tuning. K-FAC pruner sparsifies the LLM weights. Thus, the memory footprints of weights, gradients and optimizer states are all reduced compared to ZeRO and adapter approaches. However, K-FAC pruner introduces excessive memory footprints during the pre-processing step for the second-order information, which might exceed the GPU memory capacity. For example, K-FAC's pre-processing step requires at least 40GB memory for the LLaMA-3B, thus causing out-of-memory errors on 24GB GPUs. The computations during the K-FAC's pre-processing step to approximate second-order information are also expensive.

Fine-Tuning Time. On multi-GPU systems, the fine-tuning time aggregates computation and communication costs. ZeRO requires computing the gradients of all LLM weights and has large communication overheads due to gathering the sharded weights at each iteration. Serial adapter and LoRA eliminate the need to compute gradients for the frozen weights; however, they introduce extra computation costs because they attach the trainable weights as either extra layers (serial adapter) or extra computation paths at each FC layer of the LLM (LoRA). In LoRA, the trainable weights and pre-trained weights are processed *separately* in the forward and backward passes. Serial and LoRA adapters reduce the communication time compared to full fine-tuning, since they All-Reduce only the gradients of the small amount of injected trainable weights. When setting K-FAC to have the same number of trainable parameters with adapters (regardless of model quality degradation led by the pruned weights), K-FAC pruner provides much higher computation efficiency than adapters since it does not introduce extra computation paths and has a comparable communication cost.

Inference Time. Serial adapter introduces extra inference latency compared to full fine-tuning due to extra LLM layers added during fine-tuning, while LoRA merges the trainable weights into the frozen weights after fine-tuning. Thus, the inference time of LLM fine-tuned using LoRA is the same as that using full fine-tuning. Pruners remove weights from the pre-trained LLM and use sparse weight matrices, thus reducing the inference time compared to ZeRO and adapters.

Model Quality. ZeRO's memory optimizations do not affect the quality of LLMs, thus, ZeRO provides high model quality, since it trains all LLM weights with the task-specific dataset. Serial adapter and LoRA achieve comparable model quality with full fine-tuning, since they combine the learned knowledge of the pre-trained frozen

Table 1: Qualitative comparison of prior works.

Method	Memory Footprint		Fine-tuning Time		Inference Time	Model Quality	Checkpoint Saving
	Pre-processing	Fine-tuning	Computation	Communication			
ZeRO [42, 43] full fine-tuning	-	👎	👎	👎	👍	👍	✗
Serial Adapter [19]	-	👍	👎	👍	👎	👍	✓
LoRA Adapter [20]	-	👍	👎	👍	👍	👍	✓
K-FAC Pruner [56]	👎	👍👍	👍👍	👍	👍👍	👎	✓
Sylva	👍	👍	👍	👍	👍	👍	✓

weights with the knowledge of the additional weights trained with the task-specific dataset. Instead, K-FAC pruner trains a *small* subset of pre-trained LLM weights with the task-specific dataset; thus, it typically degrades model quality.

Checkpoint Saving. ZeRO does not allow sharing the checkpoints of fine-tuned LLMs since all weights are specialized to a particular downstream task. Instead, adapters save storage resources for LLM checkpoints by enabling sharing the frozen weights across multiple downstream tasks, while the injected *task-specialized* trainable weights are stored separately. K-FAC pruner also reduces the LLM checkpoints storage costs: this benefit is obtained via weight pruning, i.e., creating *sparsified* fine-tuned LLMs, instead of sharing the weight values across downstream tasks as adapters do.

Our analysis shows that adapters achieve high model quality by exploiting all the pre-trained LLM weights during fine-tuning, but they introduce additional computation costs for to the injected trainable weights. Instead, pruners leverage sparse trainable weights to reduce computation costs during fine-tuning significantly. However, they downgrade model quality and introduce an expensive pre-processing step. To get the best of both approaches, we introduce Sylva as a combination of adapters and pruners, as described in the next section. Table 1 shows that Sylva provides the most effective solution in pre-processing memory and performance in both fine-tuning and inference, and achieves state-of-the-art model quality. Sylva also reduces the storage costs for checkpoints by storing only the sparse trainable weights for each downstream task.

3 SYLVA: OVERVIEW

Sylva interpolates between the adapter and pruner approach to overcome their shortcomings and acquire their strengths. Figure 1e presents an overview of Sylva. 1) Sylva adopts the pruner's key idea of identifying a small number of important weights of the pre-trained LLM (named sparse trainable weights) via a pre-processing step, and only training these sparse weights during fine-tuning to achieve low computation costs. 2) Sylva adopts the adapter's key idea of freezing all the pre-trained LLM weights in the fine-tuning step and leveraging them in prediction to provide high model quality. In this design, we need to address two key challenges: i) how to minimize the computation and storage costs required by the second-order information in the pre-processing step (**Challenge 1**), and ii) how to eliminate the computation overheads introduced by injecting trainable weights into the pre-trained LLM during fine-tuning (**Challenge 2**). To address these challenges, we propose three key techniques.

1) Hierarchical Approximate Kronecker Factors. During pre-processing, the weight importance is determined by minimizing the impact of pruning on the model's prediction loss, formally, a quadratic model of the loss landscape [18, 26]. Solving this minimization problem by Lagrange multipliers yields the weight importance that involves the Hessian $\mathbf{H}$. In K-FAC pruner, the Hessian $\mathbf{H}$ is approximated by the Kronecker products $\mathbf{A} \otimes \mathbf{G}$, where $\mathbf{A}$ and $\mathbf{G}$ are the covariances of input and output gradients. Henceforth, $\mathbf{A}$ and $\mathbf{G}$ are called the Kronecker factors. The (i, j)-th element in the Kronecker factor $\mathbf{A}$ stands for the correlation (i.e., the degree to which two variables are related) between the i-th and j-th neuron in the LLM layer's inputs. A similar representation holds for the Kronecker factor $\mathbf{G}$. To reduce the memory overheads of the pre-processing step of multi-billion parameter LLMs (**Challenge 1**), we propose a hierarchical approximation of the Kronecker factors. We observe that the magnitude of entries in the Kronecker factors tends to decrease as their distances to the diagonal increase. This is because the neurons closer to each other in the LLMs have a higher correlation. If the i-th and j-th dimensions are close in the LLM layer's inputs, then the (i, j)-th entry in the Kronecker factor $\mathbf{A}$ is close to the diagonal. Based on this observation, we recursively halve the Kronecker factors and approximate the off-diagonal blocks using Singular Value Decomposition (SVD). Let r be the rank in the low-rank decomposition provided by SVD and b be the size of a block in the partition. As shown in Figure 2a (ii), each off-diagonal block $\mathbf{B} \in \mathbb{R}^{b \times b}$ (in purple for $\mathbf{A}$ and in green for $\mathbf{G}$) is approximated using SVD: $\mathbf{B} \approx \mathbf{U}\mathbf{\Lambda}\mathbf{V}^\top$, where each of $\mathbf{U} \in \mathbb{R}^{b \times r}$ and $\mathbf{V} \in \mathbb{R}^{b \times r}$ contains a set of r orthonormal bases that span the approximating subspace, $\mathbf{\Lambda}$ contains a vector of r singular values that stretches the bases. As a result, at each partition, we only store (a) two tall-and-skinny matrices ($\mathbf{U}$ and $\mathbf{V}$ matrices in blue) and a vector ($\mathbf{\Lambda}$ in blue) for each off-diagonal block (instead of the entire block $\mathbf{B}$), and (b) the exact diagonal blocks of the finest partition (shown in red in Figure 2a (ii)). Denote the dimension of $\mathbf{A}$ and $\mathbf{G}$ by m and n, respectively. Let K be the total number of recursive partitions in the hierarchical approximation. The original Kronecker factor $\mathbf{A}$ takes $O(m^2)$ storage, while that of Sylva's hierarchical approximation of $\mathbf{A}$ is much smaller: the approximated off-diagonal blocks take $O(mr(2^k - 1))$ storage, and the exact diagonal blocks of the finest partition take $O(m^2/2^k)$. The storage complexity of $\mathbf{G}$ is analog to that of $\mathbf{A}$ except that it is on the output dimension n.

2) Online Projection and Rediagonalization. We divide the dataset into many mini-batches of data and then process the mini-batches one by one because fitting the entire dataset onto the GPU is prohibitive, in the pre-processing step. Thus, we need to aggregate second-order information from a mini-batch of data with the previous ones, i.e. update an existing hierarchical approximation using incoming data. To further reduce the pre-processing step's

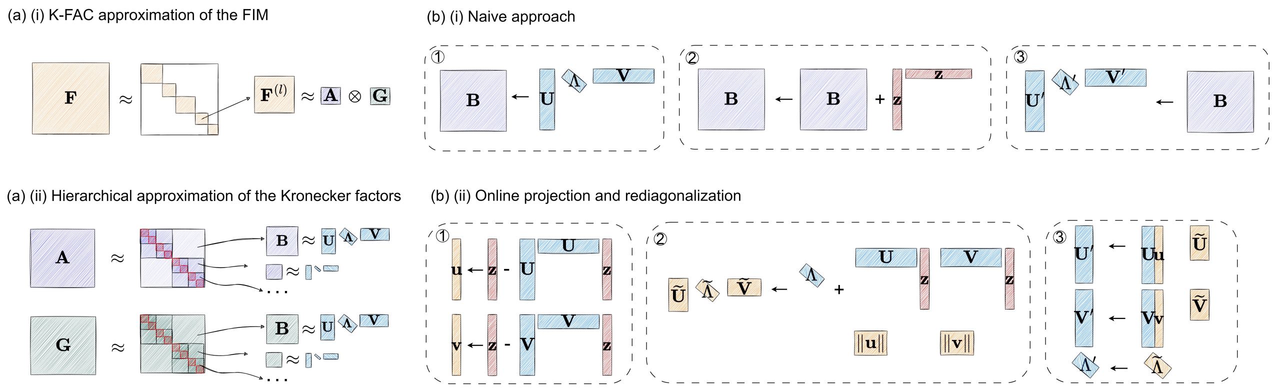
(a) (i) K-FAC approximation of the FIM
(b) (i) Naive approach
(a) (ii) Hierarchical approximation of the Kronecker factors
(b) (ii) Online projection and rediagonalization

Figure 2: (a) K-FAC approximation of the FIM and Sylva's hierarchical approximation of Kronecker factors (b) Comparison between the brute force approach and the proposed online algorithm (projection and rediagonalization) to update the SVD factors in the hierarchical approximations.

storage cost and minimize the computation costs of hierarchically approximating the Kronecker factors (**Challenge 1**), we propose an online algorithm that directly updates the SVD factors without materializing the approximated blocks. For each layer, we save the inputs $\mathcal{I}$ and output gradients $\mathcal{G}$. We use $\mathbf{z}$ to uniformly denote the input $\mathcal{I}$ and output gradients $\mathcal{G}$ since the process is similar. Then, we update the hierarchical approximation of $\mathbf{A}$ and $\mathbf{G}$ using $\mathcal{I}$ and $\mathcal{G}$, respectively. Naively, as shown in Figure 2b(i), we would need to ① materialize the approximated block using the SVD factors, ② add the blocks and ③ do another SVD on the aggregated block. Materializing the approximated blocks leads to high memory footprints, and the computation of SVD is expensive. To address these issues, we propose online projection and rediagonalization (Figure 2b (ii)). There are three main steps in our algorithm: ① projecting the incoming data onto the existing basis ② rediagonalization to get the rotation matrices $\widetilde{\mathbf{U}}$, $\widetilde{\mathbf{\Lambda}}$ and $\widetilde{\mathbf{V}}$, and ③ rotating the extended subspace (concatenation of existing basis ($\mathbf{U}$, $\mathbf{V}$) and the orthogonal vectors ($\mathbf{u}$, $\mathbf{v}$) to obtain the updated SVD factors ($\mathbf{U}'$, $\mathbf{V}'$). The updated singular values $\mathbf{\Lambda}'$ are simply $\widetilde{\mathbf{\Lambda}}$. In a nutshell, our algorithm directly operates on the SVD factors $(\mathbf{U}, \mathbf{\Lambda}, \mathbf{V})$ without materializing the corresponding approximated block $\mathbf{B}$, which has a much larger number of elements than that of the SVD factors. Thus, both computation and storage costs are reduced. Specifically, in the naive approach, ① takes $O(m^2r + mr)$ computation to retrieve the approximated block $\mathbf{B}$ from the SVD factors, ② takes $O(m^2)$ computation to add the retrieved block and new block (outer product of $\mathbf{z}$ and itself), and ③ takes $O(m^3)$ to recompute the SVD of the aggregated block. Throughout the entire process, $O(m^2)$ storage is required. With online projection and rediagonalization, ① obtains the new basis ($\mathbf{u}$ and $\mathbf{v}$) orthogonal to existing subspace (spanned by $\mathbf{U}$ and $\mathbf{V}$) takes $O(mr)$ computation and storage, ② takes $O((r+1)^3)$ computation and $O((r+1)^2)$ storage, since we compute SVD on the lower dimension r instead of the higher dimension m, and ③ to obtain the new SVD factors via rotating the extended subspace takes $O(m(r+1)^2)$ computation and $O(m(r+1))$ storage. Our proposed algorithm has much lower computation and storage costs than the naive approach since $r \ll m$.

3) Embedded Sparse Trainable Weights. To eliminate extra computation introduced by injected trainable weights (**Challenge 2**), we propose to embed the sparse trainable weights into the pre-trained weights during fine-tuning. Specifically, we keep a single copy of weights as shown in Figure 3b, instead of two: one for the frozen weights and one for the trainable weights as the naive approach shown in Figure 3a. This key technique eliminates the extra computation (red rectangles in Figure 3a) due to injecting trainable weights. Furthermore, during the backward pass, we leverage an optimized library that provides fast Sampled Dense-Dense Matrix Multiplication (SDDMM) operation to compute the sparse gradients. The sparse gradients are stored in block sparse matrix format. Specifically, the non-zero blocks are stored contiguously, and there is a mask indicating whether each block is non-zero or not. We scatter and add the sparse gradients to the dense weights in each optimization step. We use a single copy of weights, i.e., the frozen weights and trainable weights are stored in the same dense matrix, while the frozen weights are not updated throughout the fine-tuning. We fuse the optimization steps of layers into one CUDA kernel launch instead of performing a sequence of small updates. Denote the mini-batch size by b and the sequence length by s. Let the sparsity (i.e., percentage of zeros in the sparse weight matrix) be σ. For the naive approach, the extra computations (red rectangles in Figure 3a) take $O(bsm^2n\sigma + bsn)$ in the forward pass and $O(bsmn^2\sigma + bsm)$ in the backward pass, which are completely eliminated thanks to the embedding of sparse trainable weights into the pre-trained weights. Instead, Sylva requires $O(bsm^2n)$ in the forward pass. In the backward pass, Sylva requires $O(bsmn^2)$ computation for the loss with respect to the inputs and $O(bsmn\sigma)$ computation for the loss with respect to the output gradients.

4 SYLVA: DESIGN DETAILS

4.1 A Two-Stage Fine-Tuning Procedure

Based on the ideas that we present in Section 3, we propose a two-stage fine-tuning procedure as shown in Algorithm 1.

Pre-processing. We sample a subset from the fine-tuning dataset to approximate second-order information. The ***number of pre-processing samples*** is a configurable hyperparameter in Sylva

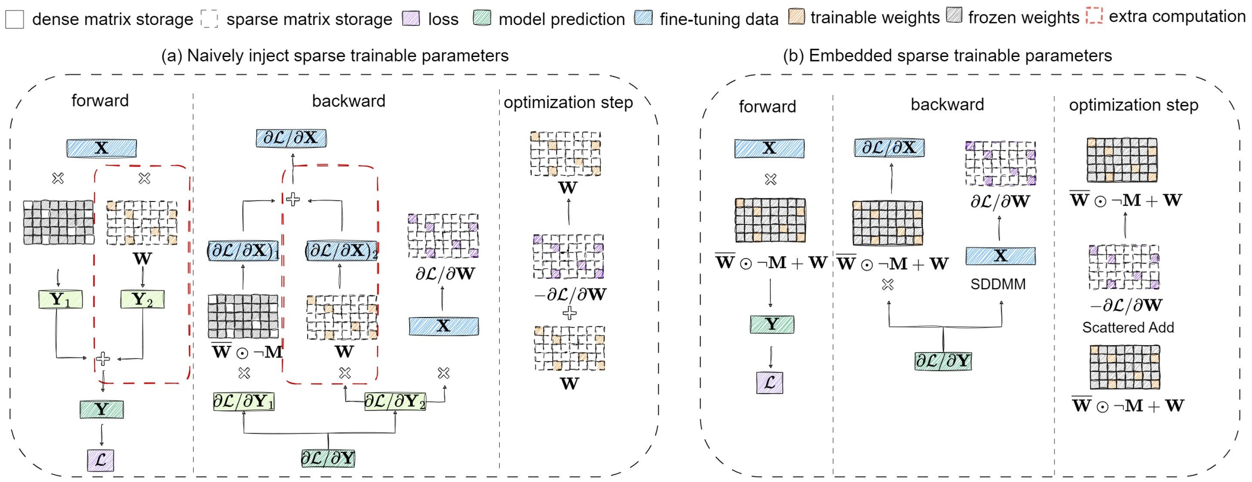

Figure 3: Comparison between naively injecting sparse trainable weights and embedded sparse trainable weights.

(See Section 5.5). We perform a forward-backward pass and store per-sample activation $\mathcal{I}$ and output gradients $\mathcal{G}$ (lines 5 - 7). At the first iteration, we initialize hierarchical approximate of $\mathbf{A}$ and $\mathbf{G}$ (lines 8 - 11) using Algorithm 2. For the following iterations, we update the hierarchical approximations (lines 13 - 15) using online projection and rediagonalization (Algorithm 3).

After one round over the sampled data, we compute the inversion (line 19) by recursively applying Eqn. (5) - (8). We compute the importance of each input dimension using Eqn. (19) and that of each output dimension using Eqn. (20) (line 20). To obtain the element-wise importance of the weights, we compute an outer product of the importance of input and output dimensions. We sort the weights based on their importance to obtain the weight mask and keep the top candidates (line 21). We set the selected weights as trainable and the other weights as frozen, i.e., we compute only the gradients and update the sparse weights (line 21).

Fine-Tuning. During fine-tuning, we compute the forward pass as in Eqn. (21) (line 26). The backward pass includes two parts: (i) computing the gradients of loss with respect to the inputs, as in Eqn. (22), and (ii) compute the gradients of loss with respect to the sparse weights, as in Eqn. (16) (line 27). The sparse gradients are all-reduced in data-parallel training (line 28). In every optimization step, we scatter-add the sparse gradients to the dense weight matrix (line 29). The optimization step takes a list of dense weight matrices, the index of non-zero elements and the sparse gradients stored contiguously.

4.2 Hierarchical Approximate Curvature

Partition of Kronecker Factors. To reduce the storage required by the Kronecker factors ($\mathbf{A}$ and $\mathbf{G}$), we note that the off-diagonal elements of $\mathbf{A}$ and $\mathbf{G}$ stand for the correlation between two dimensions in input or output. Intuitively, the neurons closer to each other in terms of their positions have a higher correlation The ***number of partition*** in the hierarchical approximation is a configurable hyperparameter in Sylva (See Section 5.5). Let k denote the current level of partition and $\mathbf{B}^k$ denote a block at level k. So trivially, $\mathbf{A} = \mathbf{B}^0$. The same holds for $\mathbf{G}$. Leveraging this favourable structural property, we can get the hierarchical approximation for each of the Kronecker factors $\mathbf{A}$ and $\mathbf{G}$ using the following algorithm: (1) We partition a matrix $\mathbf{B}^k$ into four equally sized blocks:

$$\mathbf{B}^k = \begin{bmatrix} \mathbf{B}_{11}^{k+1} & \mathbf{B}_{12}^{k+1} \\ \mathbf{B}_{21}^{k+1} & \mathbf{B}_{22}^{k+1} \end{bmatrix} \tag{1}$$

(2) We approximate the off-diagonal blocks $\mathbf{B}_{12}^{k+1}$ and $\mathbf{B}_{21}^{k+1}$ using Singular Value Decomposition (SVD):

$$\mathbf{B}_{12}^{k+1} \approx \mathbf{U}_{12}^{k+1}\mathbf{\Lambda}_{12}^{k+1}\mathbf{V}_{12}^{k+1\top} \tag{2}$$

$$\mathbf{B}_{21}^{k+1} \approx \mathbf{U}_{21}^{k+1}\mathbf{\Lambda}_{21}^{k+1}\mathbf{V}_{21}^{k+1\top} \tag{3}$$

where $\mathbf{U}$, $\mathbf{V}$ are orthonormal rotation matrices and $\mathbf{\Lambda}$ are diagonal matrices of the top singular values. (3) We recursively partition each of the diagonal blocks $\mathbf{B}_{11}^{k+1}$ and $\mathbf{B}_{22}^{k+1}$ into smaller blocks by repeating steps (1) and (2), until the dimension of the finest partition is smaller than a user-defined threshold.

A hierarchical approximation is (r, κ) if the off-diagonal blocks are approximated using rank-r SVD and the matrix is partitioned κ times.

We denote a (r, κ) hierarchical approximation by $\mathcal{H}_{(r,\kappa)}$. A block at the k-th level partition is

$$\mathbf{B}^k = \begin{bmatrix} \mathbf{B}_{11}^{k+1} & \mathbf{U}_{12}^{k+1}\mathbf{\Lambda}_{12}^{k+1}\mathbf{V}_{12}^{k+1} \\ \mathbf{U}_{21}^{k+1}\mathbf{\Lambda}_{21}^{k+1}\mathbf{V}_{21}^{k+1} & \mathbf{B}_{22}^{k+1} \end{bmatrix} \tag{4}$$

for $k = 0, ..., \kappa$. The dimension of $\mathbf{B}^k$ is $\frac{1}{2^k}$ of the original matrix. The blocks at the finest partition are kept as exact.

Recursive Block Inversion. Given a hierarchical approximation, we can compute the matrix inversion in a bottom-up manner by recursively applying the block inversion formula given the inversion (or SVD) of submatrices. Consider a block $\mathbf{B}$ in the hierarchical approximation. The base case is the inversion of the diagonal blocks. Consider block at the k-th level partition as in Equation 4. Given the inverses of the diagonal blocks $\mathbf{B}_{11}^{-1} = [\mathbf{B}_{11}^{k+1}]^{-1}$ and

Algorithm 1 Sylva's Two-Stage Fine-Tuning Procedure

```
1:  Input: model parameters $\mathbf{W}$, training dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$,
    sparsity $\sigma$, number of sample $N$, number of partition $K$, rank $r$
2:  /* Stage 1. Pre-processing */
3:  $\mathcal{H}_\mathrm{A} \leftarrow null$, $\mathcal{H}_\mathrm{G} \leftarrow null$
4:  for $i = 1$ to $N$ do
5:      draw a mini-batch from the dataset $\mathcal{D}$
6:      compute a forward and backward pass (Eqn. 14 - 16)
7:      save inputs $\mathcal{I}$ and output gradients $\mathcal{G}$
8:      if $\mathcal{H}_\mathrm{A}$ is null or $\mathcal{H}_\mathrm{G}$ is null then
9:          /* Initialize hierarchical approximate curvature */
10:         $\mathcal{H}_\mathcal{I} \leftarrow \mathrm{HAC}(\emptyset, \mathcal{I}, K, 1, r)$
11:         $\mathcal{H}_\mathcal{G} \leftarrow \mathrm{HAC}(\emptyset, \mathcal{G}, K, 1, r)$
12:     else
13:         /* Online projection and rediagonalization */
14:         $\mathcal{H}_\mathrm{A} \leftarrow \mathrm{OPD}(\emptyset, \mathcal{I}, K, r)$
15:         $\mathcal{H}_\mathrm{G} \leftarrow \mathrm{OPD}(\emptyset, \mathcal{G}, K, r)$
16:     end if
17: end for
18: /* Compute weight importance and mask */
19: compute $\mathcal{H}_\mathrm{A}^{-1}, \mathcal{H}_\mathrm{G}^{-1}$ using recursion (Eqn. 5 - 8)
20: compute weight importance $\Omega$ (Eqn. 19 - 20)
21: compute mask $\mathbf{M} = \Omega >$ the $\sigma$-th quantile of $\Omega$
22: set $\mathbf{W} \odot \mathbf{M}$ as trainable weights
23: /* Stage 2. Fine-tuning */
24: while not converged do
25:     draw a mini-batch from the dataset $\mathcal{D}$
26:     compute a forward pass (Eqn. 14 - 16)
27:     compute a sparse backward pass
28:     all-reduce sparse gradients $\mathbf{W} \odot \mathbf{M}$
29:     optimize sparse weights using ScatterAdd (Eqn. 17)
30: end while
```

$\mathbf{B}_{22}^{-1} = [\mathbf{B}_{12}^{k+1}]^{-1}$, we compute the kernel matrix

$$\mathbf{K} = \begin{bmatrix} \mathbf{V}_{21}^\top \mathbf{B}_{11}^{-1} \mathbf{U}_{12} & \Lambda_{12} \\ \Lambda_{21} & \mathbf{V}_{12}^\top \mathbf{B}_{22}^{-1} \mathbf{U}_{21} \end{bmatrix}^{-1}. \tag{5}$$

The dimension of $\mathbf{K}$ is $r \times r$, where r is the rank in the hierarchical approximation. Denote the inversion of diagonal blocks by

$$\mathbf{D} = \begin{bmatrix} \mathbf{B}_{11}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_{22}^{-1} \end{bmatrix} \tag{6}$$

and the rotation matrices by

$$\mathbf{L} = \begin{bmatrix} \mathbf{B}_{11}^{-1} \mathbf{U}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_{22}^{-1} \mathbf{U}_{21} \end{bmatrix},$$

$$\mathbf{R} = \begin{bmatrix} \mathbf{V}_{21}^\top \mathbf{B}_{11}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_{12}^{-1} \mathbf{B}_{22} \end{bmatrix} \tag{7}$$

As a result, the inverse of $\mathbf{B}$ is

$$\mathbf{B}^{-1} = \mathbf{D} - \mathbf{LKR}. \tag{8}$$

4.3 Online Projection and Rediagonalization

To efficiently aggregate second-order information over many mini-batches of training data into the hierarchical approximation of Kronecker factors, we propose an online algorithm that does not need to materialize the approximated blocks. Brand [3] proposes a numerical method for fast additive low-rank modifications for tracking singular values and subspaces. Based on it, we design an online algorithm that directly updates the SVD factors in our hierarchical approximation without re-computing the approximated blocks and maintains the best rank-r approximation greedily. Consider a block $\mathbf{B}$ in the hierarchical approximation. Denote the block after t iterations of update by $\mathbf{B}_t$. In each iteration, given the per-sample input or output $\mathbf{z}$ of the mini-batch, we want to update the singular value decomposition $\mathbf{B}_t = \mathbf{U}_t \Lambda_t \mathbf{V}_t$ to incorporate the rank-1 perturbation $\mathbf{z}\mathbf{z}^\top$ resulting from an incoming data. Firstly, we find the additional basis $\mathbf{u}_t$ and $\mathbf{v}_t$ that are orthogonal to the rotation matrices $\mathbf{U}_t$ and $\mathbf{V}_t$

$$\mathbf{u}_{t+1} = \mathbf{z}_{t+1} - \mathbf{U}_t \mathbf{U}_t^\top \mathbf{z}_{t+1}. \tag{9}$$

Similarly, we have

$$\mathbf{v}_{t+1} = \mathbf{z}_{t+1} - \mathbf{V}_t \mathbf{V}_t^\top \mathbf{z}_{t+1}. \tag{10}$$

Next, we compute the rediagonalization

$$\Psi = \underbrace{\begin{bmatrix} \Lambda_t & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix}}_{①} + \underbrace{\begin{bmatrix} \mathbf{U}_t^\top \mathbf{z}_{t+1} \\ \|\mathbf{u}_{t+1}\| \end{bmatrix} \begin{bmatrix} \mathbf{V}_t^\top \mathbf{z}_{t+1} \\ \|\mathbf{v}_{t+1}\| \end{bmatrix}^\top}_{②} \tag{11}$$

where ① is a diagonal matrix and ② is a rank-1 matrix (outer product of two vectors). We obtain the rotation of extended subspaces by diagonalizing Ψ

$$\Psi = \widetilde{\mathbf{U}}\widetilde{\Lambda}\widetilde{\mathbf{V}}^\top. \tag{12}$$

The updated SVD is then

$$\mathbf{U}_{t+1} = \begin{bmatrix} \mathbf{U}_t & \mathbf{u}_{t+1} \end{bmatrix} \widetilde{\mathbf{U}},$$
$$\Lambda_{t+1} = \widetilde{\Lambda},$$
$$\mathbf{V}_{t+1} = \widetilde{\mathbf{V}}^\top \begin{bmatrix} \mathbf{V}_t & \mathbf{v}_{t+1} \end{bmatrix}^\top. \tag{13}$$

We greedily select the top-r singular values and their corresponding eigenbasis and discard the rest to maintain a limited memory usage, regardless of the number of incoming data. By the Eckart-Young-Mirsky theorem [11, 36], the rank-r SVD provides the optimal approximation for a given r. By the end, we obtain the best rank-r approximation for each block in the hierarchy.

4.4 Embedded Sparse Trainable Weights

To optimize the memory footprint of the model during fine-tuning, we propose to construct sparse adapter modules, which provide fine-grained control over the number of trainable parameters. Denote the weights trained to convergence by $\overline{\mathbf{W}}$. Let the weight mask be $\mathbf{M}$. The injected trainable parameters are denoted as $\mathbf{W}$. We use $\odot$ to denote the element-wise product. The adapter $\mathbf{W}$ and its gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ are sparse. $\mathbf{W}$ is initialized as $\overline{\mathbf{W}} \odot \mathbf{M}$. The rest of the pre-trained weights $\overline{\mathbf{W}} \odot \neg\mathbf{M}$ are kept frozen throughout the fine-tuning. In the forward pass, we compute

$$\mathbf{Y} = (\overline{\mathbf{W}} \odot \neg\mathbf{M})\mathbf{X} + \mathbf{W}\mathbf{X}. \tag{14}$$

In the backward pass, we compute the gradient of loss with respect to the input

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \mathbf{W} \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}^\top + (\overline{\mathbf{W}} \odot \neg\mathbf{M}) \frac{\partial \mathrm{L}}{\partial \mathbf{Y}}, \tag{15}$$

and the gradient of loss with respect to the weight

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}^{\top}\mathbf{X}. \tag{16}$$

In the optimization step, we update the sparse trainable weights with their gradients

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\partial \mathcal{L}}{\partial \mathbf{W}}. \tag{17}$$

FIM-Induced Sparsity. We leverage second-order information to obtain the weight mask $\mathbf{M}$ with fined-grained sparsity. The rank deficiency of Hessian provides source and reasoning of parameter redundancy in trained networks [50]. In other words, the Hessian induces the effective dimension of trained overparameterized networks. Taking advantage of this observation, since we want to maximize the information in sparsified gradients on the effective dimensions of the network, we project the gradients onto Hessian's range, i.e., the space spanned by rows/columns of the Hessian. Denote the gradient of loss with respect to the parameters in the l-th layer by $\mathbf{\Gamma} = \nabla_{\mathbf{W}}\mathcal{L} \in \mathbb{R}^{m\times n}$, where m and n are the input and output dimensions of a layer in the neural network. We use $\Delta\mathbf{\Gamma}$ to represent the difference in the gradient and the targeting sparse gradients. The vectorized gradients and change in gradients are $\boldsymbol{g}$ and $\Delta\boldsymbol{g}$, respectively. We use $\boldsymbol{g}_q$ to denote the gradients corresponding to q-th input or output dimension. We compute the importance of input and output dimensions separately. For each of them, we solve the following constrained optimization problem,

$$\min_{q}\left\{\min_{\delta\Gamma}\left\{\mathrm{Tr}(\Delta\mathbf{\Gamma}^{\top}\mathbf{H}\Delta\mathbf{\Gamma})\right\} \text{ s.t. } \Delta\mathbf{\Gamma}\boldsymbol{e}_q + \boldsymbol{g}_q = 0\right\} \tag{18}$$

where $\mathbf{e}_q$ is a canonical basis, all elements are zeros except that the q-the element is one. Using the Kronecker factorization (Figure 2a(i)), we can rewrite the inner objective as $\mathrm{Tr}(\Delta\mathbf{\Gamma}^{\top}\mathbf{A}\Delta\mathbf{\Gamma}\mathbf{G})$. Solving problem (18) by the Lagrange multiplier, we have the importance metrics for input and output dimensions

$$\frac{\boldsymbol{g}_j^{\top}\mathbf{G}\boldsymbol{g}_j}{[\mathbf{A}^{-1}]_{jj}} \tag{19}$$

and

$$\frac{\boldsymbol{g}_k^{\top}\mathbf{A}\boldsymbol{g}_k}{[\mathbf{G}^{-1}]_{kk}}. \tag{20}$$

The outer product of Eqn. (19) and Eqn. (20) yields an importance score for each of the parameters. We sort the parameters in each layer by descending importance and select the top candidates as trainable weights in fine-tuning.

Sparse Embedded Adapters. To reduce the computation overhead caused by the additional forward-backward propagation through the adapters, we observe that some of the matrix operations can be merged as one, leading to an implicit parameterization and eliminating the redundant computation. Computing Eqn. (14) as is introduces additional computation and storage compared to a standard forward pass, as shown in Figure 3. Specifically, it requires another Sparse Matrix Multiply (SpMM) and activation memory for the intermediate tensors $\overline{\mathbf{W}} \odot \neg\mathbf{M}\mathbf{X}$ and $\mathbf{W}\mathbf{X}$. Instead of storing the frozen and trainable weights separately, we keep a unified dense weight matrix while only computing and storing the gradients for the sparse trainable weights. As a result, the forward pass takes the same computation and activation memory as the standard one

$$\mathbf{Y} = \mathbf{W}\mathbf{X}. \tag{21}$$

Similarly, Eqn. (15) becomes a General Matrix Multiply

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \mathbf{W}\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}^{\top}, \tag{22}$$

instead of an addition of two, as in Eqn. (15). The computation sparse gradient in Eqn. (16) is a Sampled Dense Dense Matrix Multiply (SDDMM). Only the rows of $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$ and columns of $\mathbf{X}$ that correspond to the non-zero elements in $\mathbf{W}$ are loaded to the memory and used to compute the gradients. We use block sparsity to demonstrate the performance benefit in Section 5. The ***block size*** and ***sparsity*** in adapters are configurable hyperparameters in Sylva.

Algorithm 2 Hierarchical Approximate Curvature (HAC)

```
1:  Input: hierarchical approximation $\mathcal{H}$, input or output gradients
    $\mathbf{z}$, number of partition $K$, current partition level $k$, rank $r$
2:  if $k < K$ then
3:      compute sub-block $\mathbf{B}^k$ using $\mathbf{z}$
4:      All-Reduce $\mathbf{B}^k$
5:      /* Approximate off-diagonal blocks using SVD */
6:      $\mathcal{H}^k_{12} \leftarrow \mathrm{SVD}(\mathbf{B}^k_{12}, r), \mathcal{H}_{21} \leftarrow \mathrm{SVD}(\mathbf{B}^k_{21}, r)$ (Eqn. 2)
7:      /* Recursively call HAC on diagonal blocks */
8:      $\mathcal{H}^k_{11} \leftarrow \mathrm{HAC}(\mathcal{H}, \mathbf{B}^k_{11}, K, k+1, r)$
9:      $\mathcal{H}^k_{22} \leftarrow \mathrm{HAC}(\mathcal{H}, \mathbf{B}^k_{22}, K, k+1, r)$
10: end if
11: return $\mathcal{H}$
```

Algorithm 3 Online Projection and Diagonalization (OPD)

```
1:  Input: hierarchical approximation $\mathcal{H}$, input or output gradients
    $\mathbf{z}$, number of partition $K$, rank $r$
2:  for $k$ in $1, \ldots, K$ do
3:      $\mathbf{U}_t, \mathbf{\Lambda}_t, \mathbf{V}_t \leftarrow \mathcal{H}^k_{12}$
4:      /* Projection */
5:      compute $\mathbf{u}_{t+1}$ orthogonal to $\mathbf{U}_t$ (Eqn. 9)
6:      compute $\mathbf{v}_{t+1}$ orthogonal to $\mathbf{V}_t$ (Eqn. 10)
7:      /* Rediagonalization */
8:      compute rotation matrices $\widetilde{\mathbf{U}}, \widetilde{\mathbf{\Lambda}}, \widetilde{\mathbf{V}}$ (Eqn. 11-12)
9:      /* Update SVD factors */
10:     $\mathbf{U}_{t+1} \leftarrow$ rotating $[\mathbf{U}_t, \mathbf{u}_{t+1}]$ using $\widetilde{\mathbf{U}}$ (Eqn. 13)
11:     $\mathbf{V}_{t+1} \leftarrow$ by rotating $[\mathbf{V}_t, \mathbf{v}_{t+1}]$ using $\widetilde{\mathbf{V}}$ (Eqn. 13)
12:     $\mathbf{\Lambda}_{t+1} \leftarrow \widetilde{\mathbf{\Lambda}}$ (Eqn. 13)
13:     /* Keep the best rank-$r$ approximation */
14:     $\mathcal{H}^k_{12} \leftarrow \mathbf{U}_{t+1}[:, :r],\ \mathbf{\Lambda}_{t+1}[:r],\ \mathbf{V}_{t+1}[:r, :]$
15:     repeat lines for $\mathcal{H}^k_{21}$
16: end for
17: return $\mathcal{H}$
```

5 EVALUATION

5.1 Methodology

Model and Datasets. We use four set of benchmarks: (i) RoBERTa-Large (355M parameters) [32] with the GLUE benchmark [55].

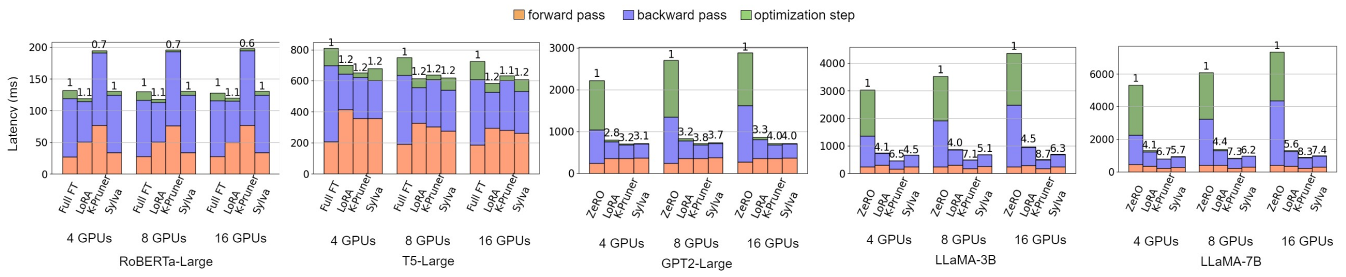

Figure 4: Fine-tuning time per iteration (Y-axis) on GPT2-Large, LLaMA-3B, and LLaMA-7B, as the number of GPUs (X-axis) increases from 4 to 16. The fine-tuning time is broken down into forward pass time (in orange), backward pass time (in purple) and optimization step time (in green).

Table 2: Comparison of Peak GPU memory consumption (GB) during fine-tuning on various rank r and sparsity σ (%).

	GPT2-L					LLaMA-3B					LLaMA-7B			
r/σ (%)	ZeRO	LoRA	Sylva	K-Pruner	r/σ (%)	ZeRO	LoRA	Sylva	K-Pruner	r/σ (%)	ZeRO	LoRA	Sylva	K-Pruner
32/95	12.5	12.4	**12.1**	**12.1**	96/95	14.1	11.9	10.5	**6.8**	64/98	14.6	16.7	15.4	**13.0**
128/80	-	13.4	12.8	**12.7**	192/90	-	13.9	11.5	**6.8**	128/95	-	20.3	17.6	**13.0**
512/50	-	19.4	15.3	**13.9**	384/80	-	20.8	13.3	**8.3**	256/90	-	OOM	20.3	**13.1**

(ii) GPT2-Large (812M parameters) [40] with the E2E NLG challenge [39]. (iii) T5-Large (738M parameters) [41] with the Super-NaturalInstructions [57] dataset. (iv) LLaMA-3B and 7B [53] on the OpenAssistant Conversations Dataset (OASST1) [25].

Baselines. We compare with (1) full fine-tuning, (2) LoRA [20] as the adapter baseline, and (3) K-FAC pruner [56] as the pruner baseline. For the performance evaluation of full fine-tuning baseline, we use ZeRO optimizer [42, 43] for GPT2-Large and LLaMA, and gradient checkpointing [7] for T5-Large. For RoBERTa-Large, there is no memory optimization needed, and we simply do full fine-tuning in PyTorch [38]. We choose the memory optimization that provides better fine-tuning time and does not result in out-of-memory (OOM) errors. The choice of memory optimization does not affect model quality.

Hardware. We use `g2-standard-48` instances on Google Cloud Platform (GCP). Each node has 4 NVIDIA L4 GPUs (24 GB) and an Intel Cascade Lake CPU. We evaluate all methods on up to 16 GPUs. All instances are located in the same regions and connected via a network bandwidth of around 15 Gbps.

Software. We use PyTorch 2.1.0 [38] for all methods and DeepSpeed 0.10.0 [44] for the ZeRO optimizer. To showcase the performance benefit of leveraging sparsity, we use Triton block sparse module [47, 52], which provides CUDA kernels for block sparse SpMM and SDDMM operations. Note that although we use block sparsity to demonstrate Sylva's performance benefit, our method is not limited to block sparse patterns and can benefit from other CUDA kernels or hardware-enabled acceleration.

5.2 Fine-Tuning Time

We compare the end-to-end fine-tuning time per iteration using up to 16 GPUs. Sylva is, on average, 5.1× faster than ZeRO optimizer on GPT2-Large and LLaMA. As shown in Figure 4, ZeRO is significantly slower than LoRA and Sylva at a typical data center network bandwidth (10 ∼ 20 Gbps) without ultra-high bandwidth interconnects (e.g. InfiniBand EDR 100Gb/s [35]). This is because ZeRO trades off latency for larger memory efficiency. ZeRO stage 1 partitions optimizer states across the GPUs, and stage 2 additionally shards reduced gradients. Both need additional gathering for every optimization step. Besides, the communication overhead is exacerbated as we increase the number of GPU nodes. This is because ZeRo adds communication overheads in both backward and optimization steps, thus increasing execution time Sylva is 1.2× faster than full fine-tuning with gradient checkpointing enabled on T5-Large. The forward pass time in K-FAC Pruner is the best on GPT2-Large and LLaMA because it uses sparse weight matrices, significantly reducing computation costs. The forward and backward pass time of Sylva is 1.2 ∼ 1.4× faster than LoRA on GPT2-Large and LLaMA. This is because Sylva eliminates extra computation led by the injected trainable parameters in LoRA. On T5-Large, LoRA and Sylva perform similarly. On RoBERTa-Large and T5-Large, K-FAC Pruner performs worse than LoRA and Sylva, because the layer dimensions are relatively small. Also, there is a trade-off between the performance gain from reduced computation FLOPs and the overhead of performing irregular memory access in the memory hierarchy. On RoBERTa-Large, LoRA performs slightly better than Sylva, specifically on the backward pass, because Sylva uses sparse instead of dense matrices, which causes irregular memory access. The optimization step time is relatively small compared to forward/backward pass time in all methods except for ZeRO.

5.3 Model Quality

Our method achieves comparable or slightly worse model quality compared to full fine-tuning and LoRA at a high sparsity in the range of 95% ∼ 99% (Table 3). As we show in Section 5.5, Sylva performs better using a moderate sparsity in the range of 86% ∼ 96%, while LoRA obtains its best scores with a small rank of 4. K-FAC Pruner degrades model quality because it uses sparse weights, which do not fully utilize learned knowledge in pre-trained weights.

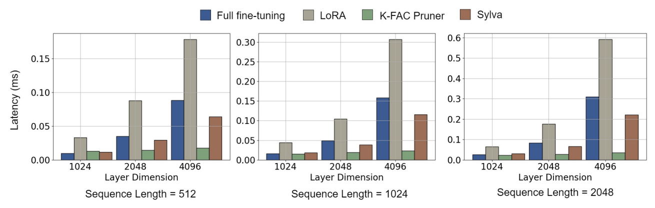

Figure 5: Latency of forward-backward propagation and optimization step for a single layer at various layer dimensions and sequence length.

5.4 Memory Footprint

We compare the peak GPU memory usage on various rank r (for LoRA) and sparsity σ (for K-FAC Pruner and Sylva). As shown in Table 2, Sylva requires less GPU memory than LoRA on all models. The advantage of Sylva grows as sparsity decreases, i.e., if we inject a larger number of trainable weights into LLMs. For example, LoRA causes OOM errors on LLaMA-7B with a rank of 256, while Sylva can fit onto a 24 GB GPU in an equivalent configuration ($\sigma = 90\%$). This is because LoRA requires storing intermediate results provided by both the frozen pre-trained weights and injected trainable weights and then taking the sum of them. In contrast, Sylva only requires one pass on the embedded weights and thus needs to store fewer intermediate results. K-FAC Pruner requires the least GPU memory during fine-tuning. However, their pre-processing step requires excessive GPU memory and might lead to OOM errors on the GPUs, as we show later (See Section 5.6 Pre-Processing Costs).

5.5 Sensitivity Study

Model Quality. We compare the hyperparameter sensitivity of LoRA and Sylva on GPT2-Large model and the E2E NLG Challenge. We control Sylva's sparsity and LoRA's rank together so that the number of trainable parameters is the same during fine-tuning. We also study Sylva's sensitivity to block size, the number of partitions in the hierarchical approximation, and the number of samples in the pre-processing step. (i) *Sparsity:* Sylva obtains better model quality than LoRA with sparsity in the range of 86.7% ∼ 96.7% (Table 4). At a sparsity of 99.5%, Sylva obtains a worse model quality than LoRA (with a rank of 4). (ii) *Block size:* As shown in Table 5, the quality slightly decreases as the block size increases. A moderate block size such as 64 balances the model quality and performance tradeoff. Specifically, it obtains high model quality while being 10% faster than a block size of 16 and only 4% slower compared to a block size of 128. With a high sparsity of over 90%, the end-to-end latency of different block sizes is comparable. (iii) *Number of Partition:* The model quality decreases as we increase the number of partitions (Table 6). We prioritize a smaller number of partitions, because it yields a more accurate approximation and has less pre-processing time as long as it is able to fit onto the GPU memory. (iv) *Number of Samples in Pre-Processing:* On average, the model quality is the highest with 1024 samples (Table 7). We noticed that Sylva's highest score for MET and ROUGE-L is obtained with 128 samples, although the average score with 128 samples is lower than that obtained with 1024 samples. This indicates that more pre-processing samples provide richer information on the dataset, thus yielding a higher average score.

Performance. Besides comparing the end-to-end fine-tuning time, we benchmark the forward/backward pass computation time for a single Linear module (common in attention or FC modules) on various layer dimensions and sequence lengths. As shown in Figure 5, the advantage of Sylva over full fine-tuning and LoRA enlarges as the layer dimensions increase and the maximum sequence length increases. Sylva is up to 2.3× faster than LoRA and 1.3× faster than full fine-tuning. This is because the computation complexity for activations increases linearly to the layer dimensions and sequence lengths. K-FAC Pruner has the lowest latency because it uses sparse weight matrices. Thus, computation in forward/backward passes is sparse matrix operations (SpMM or SDDMM) instead of dense. However, K-FAC Pruner sacrifices model quality, as we shown in Section 5.3. Note that the discrepancy between Figure 4 and 5 is because there are layers that LoRA, K-FAC Pruner and Sylva do not optimize (e.g., embedding, normalization), which results in lower end-to-end speedup compared to that on a single layer.

Model	Metric	Full FT	LoRA	K-Pruner	Sylva
RoBERTa-Large	MNLI	90.3	**90.4**	**90.2**	90.3
	SST-2	**96.4**	95.9	95.5	96.0
	MRPC	90.9	90.7	89.7	**91.9**
	CoLA	68.0	**68.1**	65.6	68.0
	QNLI	94.7	**94.8**	93.6	94.2
	QQP	**92.2**	91.4	90.2	90.9
	RTE	**86.3**	85.4	77.5	85.3
	STS-B	**92.2**	91.9	91.1	91.3
T5-Large	ROUGE-L	**47.8**	47.7	40.1	47.6
GPT2-Large	BLEU	68.2	**68.5**	64.7	66.5
	NIST	**8.8**	**8.8**	7.9	8.5
	MET	**46.7**	46.0	43.5	45.1
	ROUGE-L	**71.9**	69.5	64.0	69.2
	CIDEr	**2.4**	**2.4**	2.1	2.3
LLaMA-7B	MMLU	**36.8**	36.6	33.1	36.1

Table 3: Model quality between full fine-tuning, LoRA, K-FAC pruner and Sylva on RoBERTa-Large, T5-Large, GPT2-Large and LLaMA-7B.

5.6 Pre-Processing Costs

Sylva reduces the peak GPU memory by up to 2.3× compared to K-FAC Pruner (Table 2). For LLaMA-7B, the pre-trained model weights and Kronecker factors alone take 48 GB of GPU memory, without considering the activation memory and gradients allocated in the pre-processing step. Our hierarchical approximation and algorithm reduce peak GPU memory usage to 20.1 GB during computing second-order information. Since the pre-trained weights take about 13 GB, the storage is reduced by at least 5.0× compared to naively computing the Kronecker factors. To approximate the second-order information in the pre-processing step, we sample about 10% points of the dataset. For example, the pre-processing step of GPT2-Large takes 17.2 minutes on 4 GPUs, while fine-tuning it for 5 epochs takes about 3 hours. Thus, Sylva's pre-processing does not add significant overhead overall.

5.7 Inference Performance

Since the adapters are merged back to the pre-trained weights after fine-tuning for both LoRA and Sylva, LoRA and Sylva have a similar inference performance to full fine-tuning. The K-FAC

Method	r / σ (%)	BLEU	NIST	MET	ROUGE-L	CIDEr
LoRA	128	67.8	8.6	**46.4**	69.1	**2.4**
Sylva	86.7	**68.8**	**8.8**	46.0	**69.9**	**2.4**
LoRA	96	68.0	**8.7**	**46.3**	69.4	**2.4**
Sylva	90	**68.8**	**8.7**	46.2	**69.6**	**2.4**
LoRA	32	67.8	8.6	**46.2**	**69.1**	**2.4**
Sylva	96.7	**68.4**	**8.8**	**46.2**	69.0	**2.4**
LoRA	4	**68.5**	**8.7**	**46.3**	**69.8**	**2.4**
Sylva	99.5	66.7	8.6	45.4	68.6	2.4

Table 4: Sensitivity of model quality to rank for LoRA and sparsity for Sylva.

Method	Block Size	BLEU	NIST	MET	ROUGE-L	CIDEr
Sylva	16	**68.8**	**8.7**	**46.2**	**69.7**	**2.4**
	64	**68.8**	**8.7**	**46.2**	69.6	**2.4**
	128	67.6	**8.7**	45.4	68.3	2.3

Table 5: Sensitivity of model quality to block size in Sylva.

Method	# Partition	BLEU	NIST	MET	ROUGE-L	CIDEr
Sylva	4	**68.8**	**8.7**	**46.2**	**69.6**	**2.4**
	6	68.2	8.6	45.2	68.9	2.3
	8	67.1	8.5	44.7	67.9	2.3

Table 6: Sensitivity of model quality to the number of partitions in the hierarchical approximation of Sylva.

Method	# Samples	BLEU	NIST	MET	ROUGE-L	CIDEr
Sylva	128	67.8	8.6	**46.3**	**69.9**	2.3
	256	67.5	8.7	45.9	69.1	**2.4**
	1024	**68.8**	**8.8**	46.0	**69.9**	**2.4**

Table 7: Sensitivity of model quality to the number of samples used in the pre-processing step of Sylva.

pruner removes a subset of the pre-trained weights in the pre-processing step, yielding a sparse model after fine-tuning. Thus, K-FAC pruner obtains 1.4× speedup during inference compared to the other methods on LLaMA-7B.

6 RELATED WORK

To our knowledge, Sylva is the first work that significantly reduces (i) computation costs in LLM fine-tuning by embedding sparse trainable weights into the pre-trained LLM, and (ii) storage costs to approximate second-order information via a hierarchical approximation, and online projection and rediagonalization algorithm, while also providing high model quality.

Pruners using Second-Order Information. A few prior works [18, 26] employ Hessian to prune weight matrices. Hessian is shown to be closely related to the source of rank deficiency in pre-trained deep neural networks [50]. However, using Hessian as a second-order information method results in excessive memory footprints in the pre-processing step, making it prohibitive to be used in common data-center GPUs. K-FAC [17, 34, 56] uses the Fisher Information Matrix (FIM) to approximate the Hessian and significantly reduces the memory footprints needed during the pre-processing step. The FIM is shown to be equivalent to the Hessian for commonly used loss (e.g. cross-entropy, squared-error) [33]. In our evaluation, we show that Sylva reduces storage costs compared to K-FAC. Frantal et al. [12, 13] use the layer-wise reconstruction loss to identify redundancy between elements of weight matrices. Still, there is no theoretical result that shows optimizing this reconstruction loss is as good as that proposed in prior works [18, 26]. A recent work [28] approximates pre-trained weights via a combination of low-rank and sparse matrix. Finally, [24] proposes a mask search, rearrangement and tuning technique for fast post-training pruning. Although their method is guided by FIM, it uses a crude diagonal approximation and disregards correlations between weight elements.

Adapters for Parameter-Efficient Fine-Tuning. Houlsby et al. [19] proposes parameter-efficient transfer learning via injecting new layers after every submodule consisting of a few layers. This significantly reduces the number of trainable parameters and thus accelerates fine-tuning compared to full fine-tuning. However, this adds computation overheads to the fine-tuned model since the number of parameters increases. A few prior works [31, 46] attempt to improve on Houlsby et al. by reducing the adapter size, however, at the price of compromising model quality. LoRA [20] is the state-of-the-art adapter approach, that reduces GPU memory usage, attains high model quality and is easy to use. Recent works [10, 59] improves upon LoRA by dynamically controlling rank in adapters throughout fine-tuning. Dettmers et al. [8] further reduces GPU memory usage by quantizing the frozen pre-trained weights. We compare with LoRA in the evaluations (Section 5).

Other LLM Fine-Tuning Approaches. Sung et al.[51] use Fisher diagonal to extract a fixed sparse mask on the pre-trained LLM, and identify a set of sparse masked weights (i.e., a subset of the pre-trained LLM weights) to be re-trained in fine-tuning. This method has been shown to be effective only in $\text{BERT}_{\text{LARGE}}$ model with 345 million parameters. Furthermore, there is no efficient implementation to take advantage of the sparsity in the computation.

Efficient Sparse Linear Algebra Kernels for GPUs. Prior works[5, 14, 30] have proposed software optimizations using CUDA for sparse linear algebra (e.g., Sparse Matrix Multiply) for GPU architectures. Sylva can work synergistically with these prior works, i.e., integrating optimized sparse computation kernels to improve performance on GPUs further.

7 CONCLUSION

In this work, we propose **Sylva**, a novel approach that accelerates the ubiquitous fine-tuning process that adapts pre-trained models to downstream tasks. We design a hierarchical approximation of the second-order information, and an online projection and rediagonalization algorithm to significantly reduce the pre-processing costs to identify important weights compared to K-FAC, the most widely used approximation to the Fisher Information Matrix. We effectively embed the sparse trainable weights into the pre-trained LLM weights to eliminate computation overheads introduced in adapters. Sylva's end-to-end fine-tuning is on average 5.1× faster than ZeRO, a memory efficient optimizer for full fine-tuning, and on average 1.2× faster than LoRA, the state-of-the-arts adapter approach for fine-tuning LLMs. In comparison to K-FAC Pruner, Sylva's hierarchical approximation reduces the peak GPU memory by 2.3× in the pre-processing step and provides better model quality after fine-tuning. We conclude that Sylva is a highly efficient fine-tuning method for LLMs and we hope that our work encourages further studies on optimizing the execution (training and inference) of LLMs as well as other deep neural networks.

ACKNOWLEDGMENTS

We thank all UofT EcoSystem research group members for the technical discussions and feedback on this paper. Special thanks to Yaoyao Ding for assistance with compiler techniques and CUDA programming, and Zhanda Zhu for sharing his insights into various LLM training techniques.

REFERENCES

[1] Shun-ichi Amari. 1998. Natural Gradient Works Efficiently in Learning. *Neural Computation* 10, 2 (02 1998), 251–276. https://doi.org/10.1162/089976698300017746

[2] Fengxiang Bie, Yibo Yang, Zhongzhu Zhou, Adam Ghanem, Minjia Zhang, Zhewei Yao, Xiaoxia Wu, Connor Holmes, Pareesa Golnari, David A. Clifton, Yuxiong He, Dacheng Tao, and Shuaiwen Leon Song. 2023. RenAIssance: A Survey into AI Text-to-Image Generation in the Era of Large Model. arXiv:2309.00810 [cs.CV]

[3] Matthew Brand. 2006. Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra Appl.* 415, 1 (2006), 20–30. https://doi.org/10.1016/j.laa.2005.07.021 Special Issue on Large Scale Linear and Nonlinear Eigenvalue Problems.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., Vancouver, Canada, 1877–1901.

[5] Roberto L. Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B. Fraguela, and Torsten Hoefler. 2023. VENOM: A Vectorized N:M Format for Unleashing the Power of Sparse Tensor Cores. arXiv:2310.02065 [cs.DC]

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG]

[8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. In *Advances in Neural Information Processing Systems*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., New Orleans, USA, 10088–10115.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Ning Ding, Xingtai Lv, Qiaosen Wang, Yulin Chen, Bowen Zhou, Zhiyuan Liu, and Maosong Sun. 2023. Sparse Low-rank Adaptation of Pre-trained Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 4133–4145. https://doi.org/10.18653/v1/2023.emnlp-main.252

[11] Carl Eckart and G. Marion Young. 1936. The Approximation of One Matrix by Another Of Lower Rank. *Psychometrika* 1 (1936), 211–218.

[12] Elias Frantar and Dan Alistarh. 2022. Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., New Orleans, USA, 4475–4488.

[13] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, Honolulu Hawaii USA, 10323–10337.

[14] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Atlanta, Georgia, Article 17, 14 pages.

[15] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. 2019. An Investigation into Neural Net Optimization via Hessian Eigenvalue Density. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, USA, 2232–2241.

[16] Erin Griffith. 2023. *The Desperate Hunt for the A.I. Boom's Most Indispensable Prize.* The New York Times. https://www.nytimes.com/2023/08/16/technology/ai-gpu-chips-shortage.html

[17] Roger Grosse and James Martens. 2016. A Kronecker-factored approximate Fisher matrix for convolution layers. arXiv:1602.01407 [stat.ML]

[18] Babak Hassibi, David Stork, and Gregory Wolff. 1993. Optimal Brain Surgeon: Extensions and performance comparisons. In *Advances in Neural Information Processing Systems*, J. Cowan, G. Tesauro, and J. Alspector (Eds.), Vol. 6. Morgan-Kaufmann, Denver, USA.

[19] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, USA, 2790–2799.

[20] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL]

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Vancouver, Canada. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[22] Ryo Karakida, Shotaro Akaho, and Shun-ichi Amari. 2019. Universal Statistics of Fisher Information in Deep Neural Networks: Mean Field Approach. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 89)*, Kamalika Chaudhuri and Masashi Sugiyama (Eds.). PMLR, Naha, Okinawa, Japan, 1032–1041.

[23] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. 2022. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 4163–4181. https://doi.org/10.18653/v1/2022.emnlp-main.279

[24] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. 2022. A Fast Post-Training Pruning Framework for Transformers. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., New Orleans, 24101–24116. https://proceedings.neurips.cc/paper_files/paper/2022/file/987bed997ab668f91c822a09bce3ea12-Paper-Conference.pdf

[25] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. 2023. OpenAssistant Conversations – Democratizing Large Language Model Alignment. arXiv:2304.07327 [cs.CL]

[26] Yann Le Cun, John S. Denker, and Sara A. Solla. 1989. Optimal Brain Damage. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems (NIPS'89)*. MIT Press, Cambridge, MA, USA, 598–605.

[27] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[28] Yixiao Li, Yifan Yu, Qingru Zhang, Chen Liang, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2023. LoSparse: structured compression of large language models based on low-rank and sparse approximation. In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*. JMLR.org, Honolulu, Hawaii, USa, Article 839, 15 pages.

[29] Zhenyu Liao and Michael W Mahoney. 2021. Hessian Eigenspectra of More Realistic Nonlinear Models. In *Advances in Neural Information Processing Systems*,

M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., Virtual, 20104–20117.

[30] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, and Fan Yang. 2023. Efficient GPU Kernels for N:M-SPARSE Weights in Deep Learning. In *Sixth Conference on Machine Learning and Systems (MLSys'23)*. Machine Learning and Systems, Miami, USA.

[31] Zhaojiang Lin, Andrea Madotto, and Pascale Fung. 2020. Exploring Versatile Generative Language Model Via Parameter-Efficient Transfer Learning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 441–459. https://doi.org/10.18653/v1/2020.findings-emnlp.41

[32] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Ro{BERT}a: A Robustly Optimized {BERT} Pretraining Approach.

[33] James Martens. 2020. New Insights and Perspectives on the Natural Gradient Method. *Journal of Machine Learning Research* 21, 146 (2020), 1–76.

[34] James Martens and Roger Grosse. 2015. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 2408–2417.

[35] Nvidia Mellanox. 2014. Introducing EDR 100Gb/s - Enabling the Use of Data. https://network.nvidia.com/pdf/whitepapers/wp_introducing_edr_100gb_enabling_use_data.pdf

[36] L. MIRSKY. 1960. Symmetric Gauge Functions And Unitarily Invariant Norms. *The Quarterly Journal of Mathematics* 11, 1 (Jan 1960), 50–59. https://doi.org/10.1093/qmath/11.1.50

[37] 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. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 721, 12 pages.

[39] Yevgeniy Puzikov and Iryna Gurevych. 2018. E2E NLG Challenge: Neural Models vs. Templates. In *Proceedings of the 11th International Conference on Natural Language Generation*, Emiel Krahmer, Albert Gatt, and Martijn Goudbeek (Eds.). Association for Computational Linguistics, Tilburg University, The Netherlands, 463–471. https://doi.org/10.18653/v1/W18-6557

[40] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. *Language Models are Unsupervised Multitask Learners.* Open AI.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[42] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. ArXiv.

[43] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[44] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[45] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. arXiv:2112.10752 [cs.CV]

[46] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. 2021. AdapterDrop: On the Efficiency of Adapters in Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 7930–7946. https://doi.org/10.18653/v1/2021.emnlp-main.626

[47] Alec Radford Scott Gray and Diederik P. Kingma. 2017. GPU Kernels for Block-Sparse Weights. https://cdn.openai.com/blocksparse/blocksparsepaper.pdf

[48] Anton Shilov. 2023. *TSMC: Shortage of Nvidia's AI GPUs to Persist for 1.5 Years.* Tom's Hardware. https://www.tomshardware.com/news/tsmc-shortage-of-nvidias-ai-gpus-to-persist-for-15-years

[49] Sidak Pal Singh and Dan Alistarh. 2020. WoodFisher: Efficient Second-Order Approximation for Neural Network Compression. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., Virtual, 18098–18109.

[50] Sidak Pal Singh, Gregor Bachmann, and Thomas Hofmann. 2021. Analytic Insights into Structure and Rank of Neural Network Hessian Maps. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., Virtual, 23914–23927.

[51] Yi-Lin Sung, Varun Nair, and Colin A Raffel. 2021. Training Neural Networks with Fixed Sparse Masks. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., Virtual, 24193–24205.

[52] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[53] 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. 2023. LLaMa: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., Long Beach, USA.

[55] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, Tal Linzen, Grzegorz Chrupała, and Afra Alishahi (Eds.). Association for Computational Linguistics, Brussels, Belgium, 353–355. https://doi.org/10.18653/v1/W18-5446

[56] Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. 2019. EigenDamage: Structured Pruning in the Kronecker-Factored Eigenbasis. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, USA, 6566–6575.

[57] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. 2022. Super-NaturalInstructions: Generalization via Declarative Instructions on 1600+ NLP Tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 5085–5109. https://doi.org/10.18653/v1/2022.emnlp-main.340

[58] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. 2023. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. *CoRR* abs/2304.13712 (2023).

[59] Bowen Zhao, Hannaneh Hajishirzi, and Qingqing Cao. 2024. APT: Adaptive Pruning and Tuning Pretrained Language Models for Efficient Training and Inference. arXiv:2401.12200 [cs.CL]