Minimizing Data Movement in Machine Learning Systems

by

Saurabh Agarwal

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 08/23/2024

To my family, friends and the kindness of strangers

ACKNOWLEDGMENTS

First and foremost I will like to thank my advisors - Dimitris Papailiopoulos and Shivaram Venkataraman. They spent countless hours, teaching me how to ask questions and than converting those questions into research. Beyond just research, they have spent incredible amount of time and energy painstakingly providing feedback and editing manuscripts, presentations and posters. Further, they have always provided me freedom to pursue research directions of my own and have been more than benevolent when those directions returned no results. Any good I do in future will have very strong imprints of my advisors.

I am also greatful to Stephen Wright, Remzi Arpaci-Dusseau and Kangwook Lee for serving on my commitee and helping with completing this dissertation. I will also like to thank Theo Rekatsinas for several insightful conversations and often providing me research directions.

At Madison I had pleasure collaborating with several researchers both in Shivaram's and Dimitris's group: Song Bian, Rutwik Jain, Nayong Lee, Shashank Ranjput, Hongyi Wang, Minghao Yan, Liu Yang I would also like to thank the systems research group at UW-Madison to put up with preliminary versions of my conference talks and provide extremely valuable feedback.

I will like to thank several collaborators I had the pleasure to work outside of UW-Madison. During the summer of 2019, I spent the summer working at Apple with Yucheng Low and Sri Krishna Sridhar. I am really greatful for insightful conversation with Yucheng and the time he took to teach me about system design for Machine Learning.

During Summer of 2021 I interned at Microsoft working with Amar Phanishayee. Amar will always be a very strong influence on my research and my research style. Amar taught me several invaluable lessons on managing and approaching research.

I also spent Summer of 2023 working with SysML group at Meta FAIR, working with Bilge Acun. The group consists of some the smartest and kindest individuals I have met.

Apart from research I had a lot of fun living in Madison and made some great friends. I will especially like to thank Christina Healy, who truly transformed me as a person and forced me to pick hobbies and explore the world beyond my desk. I would also like to thank Neydis Moreno for invaluable friendship and company and for always being ready to accompany me on unplanned adventures. Beyond this I will really like to thank the several friends I made skiing at Granite Peak and Cascade Mountain who always provided great ski lift conversations. I will also like to thank several friends I made while learning, teaching and racing keelboats at Hoofer Sailing Club. Their company has been invaluable, without them, I probably would have graduated a year earlier but very likely at the cost of my happiness. I especially will like to thank Keith, Dana and Lisa and others who crew during races on Scurvy Dog.

Finally, I will like to thank my parents and sister who have been source of constant support.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Rapid deployment of Machine Learning (ML) applications like recommendation engines, chat-bots and image synthesis application have made them a dominant workload. These applications are being powered by ML models of increasingly expansive scale, with models comprising of trillions of parameters becoming quite common.

Due to massive compute requirements, ML models are exclusively trained on specialized accelerators and often in a distributed setting. However, a closer analysis of compute utilization shows that ML models are not fully utilizing the compute available on these accelerators. The primary reason for this poor compute utilization is data movement bottlenecks. In this dissertation we primarily focus on data movement bottlenecks associated with intermediate activations.

First, we study Gradient Compression, an approach to minimize the amount of synchronization. In *Accordion* we retrofit existing compression algorithms to automatically vary the amount of compression to reduce the communication during training. Next, we study lack of wall clock speedups when using gradient compression algorithms in *On the utility of Gradient Compression* and propose several guidelines which can be used to design new gradient compression algorithms.

The second part of this dissertation studies distributed training of recommendation models, where we introduce *Bagpipe* a system to minimize embedding access overhead in distributed training.

Finally, we introduce *Clustered Head Attention*, in which we aim to reduce the memory bandwidth bottlenecks of multi-head attention by identifying attention heads with similar output at inference time.

Part I

Introduction

1 INTRODUCTION

Modern Machine Learning (ML) has ushered an era of a completely new set of applications. From ubiquitous tasks like suggesting products which a user might buy [129] to incredibly challenging tasks such as performing image editing [112] and generating new images from text description [159] all depend on some underlying ML models. However, in conjunction with increase in modelling complexity and dataset sizes, the model sizes have increased rapidly.

Training complex machine learning models to state-of-the-art accuracy levels on large datasets [41, 101] is performed on specialized accelerators and often in distributed setting. However, upon close observation prior works have observed that these specialized accelerator have utilization of less than 50% [82, 200]. On of the primary reasons for this poor utilization is data movement and synchronization overheads. There are primarily three different types of data movement - (i) Training data, where training data needs to be pre-processed and moved to the GPU, (ii) Model Parameters, requires moving model parameters from storage or main memory to the GPU, (iii) Intermediate activation, requires moving intermediate activations from GPU memory to registers and in case of distributed training the activations need to be synchronized. In this dissertation we primarily focus on data movement bottlenecks associated with intermediate activations and model parameters.

In the first part we study Gradient compression, a popular approach to reduce synchronization overhead in distributed training in distributed data parallel setting. Distributed data parallel SGD is one of the most common approach for distributed computing, one iteration of distributed data parallel SGD (DDP) comprises two main phases: gradient computation and gradient aggregation. During the computation phase, the gradient of the model is typically computed using backpropagation. This is followed by an aggregation phase, where gradients are synchronously averaged among all participating nodes [75, 55]. During this second phase, for state-of-the-art neural network models, millions to billions of parameters are communicated among nodes [24], which has been shown to lead to communication bottlenecks [40, 149, 132, 57, 16]. Alleviating these communication bottlenecks has been an active area of research in recent years. One extremely popular approach to reduce the communication bottleneck is to perform lossy gradient compression [150, 16, 199, 20, 5]. All the lossy gradient compression methods require users to specify an additional hyper-parameter that determines the degree of compression or sparsification before training begins. In this dissertation we first present Accordion [9]. Accordion adaptively chooses compression parameters, by determining critical periods of ML training. It keeps low compression ratio (high accuracy) during critical periods and high compression ratio otherwise. This allows Accordion to balance both accuracy and amount of communication.

Concurrent to the work on gradient compression,a number of system-level optimizations have been proposed to speed up distributed data-parallel synchronous SGD (syncSGD). Techniques like ring-reduce [172] and tree-reduce [144] have been implemented in several high performance communication libraries (e.g. NCCL and Gloo) which in turn are tightly integrated into popular deep learning libraries like PyTorch [130, 105] and Tensorflow [4]. Both ring-reduce and tree-reduce, are bandwidth efficient and have a constant, and logarithmic dependence on the number of nodes, respectively, i.e. the total number of bytes communicated remains sublinear in the number of machines used for training. During the process of evaluating Accordion [9], we observed that *no prior work studies the utility of gradient compression* in distributed training and compares it to optimized system level implementations. This led us to perform a detailed study to understand the utility of gradient compression in distributed training systems. We first compared state-of-the-art compression systems against optimized system implementations. In the study we show, existing gradient compression methods do not provide significant speedups over. Further, with the aid of a performance model we show how gradient compression methods can be improved and under what setups existing gradient compression methods can be utilized. This study leads to the second contribution in this thesis [10], where we perform a detailed study on existing gradient compression methods and compare them with system based optimization techniques.

Gradient Compression is only applicable in case of data-parallel training, i.e. models fit in one GPU. To further study bottlenecks due to data movement we looked at models which can not fit in single GPU. This led us to study some of the largest models at scale. Recommendation models are distributed as a mix of model and data-parallel approaches, where weights associated with embeddings are partitioned using model parallelism and the weights associated with neural networks are partitioned in data-parallel mode. Recommendation model training is heavily bottle necked by remote embedding accesses. To reduce embedding access overhead we introduce Bagpipe [11]. The main insight in Bagpipe is that in offline large batch training, one can look beyond the current training batch and decide to keep certain embeddings on the trainer ma- chine (caching) and fetch other embeddings out of order (prefetch). We observe that Bagpipe can provide speedups of more than 3.7× over highly optimized existing systems like TorchRec [118].

Next, we shift our focus Large Language Models (LLMs) and particularly to Multi-head attention [183]. Multi-head attention is the core component of LLMs, and is responsible for around 50-60% of time spent during inference [114, 54]. One of the primary reason for poor GPU utilization is the memory bandwidth requirements of multi-head attention [183]. This highlights that for certain operators the data movement bottleneck can be within a single GPU even in distributed setting. To overcome the memory bottleneck we propose an approximation based approach, where we show that several attention heads in multi-head attention are performing are providing similar attention score. In [12] we show that these redundant heads can be efficiently identified at run-time and can be used to minimize the memory bandwidth and compute requirements of attention based architectures.

1.1 List of Papers

This dissertation is primarily composed of following papers -

1. *Accordion: Adaptive gradient compression via critical regime identification* [9], MLSys'21
2. *On the utility of Gradient Compression* [10], MLSys'22
3. *Bagpipe: Accelerating Deep Recommendation model training* [11], SOSP'23
4. *CHAI: Clustered Head Attention for efficient LLM inference* [12], ICML'24

Apart from the topics and papers presented in this dissertation, the author has also worked on Deep Learning Cluster scheduling [13], adversarial attacks in federated learning [188], and zero overhead gradient compression [191].

2 BOTTLENECKS IN MACHINE LEARNING SYSTEMS

Next we provide background on the types of data movement typically observed in machine learning systems and discuss corresponding approaches in reducing those bottlenecks. In Machine learning systems we primarily observe three different types of data movement bottlenecks - (i) Input Data movement related bottlenecks, (ii) Model Parameter related bottlenecks and (iii) Intermediate Activations related bottlenecks. Depending on training setup like model, distribution strategy and infrastructure setup one or several of these bottlenecks appear. In the following subsection we discuss each of these bottlenecks and provide brief overview of different strategies used to alleviate these bottlenecks.

2.1 Input Data bottlenecks

The input data pipeline forms a crucial component of each machine learning training jobs. Input data-pipelines are responsible for reading data often from remote blob storage, applying stochastic transformations and then moving this data to the accelerator memory for training. As the dataset sizes have been increasing where datasets with tens of trillions of data-points are becoming a norm and thousand of accelerators are concurrently being used, the input data throughput requirements has grown at immense rate [219, 123, 220, 221]. To avoid the input training bottlenecks large jobs have been increasing the amount of CPU resources to speed up data-processing, in some cases using upto 5000 workers for a single training workers [127]. There have been several prior works which have looked at different approaches to alleviate input data movement bottlenecks-

Enhanced Parallelism [220, 221, 181, 127] provide an interface for parallelization and large scale distribution of data pre-processing. The primary idea is to enable users to efficiently launch several hundreds of workers and perform distributed data pre-processing. However, these methods do not reduce the amount of actual compute required, or provide efficient implementation of underlying operators.

Caching A common approach to minimize input data bottlenecks is to cache intermediate and reusable data on the training nodes. Prior works [181, 28, 56, 77, 90, 97, 222] have successfully applied caching to alleviate input data bottlenecks.

Operator Fusion and Inter Job Coordination Another common approach to minimize the overhead of data movement is by fusing several operators into a single operation [127]. Another approach several prior works [56] have taken is merging the data processing pipelines for several different jobs by identifying the shared components. This approach allows reuse of processed data.

2.2 Model Parameter bottlenecks

As the model sizes have increased and newer embedding based workloads are becoming common, model parameters have also become major bottlenecks. Model parameter access bottlenecks can be classified into two parts - (i) Offloading based bottlenecks (ii) Remote access bottlenecks.

Offloading based bottlenecks. As the model size has been increasing, several works have proposed offloading parts of neural network to main memory. The core idea is that neural networks are inherently layer based thus only a small part of total parameter count is mandatory to perform compute. Several prior works have looked at offload based mechanisms for enabling training of models which are larger than GPU memory available.

The challenge in offloading is to minimize the overhead on the critical path. Several prior works [71, 153, 139, 92] propose policies to alleviate this bottleneck. Several specialized systems like Marius [124] for learning graph embeddings, Marius++ [186] have been proposed, which offload a portion of the graph and data to disk to overcome the limited memory available on the accelerators.

Remote Access bottlenecks. Another case of bottlenecks with model parameters is the remote access bottleneck. In cases where a model does not fit even in the main memory, it needs to be stored in a remote machine. This has become increasingly common in recommendation models where embedding tables are of terabyte sizes [129]. In this cases the embedding tables need to fetched from remote machines, leading to remote access bottlenecks. There have been several ways which have been introduced to minimize the embedding access overhead, systems like TorchRec [118],Zeus [126] try to hide embedding access latency by overlapping it with different components.

2.3 Intermediate Activation bottlenecks.

Intermediate activation are a big source of data bottlenecks machine learning systems. These bottlenecks primarily manifest in two forms - (i) Memory access bottlenecks and (ii) Activation synchronization bottlenecks.

Memory access bottlenecks. Memory access bottlenecks, several operators in machine learning have a very low compute densite, e.g., multi-head attention, fully connected layers, non-linearity operations. These operators read a large amount of data but compute performed is quite small, which leads to poor utilization of memory. Several high performance computing based libraries [39, 38] and deep learning compilers [26, 223, 175] target this space, by performing operator fusion, tiling and intermediate computation.

Activation synchronization bottlenecks. When performing distributed training, intermediate activation need to synchronized. There has been a plethora of work in reducing these synchronization bottlenecks. Inspired by the fact that SGD can make good progress even with approximate gradients,various Gradient Compression methods have been proposed in recent literature. They can be broadly grouped into quantization, sparsification and low rank aproximations. For quantization, [150, 20] replace each weight with just the sign values. While [15, 110, 154, 155] use the largest few co-ordinates to create a sparse gradient. Wangni et al. [197] randomly drop coordinates in the gradient update to create sparse gradient updates. For quantization [16, 199] quantize each gradient coordinate. In [187, 184] authors show that extremely low rank updates can achieve good compression without loss in accuracy. Yu et al. [212] utilize correlation between gradients for linear compression. Several works have looked at improving communication efficiency by use of Gossip based protocols [108, 169, 96, 95]. Other methods have looked into improving efficiency of distributed training by enabling use of large batch sizes [211, 210, 158, 44] or lower precision [121] without accuracy loss. Other works have also looked at different forms of parallelism [84, 83, 72, 157, 128, 138] for speeding up distributed training.

Similarly there have been system advances in minimizing activation synchronization bottlenecks. Gradients for DNNs are calculated layerwise, therefore, gradients of later layers are available before initial layers. Instead of waiting for the availability of all the gradients, popular deep learning frameworks [105, 130, 4] start gradient communication when some of the gradients are available. This leads to overlapping gradient computation with communication, hiding the time spent in communication.

In this dissertation, we primarily target Activation related bottlenecks in Part II and IV and Model parameter bottlenecks in Part III.

Part II

Reducing Data Movement in Distributed Training

3 ADAPTIVE GRADIENT COMPRESSION

One of the most widely adopted approach for distributed training is Synchronous Data Parallel SGD. One iteration of distributed data parallel SGD (DDP) comprises two main phases: gradient computation and gradient aggregation. During the computation phase, the gradient of the model is typically computed using backpropagation. This is followed by an aggregation phase, where gradients are synchronously averaged among all participating nodes [75, 55]. During this second phase, for state-of-the-art neural network models, millions to billions of parameters are communicated among nodes [24], which has been shown to lead to communication bottlenecks [40, 149, 132, 57, 16]. Alleviating these communication bottlenecks has been an active area of research in recent years. One extremely popular approach to reduce the communication bottleneck is to perform lossy gradient compression [150, 16, 199, 20, 5]. All the lossy gradient compression methods require users to specify an additional hyper-parameter that determines the degree of compression or sparsification before training begins. Choosing compression ratios presents a seemingly inherent trade-off between final model accuracy and the per-iteration communication overhead. With Accordion we automate this process of choosing compression ratios.

3.1 Preliminaries

First we formally describe the distributed SGD setting.

Consider the standard synchronous distributed SGD setting with N distributed workers [151]. For simplicity, we assume that each worker stores n data points, giving us a total of $N \times n$ data points, say $\{(x_i, y_i)\}_{i=1}^{Nn}$. The goal is finding parameter w that minimizes $f(w) = \frac{1}{Nn}\sum_{i=1}^{Nn} \ell(w; x_i, y_i)$ where (x_i, y_i) is the i-th example. In particular, we minimize $f(w)$ using distributed SGD that operates as follows: $w_{k+1} = w_k - \gamma_k \frac{1}{N} \sum_{i=1}^{N} \widehat{g}_i(w_k)$ for $k \in \{0, 1, 2, \ldots\}$, where w_0 is the initial model, γ_k is the step size, and $\widehat{g}_i(w)$ is a gradient computed at worker i for a minibatch (of size B, with $B < n$).

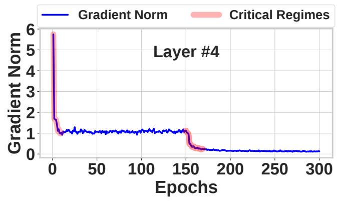
Distributed SGD with adaptive gradient compression Vanilla distributed SGD incurs a huge communication cost per iteration that is proportional to the number of workers N and the size of the gradient. To reduce this communication overhead, we consider a gradient compression strategy, say $C(\cdot, \ell)$, where ℓ is the parameter that determines the compression level used. With such a gradient compression strategy, the update equation becomes $w_{k+1} = w_k - \gamma_k \frac{1}{N} \sum_{i=1}^{N} C(\widehat{g}_i(w_k), \ell_k)$ for $k \in \{0, 1, 2, \ldots\}$, where communicating $C(\widehat{g}_i(w_k), \ell_k)$ requires much fewer bits than communicating the original gradients.

Distributed SGD with adaptive batch size The number of communication rounds in a given epoch also depend on the batch size. For example a batch size $B_{high} > B_{low}$ will communicate $\left\lfloor \frac{B_{high}}{B_{low}} \right\rfloor$ times less than using batch size B_{low} in a given epoch. Although the update equation remains the same $w_{k+1} = w_k - \gamma_k \frac{1}{N} \sum_{i=1}^{N} \widehat{g}_i(w_k)$ for $k \in \{0, 1, 2, \ldots\}$, the number of steps k, taken by a model decreases by $\left\lfloor \frac{B_{high}}{B_{low}} \right\rfloor$ times for a fixed number of epochs.

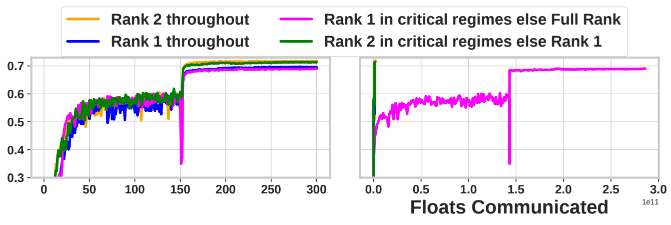
Goals Our goal is to design an algorithm that automatically adapts the compression rate $\{\ell_k\}$ or batch size B_k while training. Although the interplay between batch size and compression ratio is interesting, we don't explore these together, i.e. we don't vary batch size when training with gradient compression. Here, we consider a centralized algorithm, *i.e.*, one of the participating node decides ℓ_{k+1} or B_{k+1} based on all the information available up till step k. This communication rate is then shared with all the

N workers so that they can adapt either their compression ratio or batch size.

3.2 Accordion



(a) Critical Regimes



(b) Accuracy vs Epochs and Floats Communicated

Figure 3.1: **Effect of gradient compression in Critical Regimes when Training ResNet-18 on Cifar-100:** (a) Critical regimes in Cifar-100, ResNet-18 (b, Left) Accuracy vs Epochs. Show the significance of critical regimes in training, using low compression(Rank 2) in critical regimes is enough to get similar accuracy as using low compression throughout . (b, Right) Accuracy vs Floats Communicated, Even when we use uncompressed (Full Rank) gradients everywhere but use high compression (Rank 1) in critical regimes it is not possible to bridge accuracy gap.

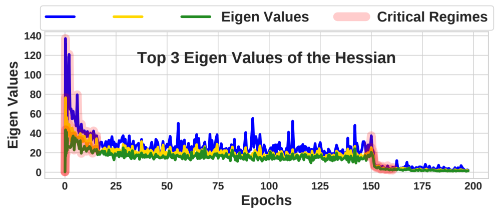
We first explain why adaptive gradient communication can help maintain high generalization performance while minimizing the communication cost. We study this first with gradient compression techniques and then based on these insights we propose Accordion a gradient communication scheduling algorithm. Finally, we show that there is a connection between batch size and gradient compression, and thus Accordion can also be used to enable large batch training without accuracy loss.

3.2.1 Adaptive communication using critical regimes

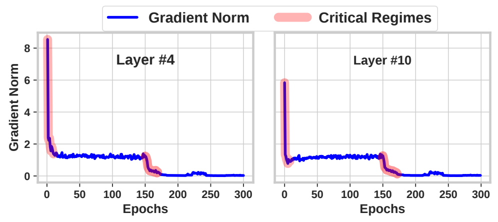
Recent work by [6] has identified *critical regimes* or phases of training that are important for training a high quality model. In particular, [6] show that the early phase of training is critical. They setup an experiment where the first few epochs have corrupted training data and then continue training the DNN with clean training data for the rest of the epochs. Surprisingly, the DNN trained this way showed a significantly impaired generalization performance no matter how long it was trained with the clean data after the critical regime.

We extend these ideas to aid in the design of an adaptive communication schedule and first study this using PowerSGD as the gradient compression scheme. We begin by observing how the gradient norm for each layer behaves while training. When training ResNet-18 on Cifar-100, in Figure 3.1a we see two regions where gradient norm decreases rapidly; during the first 20 epochs and the 10 epochs right after the 150-th epoch, i.e., the point at which learning rate decay occurs. We experimentally verify that these two regions are critical by considering the following compression schedule $\ell = \textsc{low}$ for the first 20 epochs and for 10 epochs after the 150 epoch, and $\ell = \textsc{high}$ elsewhere. Under this scheme the gradients will not be over-compressed in the critical regimes, but at the same time the overall communication will be close to high compression. Figure 3.1b shows the experimental results with ResNet-18 on Cifar-100 for the above scheme. It can be observed that just using low compression (rank 2) in these critical regimes and high compression (rank 1) elsewhere is sufficient to get the same accuracy as using low compression throughout while reducing communication significantly.

Interestingly we also observe in Figure 3.1b that any loss in accuracy by using high compression in critical regimes is not recoverable by using low compression elsewhere. For instance, consider the following compression schedule: $\ell = \textsc{high compression rate}$ for first 20 epochs and for 10 epochs after the 150 epoch, and $\ell = \textsc{no compression}$ elsewhere. Under this schedule, gradients will be over-compressed in the critical regimes, but will be *uncompressed* elsewhere. We see that for ResNet-18 on Cifar-100 even with



(a) Critical Regimes based on Hessian



(b) Critical Regimes based of Gradient Norm

Figure 3.2: **Comparison of Critical Regimes found using Analysis of eigenvalues of Hessian vs Using the Norm of the Gradient:** The experiment is performed on ResNet-18, for Cifar-10. We show that Critical Regimes detected by rapid decay in top eigenvalues of Hessian can also be detected using decay in gradient norm.

significantly higher communication one can not overcome the damage done to training by over compressing in critical regimes. We hypothesize that in critical regimes, SGD is navigating to the steeper parts of the loss surface and if we use over-compressed gradients in these regimes, then the training algorithm might take a different trajectory than what SGD would have taken originally. This might cause training to reach a sub-optimal minima leading to degradation in final test accuracy.

Detecting Critical Regimes: Prior work for detecting critical regimes [80] used the change in eigenvalues of the Hessian as an indicator. We next compare the critical regimes identified by the gradient norm approach described above with the approach used in [80]. In Figure 3.2, we show that these two approaches yield similar results for ResNet-18 on Cifar-10, with the latter having an advantage of being orders of magnitude faster to compute.

Thus, we can see that finding an effective communication schedule is akin to finding critical regimes in neural network training and these *critical regimes* can be identified by measuring the change in gradient norm.

3.2.2 Accordion's Design

We now provide a description of Accordion, our proposed algorithm that automatically switches between lower and higher communication levels by detecting critical regimes. Accordion's first goal is to identify critical regimes efficiently. Our experiments, as discussed previously (Figure 3.2), reveal that critical regimes can be identified by detecting the rate of change in gradient norms without using the computationally expensive technique of [89, 80, 81], where eigenvalues of the Hessian are used to detect critical regimes. This leads us to propose the following simple way to detect critical regimes:

$$\frac{\big|\,\|\Delta_{\text{old}}\| - \|\Delta_{\text{curr}}\|\,\big|}{\|\Delta_{\text{old}}\|} \geqslant \eta,$$

where Δ_{curr} and Δ_{prev}, denotes the accumulated gradient in the current epoch and some previous epoch respectively, and η is the threshold used to declare critical regimes. We set $\eta = 0.5$ in all of our experiments.

We depict Accordion for gradient compression in Algorithm 1. For simplicity and usability, Accordion only switches between two levels of compression levels: ℓ_{low} and ℓ_{high}. Once Accordion detects critical regimes, it sets the compression level as ℓ_{low} to avoid an undesirable drop in accuracy. Based on our observation, critical regimes also almost always occur after learning rate decay, therefore we let Accordion declare critical regime after every learning rate decay. If Accordion detects that the critical phase ends, it changes the compression level to ℓ_{high} to save communication cost. For batch size we use the same algorithm, except instead of switching between ℓ_{low} and ℓ_{high} we switch between B_{low} and B_{high}.

We remark that Accordion operates at the granularity of the gradient compressor being used. For instance, PowerSGD approximates the gradients of each layer independently, so Accordion will also operate at each layer independently and provide a suitable compression ratio for each layer in an adaptive manner during training. While batch size scheduling

Algorithm 1: Accordion for Gradient Compression

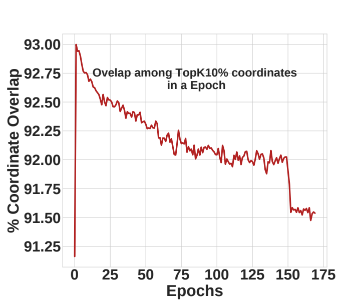
```
HyperParameters: compression levels $\{\ell_{\text{low}}, \ell_{\text{high}}\}$ and detection threshold $\eta$
Input: accumulated gradients in the current epoch ($\Delta_{\text{curr}}$) and in the previous epoch ($\Delta_{\text{prev}}$)
Input: learning rate of the current epoch ($\gamma_{\text{curr}}$) and of the next epoch ($\gamma_{\text{next}}$)
Output: compression ratio to use $\ell$
if $|\|\Delta_{\text{prev}}\| - \|\Delta_{\text{curr}}\||/\|\Delta_{\text{prev}}\| \geqslant \eta$ or $\gamma_{\mathtt{next}} < \gamma_{\mathtt{curr}}$ then
  return $\ell_{\text{low}}$
else
  return $\ell_{\text{high}}$
end if
```

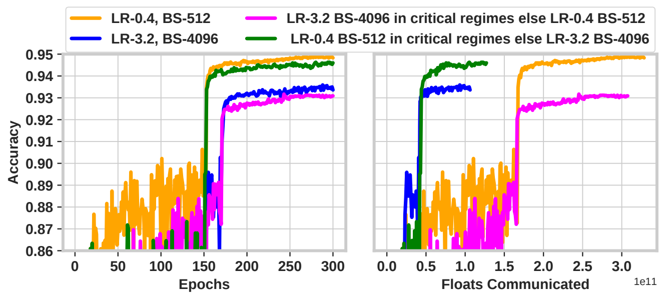
operates at the whole model so Accordion looks at the gradient of whole model and chooses a suitable batch size.

Computational and memory overhead: Accordion accumulates gradients of each layer during the backward pass. After each epoch, norms are calculated, creating $\|\nabla_{\mathtt{curr}}\|$. Once the compression ratio is chosen $\|\nabla_{\mathtt{curr}}\|$ becomes $\|\nabla_{\mathtt{old}}\|$. Thus requiring only size of the model(47 MB in ResNet-18) and a few float values worth of storage. Also Accordion only uses the ratio between previous and current gradient norms to detect critical regimes. This allows Accordion to be easily integrated in a training job where gradients are already calculated, thus making the computational overhead negligible.

3.2.3 Relationship between gradient compression and adaptive batch-size

We first evaluate the effect of batch size on neural network training through the lens of *critical regimes*, which suggests using small batch sizes in critical regimes and large batch size outside critical regimes should not hurt test





(a) Overlap in coordinates (b) Effect of Different Batch sizes in critical regimes

Figure 3.3: **Effect of batch size (ResNet-18 on Cifar-10):** (a) We show that there is significant overlap among the Top10% coordinates. (b, left) Shows that using small batches only in critical regimes is enough to get performance similar to using small batches everywhere. We scale learning rate linearly with batch size as in [55], at steps 150 and 250 we decay the learning rate by 10 and 100 respectively. (b, right) accuracy vs communication.

accuracy. We empirically show in Figure 3.3b that this is indeed true.

Next, the connection between compression and batch size tuning can be made more formal under the following assumption: "each stochastic gradient is the sum of a sparse mean and a dense noise", i.e.,

$$\begin{aligned}\nabla_w \ell(w; x_i, y_i) &= \underbrace{\mathbb{E}_j \nabla_w \ell(w; x_j, y_j)}_{\text{sparse, large magnitudes}} \\ &\quad + \underbrace{(\nabla_w \ell(w; x_i, y_i) - \mathbb{E}_j \nabla_w \ell(w; x_j, y_j))}_{\text{dense, small magnitudes}}\end{aligned} \tag{3.1}$$

Under this assumption, we can see that "large batch gradient $\approx$ highly compressed gradient", as a large batch gradient will be close to $\mathbb{E}_j \nabla_w \ell(w; x_j, y_j)$ by the law of large numbers, a highly compressed gradient will also pick up the same sparse components. Similarly, a small batch gradient is equivalent to weakly compressed gradient. We will like to point out that this assumption is not general and is not applicable on all data or models. It will only hold for models trained with sparsity inducing norms.

We also conduct a simple experiment to support our intuition. We collect all stochastic gradients in an epoch and compute the overlap in coordinates of *Top10%* entries to find how much their supports overlap. Figure 3.3a shows that $> 90\%$ of the top-K entries are common between a pair of stochastic gradients, thereby justifying the above gradient modeling.

Thus, our findings along with prior work in literature can be summarized as high gradient compression, noisy training data, or large batch size in the critical regimes of training hurts generalization. This connection also suggests that Accordion can also be used to schedule batch size.

3.3 Evaluation

We experimentally verify the performance of Accordion when paired with two SOTA gradient compressors, *i.e.*, (i) PowerSGD [184], which performs low-rank gradient factorization via a computationally efficient approach, and (ii) TopK sparsification [15], which sparsifies the gradients by choosing the K entries with largest absolute values. Further we also use Accordion to schedule batch size switching between batch size 512 and 4096 for Cifar-100and Cifar-10.

Evaluation Setup We implement Accordion in PyTorch [130]. All experiments were conducted on a cluster that consists of 4 `p3.2xlarge` instances on Amazon EC2. Our implementation used NCCL an optimized communication library for use with NVIDIA GPUs. For PowerSGD and Batch Size experiments we used the all-reduce collective in NCCL and for TopK we used the all-gather collective. We fix η to be 0.5 and run Accordion every 10 epochs i.e. Accordion detects critical regimes by calculating rate of change between gradients accumulated in current epoch and the gradients accumulated 10 epochs back. We empirically observe that these choices of hyper-parameters lead to good results and have not tuned them. One of our primary goal was to design Accordion such that it should not require signifcant amount of hyper-parameter tuning. Therefore for all of our experiments we didn't perform any hyper-parameter tuning and used the same hyper-parameters as suggested by authors of previous compression methods, e.g. For PowerSGD we used the same setting as suggested by Vogels et al. [184]. For large batch size experiments we use the same hyper-parameters as used for regular training. For all our experiments on batch size we performed LR Warmup of 5 epochs as suggested by Goyal et al. [55], i.e. for batch size 512 we linearly increase the learning rate from 0.1 to 0.4 in five epochs where 0.1 is learning rate for batch size 128. Due to relationship shown between batch Size and learning rate by Smith et al. [158], Devarakonda et al. [44] when Accordion shifts to large batch it also correspondingly increases the learning in the same ratio, i.e. when switching between Batch Size 512 to Batch size 4096, Accordion also scales the learning rate by $8\times$.

For image classification tasks we evaluated Accordion on Cifar-10 and Cifar-100. Cifar-10 consists of 50,000 train images and 10,000 test images for 10 classes. Cifar-100 has similar number of samples but for 100 classes. For language modeling we used WikiText which has around 2 million train tokens and around 245k testing tokens. To show the wide applicability of Accordion we consider a number of model architectures. For CNNs, we study networks both with and without skip connections. VGG-19 and GoogleNet are two networks without skip connections. While ResNet-18, Densenet, and Squeeze-and -Excitation are networks with skip connections. For language tasks we used a two layer LSTM.

Accordion on PowerSGD PowerSGD [184] shows that using extremely low rank updates (Rank-2 or Rank-4) with error-feedback [161] can lead to the the same accuracy as syncSGD. In Table 3.1 and 3.2 we show that

Table 3.1: Accordion with PowerSGD on Cifar-10

Network	Rank	Accuracy	Data Sent (Million Floats)		Time (Seconds)	
Resnet-18	Rank 2	**94.5%**	2418.4	(1×)	3509	(1×)
	Rank 1	94.1%	1350.4	(1.7×)	3386	(1.03×)
	Accordion	**94.5%**	1571.8	(**1.5**×)	3398	(**1.03**×)
VGG-19bn	Rank 4	**93.4%**	6752.0	(1×)	3613	(1×)
	Rank 1	68.6%	2074.9	(3.25×)	3158	(1.14×)
	Accordion	**92.9%**	2945.1	(**2.3**×)	3220	(**1.12**×)
Senet	Rank 4	**94.5%**	4361.3	(1×)	4689	(1×)
	Rank 1	94.2%	1392.6	(3.1×)	4134	(1.13×)
	Accordion	**94.5%**	2264.4	(**1.9**×)	4298	(**1.09**×)

Table 3.2: Accordion with PowerSGD on Cifar-100

Network	Rank	Accuracy	Data Sent (Million Floats)		Time (Seconds)	
Resnet-18	Rank 2	**71.7%**	2426.3	(1×)	3521	(1×)
	Rank 1	70.0%	1355.7	(1.8×)	3388	(1.04×)
	Accordion	**71.8%**	1566.3	(**1.6**×)	3419	(**1.03**×)
DenseNet	Rank 2	**72.0%**	3387.4	(1×)	13613	(1×)
	Rank 1	71.6%	2155.6	(1.6×)	12977	(1.04×)
	Accordion	**72.5%**	2284.9	(**1.5**×)	13173	(**1.03**×)
Senet	Rank 2	**72.5%**	2878.1	(1×)	5217	(1×)
	Rank 1	71.5%	1683.1	(1.7×)	4994	(1.04×)
	Accordion	**72.4%**	2175.6	(**1.3**×)	5074	(**1.03**×)

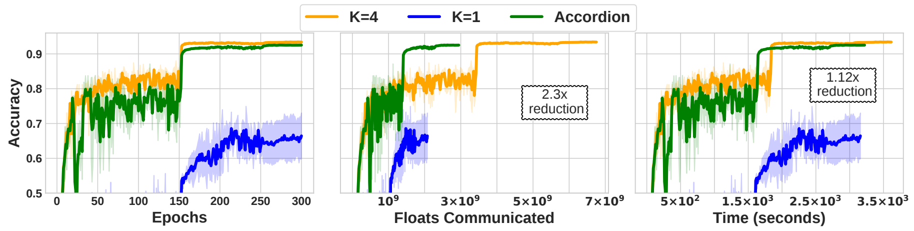
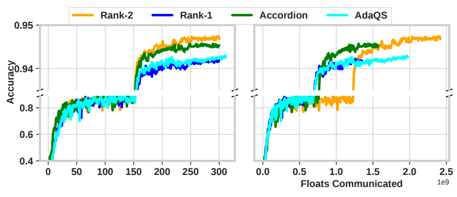


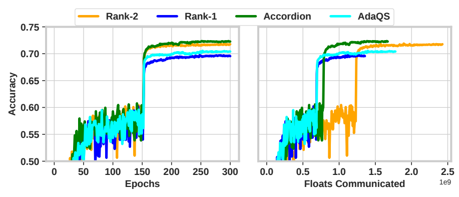
Figure 3.4: **Accordion using PowerSGD with ℓ_{low} = rank 4 and ℓ_{high} = rank 1 on VGG-19bn:** We show Accordion being able to bridge more that 25% of accuracy difference with 2.3× less communication .

Accordion by performing adaptive switching between Rank-1 and Rank-2,4 reaches similar accuracy but with significantly less communication. For e.g. in Table 3.2 with ResNet-18 on Cifar-100 using ℓ_{low} = Rank 2 leads to accuracy of 72.4% while ℓ_{high} = Rank 1 achieves 71.3%. Accordion switching between Rank 2 and Rank 1 achieves an accuracy of 72.3%. Figure 3.4 shows the result for VGG-19bn trained with Cifar-10, in this case Accordion almost bridges accuracy gap of 25% while saving almost 2.3× in communication. Results on more compression schemes can be found in Section 5 of our original paper [9].

Comparison with Prior Work We compare Accordion with prior work in adaptive gradient compression and adaptive batch size tuning. For



(a) ResNet-18 trained on Cifar-10



(b) ResNet-18 trained of Cifar-100

Figure 3.5: **Comparison with AdaQS:** We compare Accordion against AdaQS [59] on Cifar-10 and Cifar-100. We use PowerSGDas the Gradient Compressor. Even though AdaQS communicates more that Accordion it still looses accuracy compared to low compression. Accordion on other hand with less communication is able reach the accuracy of low compression.

adaptive gradient compression we consider recent work by [59] that uses the mean to standard deviation ratio (MSDR) of the gradients. If they observe that MSDR has reduced by a certain amount(a hyper-parameter), they correspondingly reduce the compression ratio by half (i.e., switch to a more accurate gradient). We use this approach with PowerSGD and our experiments in Figure 3.5 suggest that their switching scheme ends up requiring more communication and also leads to some loss in accuracy.

Accordion on extremely large batch sizes To push the limits of Batch Size scaling further we tried using Accordion for scaling Cifar-10 on ResNet-18 to batch size of 16,384.We observed that using Accordion looses around (1.6%) accuracy compared to using batch size 512. Interestingly we also observe that when Accordion first switches the batch size there is a rapid drop, but then training immediately recovers.

3.4 Conclusion

We propose Accordion, an adaptive gradient compression method that can automatically switch between low and high compression. Accordion works by choosing low compression in critical regimes of training and high compression elsewhere. We show that such regimes can be efficiently

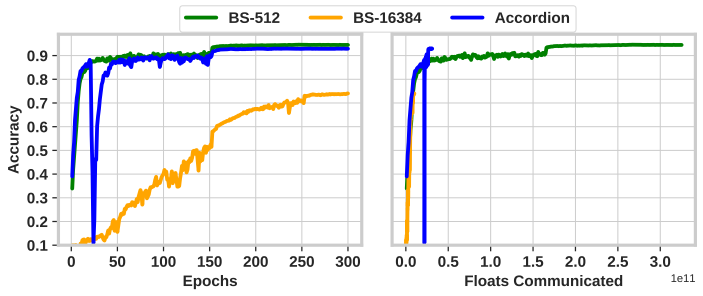


Figure 3.6: **Using Extremely Large Batch Size:** We observe that ACCORDION looses around 1.6% accuracy when we use batch size of 16,384. Showing ACCORDION can often prevent large accuracy losses while providing massive gains.

identified using the rate of change of the gradient norm and that our method matches critical regimes identified by prior work. We also discuss connections between the compression ratio and batch size used for training and show that the insights used in ACCORDION are supported by prior work in adaptive batch size tuning. Finally, we show that ACCORDION is effective in practice and can save upto 3.7× communication compared to using low compression without affecting generalization performance. Overall, our work provides a new principled approach for building adaptive-hyperparameter tuning algorithms, and we believe that further understanding of critical regimes in neural network training can help us design better hyperparameter tuning algorithms in the future.

4 UTILITY OF GRADIENT COMPRESSION

A rich body of prior work has highlighted the existence of communication bottlenecks in synchronous data-parallel training. To alleviate these bottlenecks, a long line of recent research proposes gradient and model compression methods. In this work, we evaluate the efficacy of gradient compression methods and compare their scalability with optimized implementations of synchronous data-parallel SGD across more than 200 realistic distributed setups. Surprisingly, we observe that only in 6 cases out of more than 200, gradient compression methods provide speedup over optimized synchronous data-parallel training in the typical data-center setting. We conduct an extensive investigation to identify the root causes of this phenomenon, and offer a performance model that can be used to identify the benefits of gradient compression for a variety of system setups. Based on our analysis, we propose a list of desirable properties that gradient compression methods should satisfy, in order for them to provide meaningful utility.

4.1 Preliminaries

We first provide a brief background of several different threads of prior work that aim at enabling faster distributed machine learning. Several lossy gradient compression methods based on quantization [16, 20, 88, 42, 149, 199, 21, 213, 106, 69, 170, 45, 163, 53, 224, 215, 201, 168], sparsification [162, 111, 15, 17, 111, 154, 155, 50, 115, 156, 197, 168, 147, 148], low rank decomposition [187, 184, 189], and other approaches [5, 165, 79] have been proposed in literature. Recent surveys [204, 171] describe these methods in detail.

In this work, we benchmark several popular gradient compression schemes (Table 4.1), and we then pick three gradient compression schemes

Table 4.1: Encode-Decode of gradient compression methods for ResNet-50 on V100 GPUs.

Type	Method	$T_{encode\_decode}(ms)$	All-Reduce
Sparsification	MS-TopK - 1%	103	✗
	DGC - 1%	221	✗
	TopK - 1%	273	✗
	RandomK - 1%	163	✓
Quantization	SignSGD	16	✗
	QSGD-2bit	39	✗
	TernGrad	94	✗
Low Rank	PowerSGD-Rank 4	45	✓
	ATOMO-Rank 4	1586	✗

which have the least compression overheads and high compression ratios for detailed analysis. We chose, quantization based signSGD [20, 21], low-rank decomposition based PowerSGD [184] and sparsification based MSTop-K [156]. We compare and evaluate these schemes to see if they provide any benefit over off-the-shelf implementation of syncSGD, i.e. PyTorch DDP [105].

4.1.1 System Advances

Next, we provide a brief overview of several system advances which have been applied to syncSGD to improve the performance of distributed training.

All-reduce. In recent years, systems have shifted from using a parameter server based topology to an all-reduce topology for gradient synchronization. For example, we observe that all submissions to DawnBench [34] use all-reduce for performing distributed training. Communication costs can be typically modeled using a cost model [146] where cost of sending/re-

Table 4.2: **Comparing aggregation schemes:** We show how latency and bandwidth term scale for different aggregation strategies. α is the latency, β is the inverse of bandwidth, and n is the size of vector communicated. p is the number of machines

Algorithm	Latency	Bandwidth
Ring Reduce	$2(p-1)\alpha$	$2\beta\frac{(p-1)}{p}n$
Tree Reduce	$2\alpha \log p$	$2\beta(\log p)n$
Parameter Server	2α	$2\beta(p-1)n$

ceiving a vector of size n is computed as the sum of latency and bandwidth requirements. There are several optimizations [134, 172, 66, 144] for all-reduce based collectives like ring-reduce [19], tree-reduce [144], recursive doubling [180], 2D-Torus [122, 86], and etc. These optimizations explore the trade-off between the latency and bandwidth terms. We list latency and bandwidth terms for a few aggregation strategies in Table 4.2 for synchronizing a vector of size n among p machines. In Table 4.2, α represents the latency term (typically between 0.5 to 1ms in public clouds) and β represents bandwidth term. We would like to point out that the bandwidth requirement for ring reduce stays almost constant even with increase in number of machines p. High performance implementations like NVIDIA-NCCL [1] dynamically chooses between tree and ring reduce based on several factors like number of machines, bandwidth, interconnect, communication size to list a few. In this work for simplicity, we analyze our results with the communication model of ring-reduce.

Communication and Computation Overlap. Gradients for DNNs are calculated layerwise, therefore, gradients of later layers are available before initial layers. Instead of waiting for the availability of all the gradients, popular deep learning frameworks [105, 130, 4] start gradient communication when some of the gradients are available. This leads to overlapping

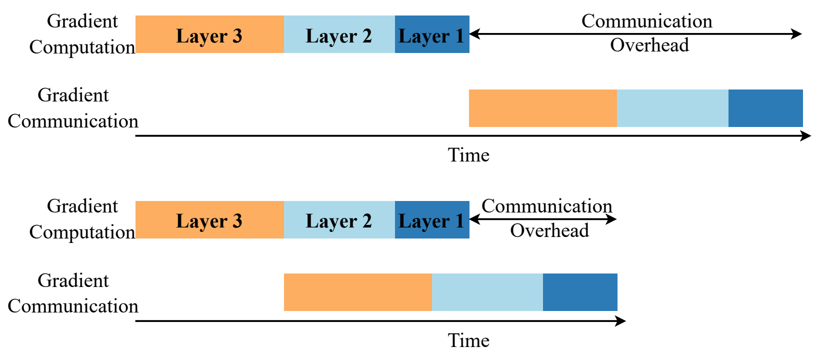


Figure 4.1: **Illustration of how overlapping can reduce the total iteration time.** (Above) Gradient computation and communication done serially. (Below) Gradient computation and communication being overlapped, i.e. when the gradient of a layer is computed, it is communicated right after the gradient of the previous layer.

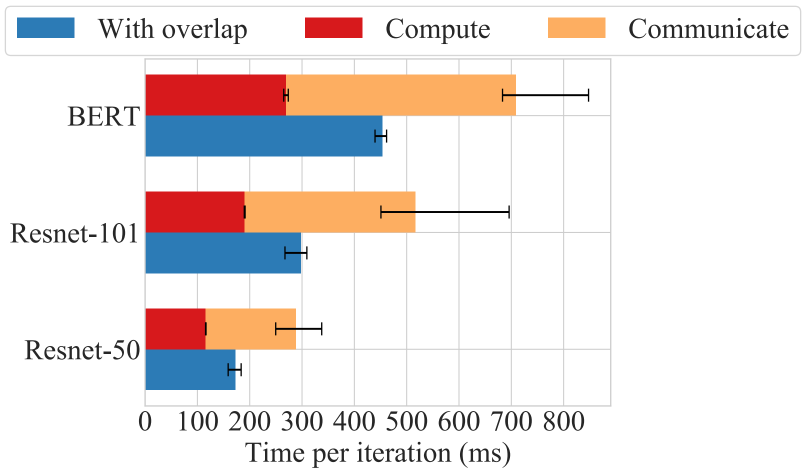


Figure 4.2: **Effect of Overlap:** We plot the iteration time for computation and gradient synchronization for 64 GPUs, both with and without overlap. In case of Resnet-50 we observe that overlapping reduces iteration time by upto 46%.

gradient computation with communication, hiding the time spent in communication. Figure 4.1 illustrates how overlap can provide speedups. In Figure 4.2, we observe that overlapping can provide speedups of almost 46% for ResNet-50.

Bucketing Gradients. Calling the all-reduce collective per layer can often lead to large overheads. To amortize the overhead of calling all-reduce, optimized implementation of syncSGD [105, 151] create fixed size buckets. Once the gradients for a bucket are calculated then *a*ll-reduce is called on the entire bucket. Bucket sizes are typically large (25 MB by default in PyTorch).

In this work, we benchmark the runtime of the systems with the aforementioned optimizations to compare against gradient compression methods on real-world computer vision and natural language processing tasks.

4.1.2 Evaluating utility of gradient compression

In this section, we perform a detailed experimental evaluation comparing the scalability of gradient compression methods with an optimized syncSGD implementation. We start by analyzing the effects of overlapping gradient compression with gradient computation. Next we run large scale experiments to study how gradient compression methods scale across a range of models.

Methodology. We begin by comparing the overhead of compression methods which have been reported to scale well. Upon comparing nine different gradient compression methods using ResNet-50 on 64 V100 GPUs. We observe that most gradient compression methods take around 100ms for compressing and decompressing gradients of ResNet-50 on 64 GPUs. However, there are some methods which are considerably faster, e.g. signSGD takes only 16ms for encoding-decoding. Among low-rank methods we find that PowerSGD is around 45× faster than ATOMO (another low rank method) [187]. Based on this comparison, we choose the most scalable method in each category. Among quantization based methods we choose signSGD [20, 21] which achieves 32× compression ratio by only communicating the sign of the gradient. Among sparsification based methods we choose MSTop-K [156], a scalable TopK method and among low rank methods we choose PowerSGD, a low overhead method with compression ratios of around 100$\times$. For syncSGD we use PyTorch-DDP module [105].

We would like to point out that we use optimistic compression ratios, e.g. for PowerSGD we use Rank-4, 8, and 16. Such high compression ratios have been shown to work [184] for small datasets like CIFAR-10 and WikiText-2 but can lead to accuracy loss for large datasets [184, 137]. While for MSTop-K we are again being optimistic and consider dropping 99.9% gradients and assuming that it will have no loss in accuracy. We chose these since we wanted to consider a best case scenario for gradient compression methods.

We use ResNet-50 (97MB), ResNet-101 (170MB) and BERT$_{\text{BASE}}$ (418MB) as the models to study given their disparate communication and computation requirements. Similar models were used by prior works [184, 205] in gradient compression to compare the performance of gradient compression schemes and our code can be easily used to benchmark other models as well. For timing measurements on vision models we use the ImageNet dataset [41] and we fine-tune the BERT$_{\text{BASE}}$ model on Sogou News dataset [164]. For the timing measurements, we run 60 iterations for each setup and discard the first 10. We plot the mean of the remaining 50. The error bars in the figure correspond to minimum and maximum values.

Our experiments are conducted over *p3.8xlarge* instances on Amazon EC2. Each instance is equipped with 4 V100 GPUs and provides around 10Gpbs of bandwidth. We scale our experiments up to 96 GPUs (24 *p3.8xlarge* instances) and consider weak scaling, i.e. the number of inputs per worker is kept constant as the number of workers increase. This is a commonly used scenario for evaluating the scalability of deep learning training [34, 128]. Thus, when we refer to a particular batch size, it is the batch size at each worker.

Using Per Iteration Time as A Metric Instead of Accuracy. We consistently use time per iteration as the metric for evaluation. It is well known from prior works that gradient compression methods can lead to some final model accuracy loss [205] when used for training. Our main goal in this work is to study the scalability of distributed training and compare per iteration time of syncSGD against state-of-the-art gradient compression methods. Though important, the final model accuracy that the gradient compression methods achieve is not the main focus of this work. The per-iteration speedup is a more critical question as if there is limited speedup from using gradient compression then there is no incentive to deploy such methods irrespective of the accuracy. Another reason for not performing an accuracy based study is that gradient compression methods often introduce new hyper-parameters while also requiring modifications to existing hyper-parameters like the learning rate schedule. It is often non-trivial to find optimal hyper-parameters which balance compression and accuracy loss and we plan to study this in future work.

4.1.3 Other Related Work

Several works have looked at improving communication efficiency by use of Gossip based protocols [108, 169, 96, 95]. Other methods have looked into improving efficiency of distributed training by enabling use of large batch sizes [211, 210, 158, 44] or lower precision [121] without accuracy loss. Other works have also looked at different forms of parallelism [84, 83, 72, 157, 128, 138] for speeding up distributed training. MLPerf [116] and DawnBench [34] are two well known industry supported efforts to perform periodic benchmarking on training and inference speed at scale. Our findings about scalability of all-reduce based compression scheme has also been reported by prior works [184, 31]. A recent survey [204] quantitatively compares several gradient compression methods. However unlike our work it does not account for systems optimization like overlap of communication and computation. Zhang et al. [217] study whether network is the bottleneck in distributed training. Unlike [217] and other listed works, our study focuses on the utility of gradient compression methods in several different settings and analyzes others aspects beyond network bandwidth like compute availability, batch size, model size, system advances etc. Further, our performance model allows to reason about performance of distributed training and to predict the performance gains without running large scale experiments.

4.2 Evaluating Gradient Compression

4.2.1 Overlapping Compression and Computation

We observe that when gradient compression is performed in parallel with the backward computation it is slower than performing gradient compression after completing backward pass. Figure 4.3 depicts this phenomenon on ResNet-50 using PowerSGD Rank-4, MSTop-K-1%, and signSGD. Since both gradient compression and gradient computation are compute-heavy steps, when performed in parallel they end up competing for compute resources on the GPU leading to an overall slow down. On the other hand, syncSGD only performs *all-reduce* operation which is communication heavy with very little compute, thus efficiently utilizing the communication resources on the GPU without affecting the backward pass. Since we consistently observe that compression schemes perform better when not overlapped, for the next set of experiments we use *non-overlapped versions of compression*.

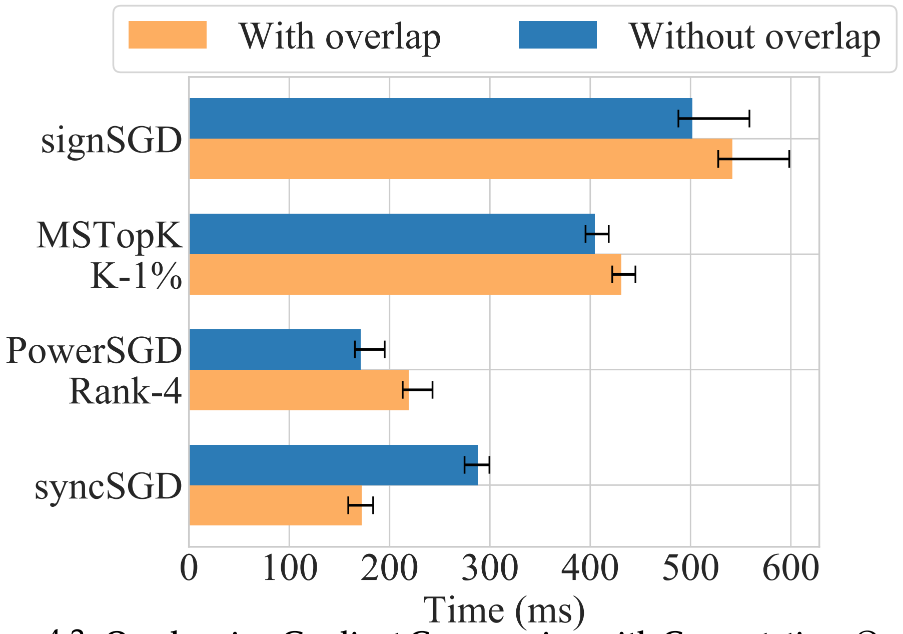
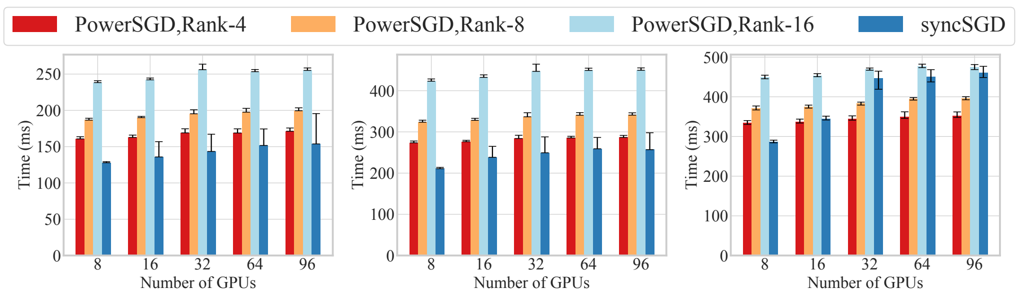


Figure 4.3: **Overlapping Gradient Compression with Computation:** Overlapping compression leads to requiring more time per iteration than performing it sequentially, due to resource contention for compute resources. The results are for 64 GPUs.

4.2.2 Comparing Gradient Compression with Optimized Sync SGD

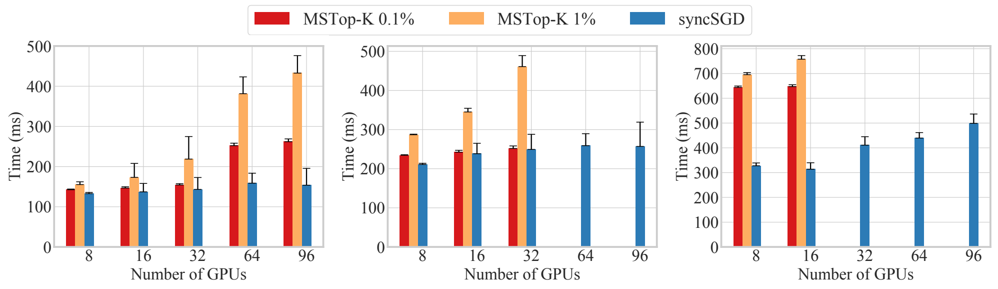
We next analyse the performance of gradient compression methods against syncSGD.

PowerSGD. We first study the scalability of PowerSGD when compared to syncSGD for ResNet-50, ResNet-101 , and BERT$_{\text{BASE}}$. We use Rank-4, 8 and 16 as discussed previously. As shown in Figure 4.4 we can see that PowerSGD with Rank 4, 8, and, 16 is *slower* than syncSGD for ResNet-50 and ResNet-101 with batch size 64. This is primarily because syncSGD does not incur any overheads from compression and is able to overlap communication with computation. On the other hand, for BERT$_{\text{BASE}}$, which is a much larger model (490MB), we see that for 96 GPUs, Rank-4 and



(a) ResNet-50: BSize 64 (b) ResNet-101: BSize 64 (c) BERT$_{\text{BASE}}$: BSize 12

Figure 4.4: **Scalability of PowerSGD:** When compared against an optimized implementation of syncSGD, PowerSGD provides speedups only in case of BERT$_{\text{BASE}}$ when using Rank-4 and Rank-8 above 32 GPUs. In other cases it has a high per iteration time.

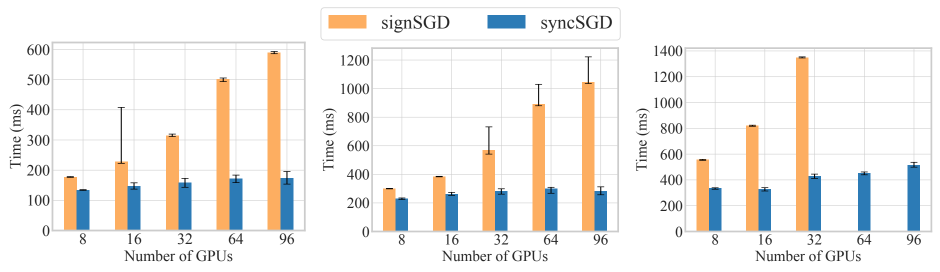


(a) ResNet-50: BSize 64 (b) ResNet-101: BSize 64 (c) BERT$_{\text{BASE}}$: BSize 12

Figure 4.5: **Scalability of MSTop-K:** Comparing MSTop-K against syncSGD we observe due to lack of compatibility with *all-reduce* MSTop-K performs slower than or comparable to syncSGD . For ResNet-101 and BERT we could not scale TopK beyond 16 and 32 GPUs respectively, due to running out of memory as memory requirement increasing linearly with number of machines.

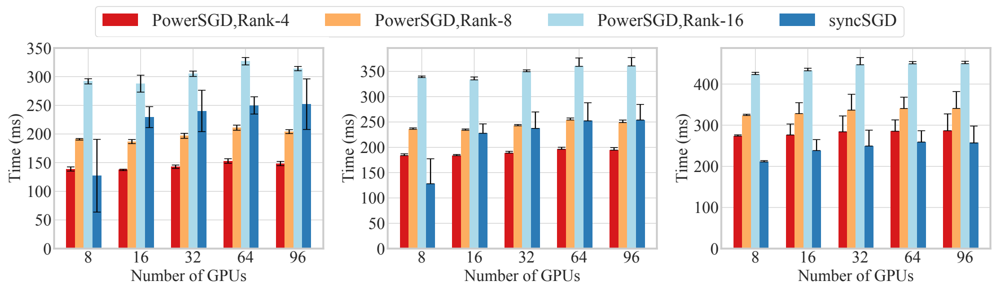
Rank-8 are faster than syncSGD by around 18.8% and 11.3% respectively, while Rank-16 still takes longer than syncSGD.

MSTop-K. Since the MSTop-K [156] operator is incompatible with *all-reduce* we use *all-gather* for communication. As shown in Figure 4.5, only in 2 out of 15 different setups we observe a minuscule speedup (around 1.3%) when compared against syncSGD. These speedups are achieved when using MSTop-K-0.1%, i.e., when 99.9% of the entries in the gradient



(a) ResNet-50: BSize 64 (b) ResNet-101: BSize 64 (c) $\text{BERT}_{\text{BASE}}$: BSize 12

Figure 4.6: **Scalability of SIGNSGD:** Due to lack of support for *all-reduce* and linearly increasing decode time, across all three models, SIGNSGD performs considerably slower than syncSGD. For $\text{BERT}_{\text{BASE}}$ we were not able to scale signSGD beyond 32 GPUs because we ran out of memory on a V100 GPU. This is due to the memory requirement increasing linearly with number of machines.



(a) ResNet-101: BSize 16 (b) ResNet-101: BSize 32 (c) ResNet-101: BSize 64

Figure 4.7: **Effect of varying batch size:** Here we compare POWERSGD against ResNet-101 on different batch sizes. We observe that large batch sizes provide more opportunity to syncSGD to hide the communication time, meanwhile at small batch sizes due to reduced computation time this overlap is not possible. Therefore gradient compression methods become more useful at small batch sizes.

are dropped. Also, due to high memory requirements for creating buffers for the all-gather primitive MSTOP-K does not scale beyond 32 GPUs for ResNet-101 and 16 GPUs for BERT on a V100 GPU.

SIGNSGD. We study SIGNSGD with majority vote, where 1 bit is sent for each float (32 bit) leading to $32\times$ compression. Majority vote operation is not associative thus requiring use of all-gather. Figure 4.6, shows that despite SIGNSGD being extremely quick to encode and decode, due to lack of compatibility with *all reduce*, communication time scales linearly. Further, due to overheads in creating buffers for the all-gather primitive we can not scale SIGNSGD on BERT$_{\text{BASE}}$ beyond 32 GPUs.

4.2.3 Effect of Batch Size on Scalability

For analysing the effect of varying batch sizes, we compare PowerSGD against syncSGD since it is the most scalable method we encounter. In Figure 4.7, for ResNet-101, we find that the benefits of using PowerSGD with Rank-4 drops as the batch size increases. For instance, when using 96 GPUs, PowerSGD Rank-4 provides almost 42.5% speedup when training using batch size 16. This speedup drops to 25.7% for batch size 32 and with batch size 64, we observe that PowerSGD Rank-4 is around 6.3% slower than synSGD. In general, increasing batch size leads to an increase in the compute time which in turn provides more opportunity for syncSGD to overlap computation and communication.

4.2.4 Exploring utility of gradient compression in additional setups

In the previous section we looked at the performance of distributed training and gradient compression of popular models on existing hardware. Next we try to identify regimes, in terms of hardware or model characteristics, where gradient compression can provide significant gains i.e. how will our above results change if we had 100Gbps bandwidth or an 8$\times$ faster GPU. To answer such questions, we develop a performance model that can be used to reason about expected performance under different setups.

4.2.4.1 Performance Model for Distributed Data Parallel.

Based on optimizations listed for syncSGD in [105] we build an analytical performance model. We assume the model can be partitioned into k buckets, where the first $k-1$ buckets are of size b and the last bucket is of size $\hat{b}$, where $\hat{b} \leqslant b$. The time observed for backward pass and gradient synchronization for synSGD becomes:

$$T_{obs} \approx max(\gamma T_{comp}, (k-1) \times T_{comm}(b, p, BW)) + \\ T_{comm}(\hat{b}, p, BW)$$

where T_{obs} is the total time observed for backward pass and synchronization, T_{comp} is the compute time for the backward pass on single machine, $(k-1) \times T_{comm}(b, p, BW)$ is the time required to communicate $k-1$ gradient buckets of size b across p GPUs at BW bandwidth, and $T_{comm}(\hat{b}, p, BW)$ is the time to communicate the last bucket of size $\hat{b}$, which can not be overlapped with computation. Finally, γ represents the factor of slowdown in backward pass due to overlap with communication. We observe γ to between 1.04 to 1.1. In case of syncSGD when using ring-reduce, $T_{comm}(b, p, BW)$ becomes

$$T_{comm}(b, p, BW) = 2\alpha \times (p-1) + 2 \times b \times \frac{(p-1)}{p \times BW} \tag{4.1}$$

where α is the latency coefficient, b is the bucket size, p is the number of GPUs and BW is the bandwidth available.

Verifying Performance Model. We empirically verify our performance model using the same experimental setup as mention in Section 4.1.2. As shown in Figure 4.9 we observe that our model very closely tracks the actual performance in all cases. The median difference between our prediction and actual runtime is 1.8% and the maximum is 13.7%.

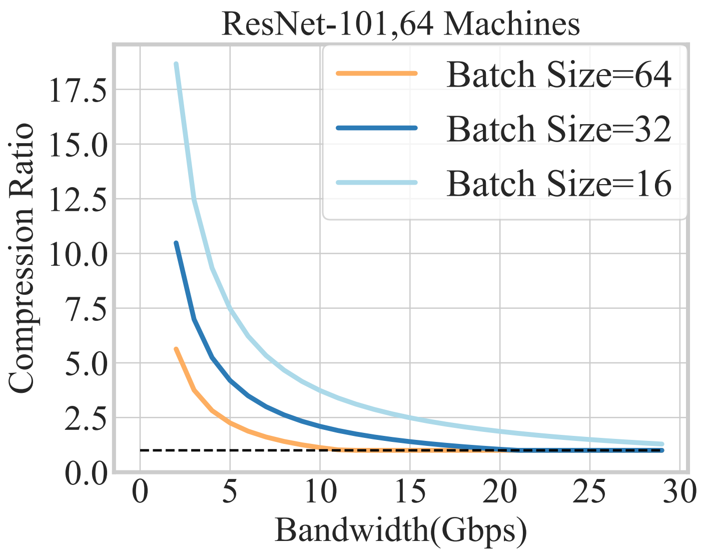


Figure 4.8: **Required gradient compression for near linear speedups (simulated):** Above figure is for ResNet-101 simulated for 64 machines. We observe that the required gradient compression for near linear scaling at 10 Gbps even for quite small batch sizes is around 4×.

4.2.5 Insights from the Performance Model

How Much Should We Compress? Using the performance model we investigate how much compression is required for linear scalability. Figure 4.8 shows that even at small batch-sizes for ResNet-101 we need around 4× compression for linear scalability, which is significantly smaller than what most compression methods offer. Our analysis shows that for linear scaling we do not need extremely high compression ratios.

Effect of Network Bandwidth on Gradient Compression. Figure 4.10 shows comparison between speedups for ResNet-101 when using sync-SGD and PowerSGD Rank-4 at different network bandwidths. In addition to estimating time taken with our performance model, we also use the TC command [2] to limit bandwidth on a real cluster, thereby verifying our performance model (the markers represent measurements on hardware). The figure shows that gradient compression is very useful in low bandwidth settings ($\leqslant 8$ Gbps). Although low bandwidths are uncom-

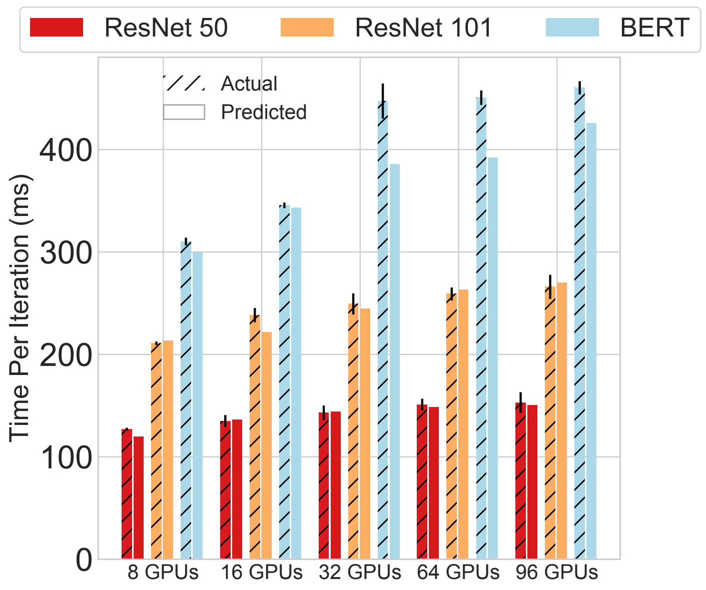


Figure 4.9: **Verifying performance model for syncSGD:** Our performance model matches the actual performance for all three models across wide range of GPUs. The median difference between predictions and actual runtime is 1.8%.

mon in data centers (10 Gbps is minimum with a V100 GPU on Amazon EC2), this shows that in certain cases like wide-area learning [23] gradient compression methods can be extremely useful.

Our performance model also allows us to consider several what-if scenarios. To understand how and where gradient compression methods will be useful, we can vary several factors like compute availability, encode-decode time, network bandwidth etc. Based on our results in Section 4.2.2 which show that PowerSGD Rank-4 is the most scalable compression scheme, we use PowerSGD with Rank-4 as the baseline for these what-if analyses.

Required Compression for linear scaling. Existing gradient compression methods provide massive amount of compression which often leads to poor accuracy. Using our performance model we study the amount of gradient compression required for linear scaling. Figure 4.11 shows

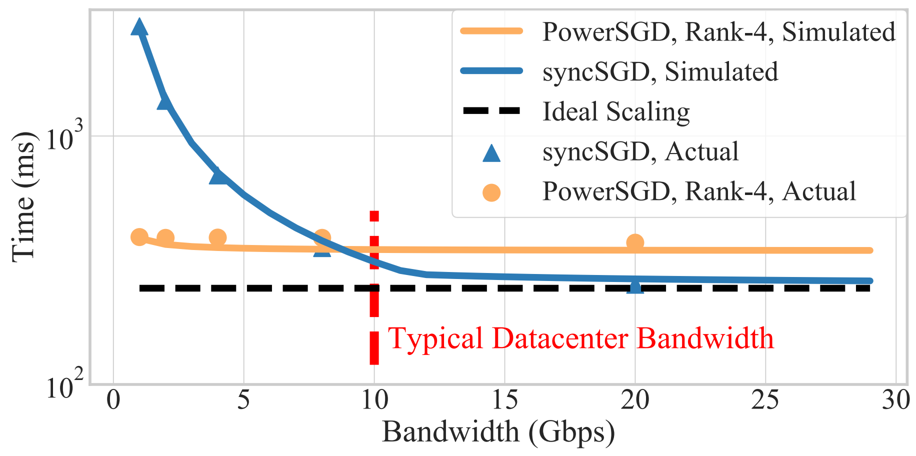
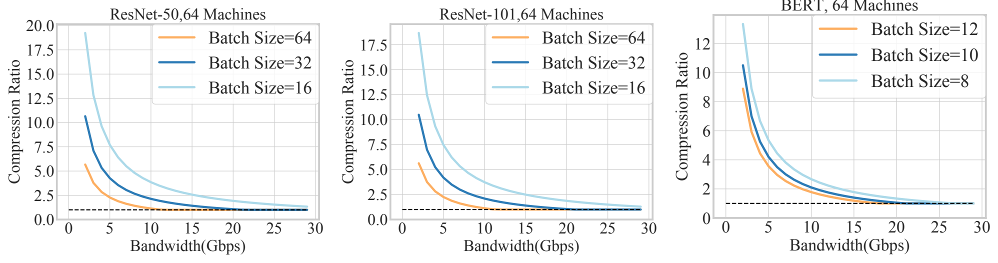


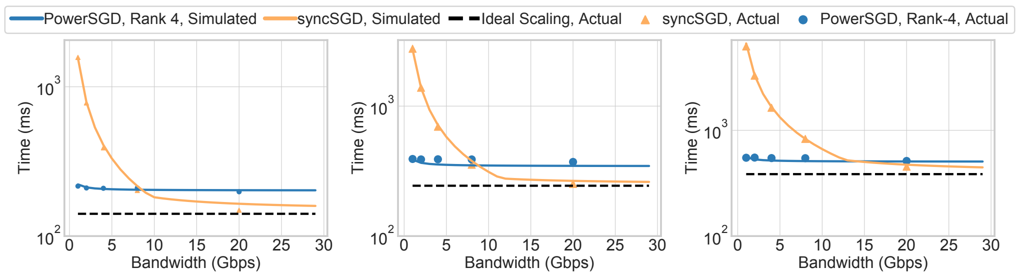
Figure 4.10: **Evaluating effect of network bandwidth (simulated):** Above curve is for Resnet-101, batch size 64 on 64 GPUs. We observe that at bandwidth lower than 8.2 Gbps, PowerSGD Rank-4 can provide speedups but above that syncSGD performs better.



(a) ResNet50: 64 GPUs (b) ResNet101: 64 GPUs (c) BERT: 64 GPUs

Figure 4.11: **Required gradient compression for near optimal speedups (simulated):** We observe that the required gradient compression for near optimal scaling is quite small. At 10 Gbps even for quite small batch sizes we need less than 4× gradient compression, which is quite small compared to what popular gradient compression methods.

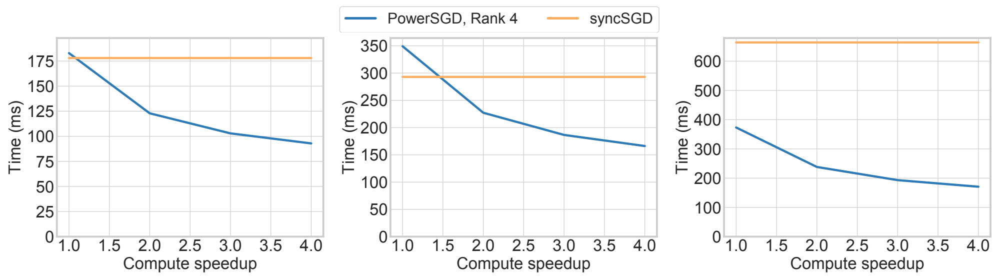
that in most common models at 10 Gbps we do not need compression greater than 4×. This shows that focus of gradient compression should be to reduce the overheads of compression rather than providing very high compression rates.



(a) ResNet50: BSize 64 (b) ResNet101: BSize 64 (c) BERT: BSize 12

Figure 4.12: **Evaluating effect of network bandwidth on training (simulated):** We vary bandwidth availability and analyse the performance of synchronous SGD vs PowerSGD Rank 4. We observe that as bandwidth increase significantly it helps synchronous SGD since it has a larger communication overhead. Moreover we observe the PowerSGD provides massive gains at extremely low bandwidth (1Gbps) but as bandwidth scales we see PowerSGD gets bounded by compute availability. The markers are values from actual experiments, this also shows how close our performance model is to actual measurement.

Effect of Network Bandwidth In Figure 4.12 we vary network bandwidth available from 1Gbps to 30Gbps and see how this changes the speedup offered by PowerSGD. We see that, for example, in the case of Resnet-50, PowerSGD offers considerable speedup at low network bandwidths (1-7 Gbps) but becomes slower than synchronous SGD when bandwidth available becomes $> 9Gbps$. This is due to the fact that syncSGD benefits more from availability of higher bandwidth since it communicates significantly more while PowerSGD is still limited by extra time spent in the encode-decode step. For BERT which is a communication heavy network, PowerSGD becomes slower than syncSGD at around 15Gbps. In Figure 4.12 the markers represent values from actual experiments. To perform these experiments we used the tc command in linux to modify the available bandwidth. For experiments with bandwidth less than 10Gbps we used p3.8xlarge instances which provide a maximum of 10Gbps bandwidth. And for 20 Gbps experiment we used p3.16xlarge instance which provides 25 Gbps bandwidth. The markers are extremely close to the



(a) ResNet50: BSize 64 (b) ResNet101: BSize 64 (c) BERT: BSize 12

Figure 4.13: **Evaluating effect of compute speedup on training time (simulated):**Assuming network capacity remains at 10Gigabit but compute capabilities go up, we observe in that case PowerSGD will end up providing significant benefit, meanwhile synchronous SGD will end up being communication bound and will not be able to utilize increased compute. Showing that if compute capabilities increase drastically but network bandwidth remains stagnant, gradient compression methods will become useful.

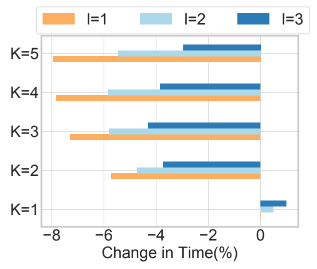
values from our analytical performance model thus verifying that our performance model can indeed be useful in several settings.

Effect of faster compute. Next we analyze how the effect of gradient compression changes when newer hardware with higher compute capabilities arrive in future.

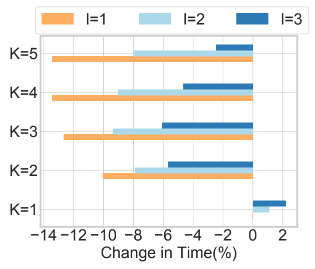
In Figure 4.13, we plot the effect of compute capabilities improving by up to 4×, while network bandwidth remains constant at 10 Gbps. We can see that for Resnet-50, PowerSGD with Rank-4 can provide 1.75x speedup if the compute becomes around 3.5x faster.

There are two reasons for this, (i) As compute gets faster, the encode-decode time also reduces by the same factor, (ii) with a faster backward pass, there is less opportunity for synchronous SGD to overlap computation with communication, making it communication bound.

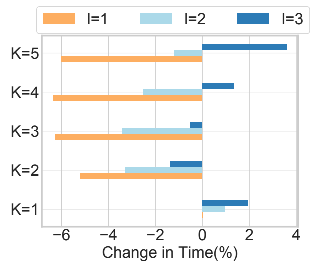
Tradeoff between encode-decode time and compression ratio. Finally, we explore the tradeoff between the effect of reducing encode-decode



(a) ResNet50: BSize 64



(b) ResNet101: BSize 64



(c) BERT: BSize 12

Figure 4.14: **Varying encoding-decoding time and compression (simulated) :** We observe that reducing encode-decode time even if it leads to reduced gradient compression is very useful and can make methods like PowerSGD more viable.

time, while simultaneously decreasing the compression ratios by similar proportions. For this we consider a hypothetical gradient compression scheme in which if we decrease encode-decode time by a factor k the size of gradients communicated increases by lk. For example, if say $k = 2$ and $l = 2$ then a 2x decrease in encode-decode time would be accompanied by a 4x increase in size of gradients. This setup is to study what would happen if we had compression schemes that offered a variety of trade-off points. We vary k from 1 to 4 in increments of 1 and try 1,2 and 3 as values of l. Using PowerSGD with Rank-4 as the baseline, we see in Figure 4.14 that any reduction in encode-decode time even at the expense of increased communication helps.

4.2.6 Key Takeaways

Here we summarize the key takeaways from our experiments and performance model-

- In Section 4.2.1 we show gradient Compression methods are not good candidates for overlap with gradient computations on popular GPUs like V100s since both gradient compression and computation are compute heavy processes leading to an overall slowdown.

- In Section 4.2.2we show existing gradient compression methods provide limited benefits either due to encoding overheads or due to lack of compatibility with all-reduce across a range of models.
- In Section 4.2.3 we observe that using large batch sizes often provides enough opportunity for syncSGD to overlap communication with communication thus reducing the extent of benefits achieved from using gradient compression.
- In Section 4.2.5, with the aid of our performance model we show that even at small batch sizes we do not need extremely high amount of compression as proposed by several works.

4.3 Conclusion

In this work, we study several gradient compression methods used to accelerate distributed ML training. We discover that existing gradient compression methods provide marginal speedups in a datacenter setup due to the overheads in compression. We develop a performance model that can help algorithm designers build scalable gradient compression algorithms. Our performance model also allows users to conduct what-if analyses and determine how much compression they need given a hardware setup. We believe this analysis provides the community clarity on the desirable properties for gradient compression and will lead to methods that can provide improved scalability in the future.

Part III

Reducing Data Movement in Recommendation Model Training

5 ACCELERATING RECOMMENDATION MODEL TRAINING

In this chapter we focus on improving recommendation model throughput by reducing the embedding access overhead.

5.1 Preliminaries

Recommendation models power widely used large-scale internet services. Recently deep learning is being used to improve the accuracy of recommendations [129, 30, 78]. All deep recommendation models consist of two types of components (i) a memory-intensive embedding layer that stores a mapping between the categorical features and their numerical representations (ii) a compute-intensive neural network-based modeling layer which models interactions between numerical features and vector representations of categorical features. Across models, the structure of embedding tables typically remains the same. The number of rows (elements in the table) usually depends on the dataset, i.e. the number of categories, while machine learning engineers vary the dimension of embedding i.e. the size of the vector to represent a categorical feature. Common dimensions of embeddings are 16, 32, 48, and 64 but sometimes can be as large as 384 [126]. However, the rows in embedding tables vary widely and can be as small as 3 elements or as big as a few billion elements depending on the dataset [209, 60]. Neural network layers have more diversity, and the type of neural network usually depends on the modeling task at hand. DLRM [129], a recommendation model popular at Meta uses fully connected layers for both bottom and top neural networks. While, DeepFM [58] uses a factorization module that learns up to two-order feature interactions between sparse and dense features (Table 5.2 summarizes other models we consider in this paper). The forward pass of training involves looking up embedding vectors corresponding to the data items.

Table 5.1: Dataset and their embedding tables

	Training Input			Embedding Tables		
Datasets	Datapoints	Categorical Features	Num Features	Num Emb	Embedding Dimension	Table Size
Kaggle Criteo	39.2 Million	26	13	33.76 Million	48	6 GB
Avazu	40.4 Million	21	1	9.4 Million	48	1.7 GB
Terabyte	4.37 Billion	26	13	882.77 Million	16	157 GB

Table 5.2: **Model Descriptions:** For DLRM, W&D, D&C the numbers indicate the structure of the different Fully Connected (FC) layers. For DeepFM, Linear Features represent a linear layer that is used to store feature interactions. For all the models, we used the standard architectures as suggested by original authors.

Model	Architecture of Dense Parameters	Number of Dense Parameters
DLRM [129]	FC - 13-512- 256-64-48 FC - 1024-1024-1024-256-128-1	2962289
W&D [30]	FC - 13-256-256-256	136673
D&C [192]	FC - 1024-512-256-64-48 FC - 1024-512-256-1	2718609
DeepFM [58]	Linear Features - 33762577-1 FC - 1248-64-64-64	33851283

All deep learning based recommendation models [193, 198, 195, 182, 35] use this step to handle categorical features such as location, product type, gender, etc. Our focus is to reduce data access overheads that arise from performing embedding lookups and thus speed up the training of all recommendation models which use embedding tables.

Training Recommendation models Next, we discuss the state-of-the-art systems used for training recommendation models.
Offline Training vs Online Training. Recommendation models are trained in both online and offline mode. Offline training involves training the model on large amounts of historical data with emphasis on throughput. Alternatively, online training is performed only on the recently acquired data to account for latest user preferences and is latency sensitive. To boost

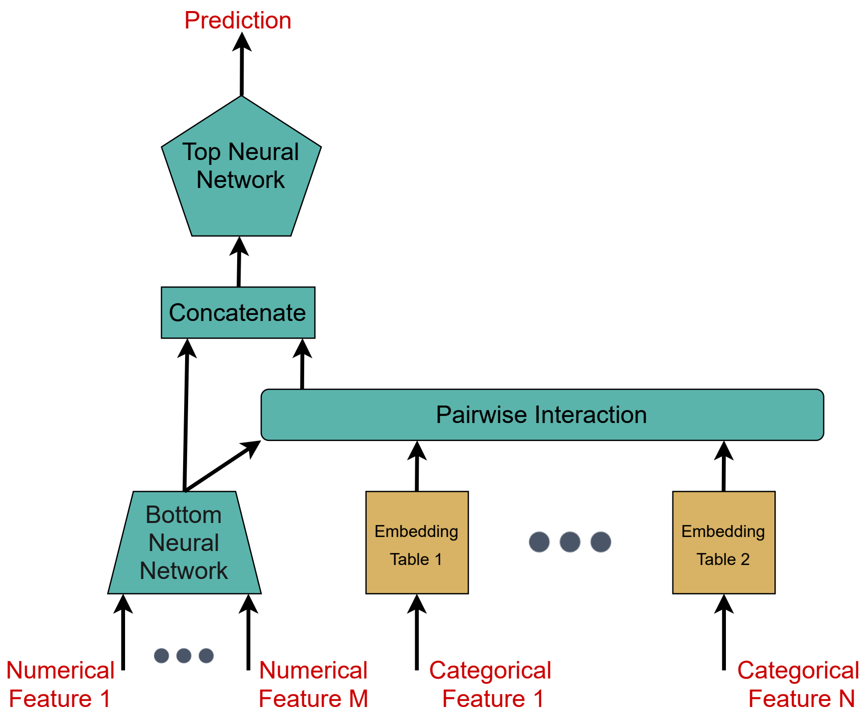


Figure 5.1: **Architecture of a recommendation model**: Model parameters include top, bottom NNs and embedding tables.

performance and prevent catastrophic forgetting [52, 98], researchers actively perform offline training, even for models in production. According to a study by Meta [7] offline training is responsible for more than 50% cycles of all ML model training cycles. This shows that offline training of recommendation models is an important workload. In this work our primary focus is on offline training of recommendation models.

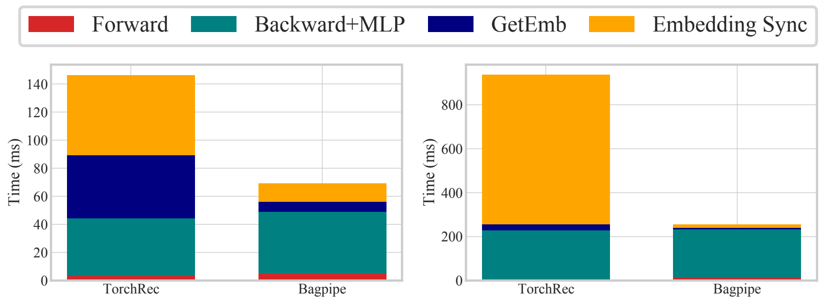
Training Setup. Recommendation models are extremely large and are currently among the largest ML models used in enterprises. Meta recently released a 12 Trillion parameter recommendation model [126]; in comparison GPT-3 has 175 Billion parameters. However, embedding tables with sparse access patterns account for more than 99% of parameters.

The combination of extremely large model sizes with the sparse ac-

cess pattern introduces several new challenges in distributed training. Figure 5.1 shows a schematic of a deep learning based recommendation model. In a typical DLRM training setup, dense neural network (NN) parameters are replicated and stored on the GPUs and trained in data-parallel fashion, where gradients are synchronized using the all-reduce communication collective. However, embedding tables are extremely large to hold in the GPU memory and are usually partitioned.
Existing Systems. Several systems have been designed to perform offline recommendation model training due to it's popularity. Training systems like TorchRec [118], FB-Research's DLRM [141] and HugeCTR [74] partition the embedding table across different GPUs and train them in a model-parallel fashion. Embeddings are fetched using all-to-all collective [125]. While, TorchRec tries to overlap embedding-related operations, like remote embedding reads and writebacks, with the compute-intensive portion of the neural network, the amount of embedding data that needs to be fetched still adds significant overhead during training. Figure 5.3 shows a breakdown of the time taken for one iteration of training when using TorchRec [118]. We observe that when using 8 training machines (AWS *p3.2xlarge instances*), the overheads when compared to an ideal baseline that does not perform any embedding lookups, is around 70% for the DLRM [129] and 75% for DeepFM [58].

Beyond spending a majority of time in embedding lookups, existing systems also couple storage and compute resources, e.g. if the embedding tables for a model are extremely large but the compute requirements are small, one still has to use a large number of GPU machines to store the embedding tables. This often leads to sub-optimal use of resources.

To alleviate the embedding access overhead and improve resource utilization, FAE [8] performed an analysis of embedding accesses and observed a similar skew in embedding access patterns. However, FAE uses a reordering approach by dividing examples into hot and cold batches



(a) DLRM (b) DeepFM

Figure 5.3: **Training Time Breakdown:** Average time spent in various stages of training when using 8 *p3.2xlarge* instances with TorchRec [118] (left) and Bagpipe (right) on DLRM and DeepFM models (Table 5.2). For large models like DeepFM, we observe that TorchRec spends 75% of each iteration on embedding access, while Bagpipe can bring it down to 10%.

based on their embedding accesses, this impacts the statistical efficiency as training continuously with hot batches changes the order of the training examples and can affect convergence [8]. Further, it is not always possible to create batches that only access cached embeddings, because some models [107, 103] use features like Unique User ID (UUID) or Session ID [178, 167, 179] that are unlikely to be repeated and thus requiring at least one cache miss per example.

Several other prior works [73, 7, 61, 119, 109] have proposed using asynchronous training to reduce embedding access overhead. With asynchronous training, embedding fetches can happen in the background, e.g. in HET trainers can use embeddings that are stale up to a certain number of iterations. If embeddings are stale beyond the bound HET synchronizes those embeddings with the embedding server before a training iteration. However, similar to other ML models, recent works [73, 126] have observed that asynchronous training can lead to degradation in accuracy for recommendation models. Accuracy degradation is *unacceptable* to large enterprises as it often directly leads to a loss in revenue [126]. Asynchronous training is also avoided due to the lack of reproducibility, which is necessary to reason about and compare different model versions. Therefore, in this work we focus on designing a system for *synchronous distributed training*.

5.2 Design

We begin by providing an overview of our design.

5.2.1 Design Overview

Bagpipe consists of four components that collectively perform training as shown in Figure 5.4. Each iteration of training begins with sampling a batch of examples, the `DataProcessors` pre-process the examples and send them to the Oracle Cacher. Based on the examples, Oracle Cacher runs a *lookahead algorithm* to determine embeddings to prefetch and to cache, and dispatches this information to the `Trainers`. The trainers typically run on GPU machines and perform gradient computation. Trainers hold the dense parameters of the model and Bagpipe's cache in the GPU memory. Also, trainers fetch necessary embeddings from the `EmbeddingServers`. Embedding servers hold the embedding tables of the recommendation model. This design introduces the following contributions:

- Bagpipe utilizes both caching and prefetching to reduce embedding access overhead. Given an offline training regime, we introduce the concept of *lookahead*, where we can look beyond the current batch and decide which elements to cache and prefetch (§5.2.2).

- We extend our scheme to the distributed setting and introduce a logically replicated, physically partitioned cache design (§5.2.3) to minimize communication overheads. To further reduce synchronization overheads we use CPA, to selectively synchronize parts of

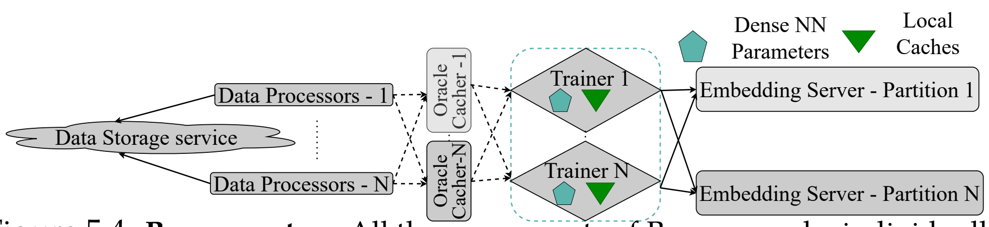


Figure 5.4: **BAGPIPE setup:** All the components of BAGPIPE can be individually scaled. The dashed arrows signify async RPCs while solid ones signify sync RPCs.

the cache that are immediately needed on the critical path while synchronizing the rest in the background.

- Finally, we discuss how BAGPIPE's dis-aggregated design can help improve efficiency, by scaling components depending on the properties of the dataset and the model, and enable low-overhead fault tolerance (§5.2.4).

5.2.2 Caching and Prefetching in BAGPIPE

In BAGPIPE, we introduce the idea of each trainer having a *local* cache. When designing a system with caching, we need to design a cache insertion policy (what to cache?) and a cache eviction policy (what to evict?) so as to maximize the hit rate. However, offline batch training of machine learning jobs like recommendation model training has additional structure: *in offline batch training future batches and their contents are predictable*, i.e. in context of recommendation models we can look beyond the current batch and infer which embeddings will be accessed in future batches. This insight helps us create a perfect or *oracular cache*. To utilize this insight, we design a lookahead algorithm. For ease of explanation, the discussion in this section assumes there is only one trainer (only one cache). We extend this to the distributed setting in §5.2.3.

Lookahead Algorithm. To decide what to cache and what to evict we develop a low overhead lookahead algorithm which also ensures consistent

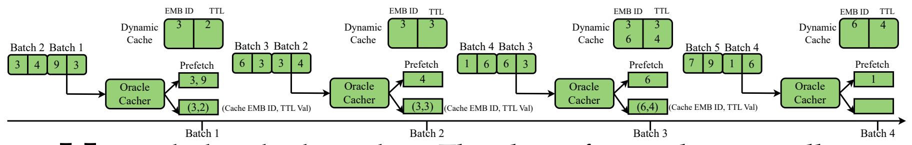


Figure 5.5: **Lookahead Algorithm**: The above figure shows an illustration at different batch steps of how the lookahead algorithm functions. In the above example, the lookahead value is 2 and the batch size is also 2.

access to embeddings. We denote the *lookahead value* ($\mathcal{L}$), as the number of batches beyond the current batch, which will be analyzed to determine what to cache, e.g. if the current batch is x, we consider embedding accesses in batches from x to $x+\mathcal{L}$, to determine which elements in batch x should be cached. The lookahead algorithm takes three inputs: the current batch, future batches (next $\mathcal{L}$ number of batches), and current state of the cache on the trainer.

The lookahead algorithm outputs two pieces of information. First, for the current batch, it generates the list of embeddings that will not be found in the cache on the trainer's GPUs. This allows Bagpipe to *prefetch* these embeddings out of order before the current batch is used for training. Prefetching allows Bagpipe to hide the data access latency for the long tail of embeddings that are not frequently accessed. Second, the lookahead algorithm determines *which embeddings **from the current batch** will be used in future batches*, and the last iteration they will be accessed in the current lookahead window. Any embeddings from the current batch that will be used by future batches in the *lookahead* window will be marked for caching, so they can be accessed from the GPU memory in the future. The last iteration an embedding is used within the *lookahead* range and serves as time-to-live (TTL) for the embedding in the cache.

Next we describe an example of how lookahead algorithm processes batches (Figure 5.5), with lookahead value ($\mathcal{L}$) as 2:

- **Batch 1** Embedding 3 and 9 are in the batch. For both embeddings we launch prefetches. However, embedding 3, is accessed again, and the last occurrence in the window is at Batch 2, so we cache it with the TTL set to 2.

- **Batch 2** Embedding 3 is in the cache so we do not send a prefetch request. But the last occurrence for 3 in the window is now in Batch 3. Therefore, a TTL update is sent for 3. We will prefetch 4 since it is not in the cache.
- **Batch 3** We prefetch embedding 6 and cache it with a TTL of 4 since it will be reused in Batch 4. At this point in our lookahead window Embedding 3 has no future occurrence so it will be evicted after batch 3. However, if embedding 3 was being used by batch 4, we would have kept it in the cache and sent a TTL update with eviction batch as 4.
- **Batch 4** We prefetch 1 since it is not in the cache. We do not send any TTL updates for 6 as it is absent in future batches and will be evicted after this batch.

Consistency with the Lookahead algorithm. Our consistency goal is to avoid staleness and ensure that trainers do not prefetch an embedding from the embedding servers while it has updates that have not yet been written back. Despite pre-fetching embeddings out of order, our formulation of what to cache and what to prefetch , provides an extremely important guarantee that allows us to maintain consistency and match the execution of synchronous training. When the trainer is processing batch number x, an embedding used by the batch will either be available in the cache with it's most recent value or no preceding batch in the lookahead range (any batch number in $[x - \mathcal{L}, x)$) would have updated that specific embedding. That is, if an embedding was needed by a batch in batch number in range $[x - \mathcal{L}, x)$ it will be in the cache; if an embedding is not in the cache it means no batch in range of $[x - \mathcal{L}, x)$ has updated it. Therefore, as long as the prefetch request for batch x is issued after updates from training batch number $x - \mathcal{L}$ have been written back, we can guarantee that we will not see stale embeddings.

5.2.3 Distributed Cache Design in Bagpipe

First we discuss the requirements for the distributed cache design and our goals. Next, we discuss the design space for cache design and finally we compare these designs both quantitatively and qualitatively.

Distributed Cache Requirements. When extending the caching scheme described above to a distributed setting, Bagpipe can provide consistency as long as the following two requirements are satisfied (i) Each trainer sends prefetch requests for batch number x only when cache eviction and updates have been performed by *all* the trainers on $x - \mathcal{L}$ batch. (ii) Each trainer's cache should contain the latest value of the embedding. The first requirement is a direct extension of our prior discussion and can be satisfied by synchronizing the iteration number that each trainer has processed. The second condition, however, creates additional communication overheads and we next discuss the design space and techniques to reduce these overheads.

Goals of distributed cache design. The primary objective of our distributed cache design is to minimize the *time spent* on cache synchronization on the *critical path*. Thus our objective includes, accounting for the number of bytes transferred (bandwidth) and connection overheads (latency) [172].

Next, we explore the distributed cache design space and discuss synchronization costs with each design.

Replicated Cache. In a replicated cache, each trainer will pre-fetch *all* the embeddings which are required by the whole batch (not just a worker's partition of the batch). After performing the backward pass we synchronize all the elements which have updated gradients across all the workers, such that embeddings in the caches are synchronized at the end of each iteration using *all-reduce*. This trivially ensures that each trainer's cache has the latest version of the embedding. A replicated design results in high bandwidth cost due to synchronization of all the elements across all trainers even if the element's updated value would not be required in future by other trainers, i.e. it will be evicted from the cache. However, there is very small control (latency) overhead because all elements are synchronized.

Partitioned Cache. A partitioned cache is on the other end of the design spectrum where each trainer is assigned an *exclusive* portion of the cache. Before the forward pass of training, each trainer fetches the embeddings not available locally from their peer trainers. Post backward pass, each trainer writes back the gradients to the respective peer trainer which has ownership of the embedding. These steps are required to ensure we always use the latest version of the embedding. Unlike the replicated cache, where all the embeddings are synchronized irrespective of whether a trainer needs it, in the case of a partitioned cache, trainers only fetch and write back the embeddings they utilize. Further, partitioned caches are more space efficient as there is only one copy of each embedding in the distributed cache.

The number of bytes communicated when using a partitioned cache depends on how batches are partitioned across trainers. To study the scenario where batch partitioning is communication aware, i.e. batches are partitioned so as to minimize bytes communicated across trainers, we formulate a mixed integer linear program (MILP). Given the cache state on all trainers, the MILP computes a partitioning of examples which minimizes the amount of inter-node communication. Given a batch of examples (b) and p trainers, we introduce $b \times p$ variables in our MILP. Each variable is denoted by $x_{i,j}$ where if $x_{i,j} = 1$, then example i will be assigned to trainer j. We then compute a cost matrix C, where given the cache state, $C_{i,j}$ represents the cost of inter-node communication that will be required to fetch embeddings for example i to location j. Our objective is to minimize the amount of inter-node communication. We formulate it using our variables and cost matrix as:

$$\text{Minimize} \quad \sum_{i \in I} \sum_{j \in J} C_{i,j} \cdot x_{i,j}$$

Where I and J represent the set of examples and trainers respectively. Further, we include two constraints to ensure that the solution is feasible and avoids load imbalance:

(i) Each example must be placed on one trainer and all examples need to be placed on at least one trainer node. $\forall i \in I \quad \sum_{j \in J} x_{i,j} = 1$. (ii) We add another constraint to make sure the batch is equally distributed across machines to prevent load imbalance. The optimization problem can be solved using existing MILP solvers like Gurobi [62].

However, using communication aware partitioned caches has two disadvantages: first solving the MILP takes around 2.36s on a 16-core machine making it infeasible when iteration times are around 100ms (Figure 5.3). Secondly, sync time does not solely depend on bytes communicated, as overheads from maintaining data-structures and establishing connections also play a role. With partitioned caches we would need to introduce additional data structures to keep track of embedding locations and establish multiple connections.

Logically Replicate Physically Partitioned Cache. Ideally, we would like to design a cache that does not perform unnecessary synchronization of embeddings but does not introduce additional overheads due to state tracking. To achieve this goal, we propose using Logically Replicated, Physically Partitioned (LRPP) caches. By *logically replicated* we mean that from the view of Oracle Cacher all caches have all data and are fully replicated but by being *physically partitioned*, the trainers decide which elements need synchronization and which elements can be evicted without

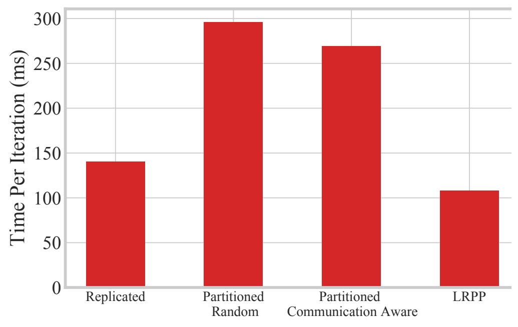


Figure 5.6: **Comparing cache designs:**We observe that LRPP provides best performance among all other cache options.

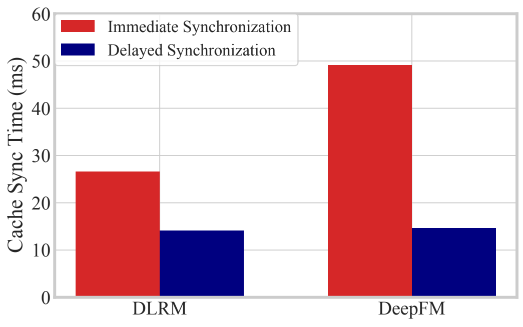


Figure 5.7: **Effect of Delayed Synchronization**:Delayed Sync can reduce time for cache synchronization by up to 44%.

synchronization. The primary insight behind our idea comes from the observation that for the Criteo dataset, around 25% of the embeddings are used by only one of the examples in batch. Therefore, these embeddings are updated at only one trainer before being evicted. Thus, fetching or synchronizing them across all trainers is a waste of network bandwidth. We design a new protocol that modifies the replicated cache based on this insight.

With LRPP caches, the Oracle Cacher marks embeddings which are only used by a single trainer. Given this metadata, these embeddings are only fetched by the trainer which needs them and are ignored by other trainers. After the forward and backward pass completes, the trainers skip synchronization for these embeddings and use all-reduce to synchronize the other embeddings. In the background, the trainer which made the only update to the marked embedding evicts it back to the Embedding Server. This optimization is able to reduce the volume of embeddings prefetched and synchronize with very minimal control logic. LRPP can be further extended with more fine-grained partitioning, i.e. we can synchronize embeddings updated by two workers using a separate communication group containing just those workers. However, further fine-grained partitioning will create additional control logic, which in turn would add additional latency and thus yield diminishing returns.

There are parallels between design of LRPP to [207] a concurrent work which only caches elements which are going to be utilized more than once with a FIFO eviction policy. This is analogous to LRPP only synchronizing elements which are going to be used in future. We plan to study extensions of LRPP in future work.

Comparing cache design choices. For Kaggle critieo dataset with batch size 16,384 and 8 trainer machines (p3.2xlarge) we observe that Replicated Cache communicates around 65K embeddings per iteration, while Communication Aware-Partitioned Cache communicates 21K embeddings per iteration and LRPP communicates around 48K embeddings. Further, we implement all these in Bagpipe and evaluate them in terms of per-iteration training time using the same setup. To consider the best case scenario for partitioned caches, we ignore the time taken by the Gurboi solver. In Figure 5.6, we observe that LRPP outperforms replicated by 22.8% and communication aware partitioned by 59.8%. Our analysis shows that despite synchronizing fewer embeddings partitioned caches do not perform well due to hotspots and additional control logic. Since some embeddings are accessed extremely frequently, the trainers that own those embeddings become a bottleneck. Further, in partitioned caches, the overhead of performing multiple collective communication calls, creating memory buffers for each collective communication call and tracking which peer to access embeddings from, leads to an additional overhead of 80-90ms for a batch size of 16K with 8 trainers. Therefore, we configure Bagpipe to use LRPP cache synchronization scheme due to it's superior performance.

Delayed Synchronization. To further optimize the LRPP protocol we use Critical Path Analysis [206]. In this scenario, CPA implies that as long as the embeddings are synchronized before being critically required it can suffice. However, directly using CPA in context of embedding synchronizations for recommendation models will lead to network contention with other competing synchronizations. Therefore, we introduce delayed synchronization, where we only synchronize the embeddings which will be required in the next iteration on the critical path. The embeddings which are not needed immediately are synchronized in the background. To, avoid network contention due to background synchronization we ensure that all background synchronizations are completed before we launch other critical path synchronizations for future iterations. On Kaggle Criteo dataset with 8 trainers and batch size 16K, we see that only 22.7K embeddings out of 48K embeddings (47.3%) need to be synchronized on the critical path, the rest can be overlapped with the forward pass of the next iteration. For two models on Criteo dataset, Figure 5.7 shows that delayed synchronization can further reduce cache synchronization time by up to 44% (in addition to LRPP) by overlapping synchronization with forward pass. We also observe that LRPP and delayed synchornization can together reduce bytes communicated on critical path by around 70%.

5.2.4 Disaggregated Design and Fault Tolerance

Existing recommendation model training systems [141, 126, 118] couple storage and compute resources, i.e. it is not possible to scale the number of embedding table partitions without increasing the number of trainers. This affects fault tolerance and resource utilization. For fault tolerance, given the extremely large embedding table sizes, checkpointing a trainer can take several minutes [46] during which the compute resources stay idle. The lack of *disaggregation* also leads to poor resource utilization [36, 18], e.g. when embedding tables are extremely large but the dense neural network parameters are small, an optimal configuration would be to use more servers for embedding tables but have fewer trainers. Thus, we design a disaggregated architecture for Bagpipe (Figure 5.4) with four major components: (i) Data Processors (ii) Oracle Cacher (iii) Distributed Trainer (iv) Embedding Servers.

Data Processor. Data processors read and batch training data which is resource intensive. Similar to prior designs [220] we offload data processing to reduce trainer overheads. Data processors are stateless and can be restarted on failure.

Oracle Cacher. Oracle Cacher is a centralized service that inspects all the training batches using the lookahead algorithm . Oracle Cacher decides which elements to prefetch for the current batch and the TTL for eviction of elements being cached. Oracle Cacher sends the training data as well as the embedding ids that need to be cached/prefetched using async RPC calls to the trainers. Oracle Cacher is designed such that all the necessary internal state is also present on the trainers. Therefore, whenever Oracle Cacher has to be restarted we only need to fetch the last iteration number processed by the trainers and the embedding IDs present on them.

Trainer. Trainers hold the dense neural network portion of the recommendation model and the LRPP cache in the GPU memory. The trainers perform forward and backward passes in a synchronous fashion. Trainers also: (i) prefetch the embeddings based on requests sent by Oracle Cacher (ii) perform cache maintenance including addition and eviction of embeddings. When a trainer fails, Oracle Cacher makes an RPC call to ask existing trainers to checkpoint their state (model parameters and cache contents) and then copies this state to the newly started trainer. Each of the trainers then discard their gradients and Oracle Cacher starts from the previous iteration. With delayed synchronization and LRPP enabled we might loose updates of at most one iteration, which is unlikely to affect model convergence [160].

Embedding Server. Embedding servers store all the embedding tables and act as a sharded parameter server, handling the prefetch and update requests from the trainers. We use the techniques presented in prior work [46] to checkpoint embedding servers periodically.

5.2.5 Discussion

Next, we discuss some benefits and limitations of our design.

Generalizing across skew patterns. Unlike prior work [8], Bagpipe's optimizations are resistant to embedding access skew changes (evaluated in §5.4.3). This is because Bagpipe does not just rely on caching of a fixed set of hot embeddings, it speeds up access to cold embeddings using prefetching. So if there exists datasets that do not display a high degree of skew, Bagpipe will still outperform prior work.

Applicability in online training. Bagpipe's optimizations are applicable to offline setup, as it relies on the ability to look at future batches to build a cache. In case of online training examples arrive sporadically thus restricting lookahead.

Scalability of Oracle Cacher. The overhead of Oracle Cacher is extremely small even for extremely large batch sizes and lookahead values. In §5.4.3 we find that Oracle Cacher, even for large batch size of 131K, can dispatch 3.27 Million samples per second. Further, Oracle Cacher only needs to be faster than the time taken by trainers for the forward and backward pass. However, if required, Oracle Cacher can be partitioned to increase scalability for datasets with a large number of embedding tables. To split the work done by Oracle Cacher, we can partition the embedding tables such that each partition of the Oracle Cacher can work on a different embedding table. For instance, if there are 1000 categorical features and we launch 10 Oracle Cacher; for each example, each Oracle Cacher generates caching decisions for their subset of 100 categorical features.

5.3 Implementation

Bagpipe is implemented in around 5000 lines of Python. Async RPC's are used to communicate across different components. For synchronization of dense parameters and caches we use collective communication primitives present in NCCL [1]. Bagpipe is completely integrated with PyTorch and existing model training code can use it with 4 to 5 lines of changes. API details will be present in our open source version.

Overlapping cache management with training. We perform, all cache management operations in a separate thread thus not affecting the training process. Our caching data structure can operate completely lock free, because in our Oracle Cacher's lookahead formulation, we guarantee that the training thread and cache maintenance thread will operate on completely separate indices of the cache. This ensures that cache management has minimal overhead on training.

Automatically Calculating Lookahead. Bagpipe uses two configuration parameters: max cache size and lookahead value ($\mathcal{L}$). Providing the max cache size is mandatory, it can be determined by computing amount of free memory available after allocating space for the dense neural network parameters. $\mathcal{L}$ can be automatically calculated if it is missing. To calculate $\mathcal{L}$, at startup Bagpipe keeps prefetching until it detects the cache is full. On detecting that the cache is full, Bagpipe selects the number of batches prefetched so far as the $\mathcal{L}$. Further, Bagpipe can also handle scenarios where the configuration variables are incompatible. Since Oracle Cacher always has a consistent view of the cache, if it observes that the cache is going to be full it can reduce the $\mathcal{L}$. We perform a sensitivity analysis on $\mathcal{L}$ in §5.4.3.

5.4 Evaluation

We evaluate Bagpipe by measuring improvements in per iteration time against four baselines, observing a speedup of 2.1× to 5.6× for the DLRM model. Further, we vary the recommendation model architecture and compare Bagpipe against the best-performing baseline with four different models and observe a speedup of up to 3.7×. We also analyze the performance of BAGPIPE on different hardware and datasets and evaluate other aspects of BAGPIPE like fault tolerance (§5.4.2) and sensitivity to configuration parameters (§5.4.3).

Baseline Systems. To compare BAGPIPE we use four open source baselines discussed in §5.1. We compare BAGPIPE with FAE [8], FB-Research's training system [141], TorchRec [118] and HET [119]. We discuss additional details of these systems when comparing them with BAGPIPE in §5.4.1.

Models and Datasets. We use four different recommendation models, Facebook's DLRM [129], Google's Wide&Deep [30], Deep&Cross Networks [192], and Huawei's DeepFM [58]. Table 5.2 describes the models used to evaluate BAGPIPE. The models differ markedly in terms of the dense parameters, e.g. the largest model has 33.8 Million parameters while the smallest one only has 136K parameters. For datasets, we use the Kaggle Criteo [100], Avazu [76] and Criteo Terabyte dataset [101] (largest publicly available dataset). Table 5.1 describes the embedding table size for each dataset.

Cluster Setup. We run all our experiments on Amazon Web Services(AWS). For trainers, we use *p3.2xlarge* instances while Embedding Server and Oracle Cacher run on a *c5.18xlarge* instance each. Each *p3.2xlarge* instance contains a Nvidia V100 GPU, 8 CPU cores and 64 GB of memory with inter-node bandwidth of up to 10 Gbps. Each *c5.18xlarge* has 72 CPU cores and 144 GB of memory. For BAGPIPE we launched dataloaders on the same *c5.18xlarge* as Oracle Cacher since the machine had ample compute. To study the performance of BAGPIPE in a setting with different amounts of compute and bandwidth we also run some experiments on *g5.8xlarge* where each machine has an Nvidia A10G GPU,32 CPU cores with inter-node bandwidth of 25 Gbps.

BAGPIPE Configuration. Unless otherwise stated, for all our experiments we set the cache size to enable lookahead of up to 200 batches. We study the sensitivity of these parameters and their effect on throughput in §5.4.3. We run all our experiments for 2000 iterations, which roughly translates to 1 epoch of Criteo Kaggle Dataset with batch size of 16,384.

Metrics. For all our experiments we plot average per-iteration time with error bars representing standard deviation. This directly translates to the time taken to train a fixed number of epochs. As Bagpipe guarantees consistent access to embeddings, the accuracy after each iteration exactly matches other synchronous training baselines (validated in §5.4.1).

5.4.1 Comparing Bagpipe

We first evaluate Bagpipe by comparing it against a number of existing systems and study how our benefits change as we vary the models, datasets, and hardware.

Comparing Bagpipe with existing systems. In Figure 5.8, we compare Bagpipe with four existing systems, FAE [8], FB-Research training system [141], TorchRec [118] and HET [119]. We use Criteo Kaggle dataset with batch size 16,384 (a common batch size among MLPerf [117] entries) and two popular recommendation models DLRM [129] and W&D [30].

FAE performs pre-processing on training data to classify embeddings as either hot or cold. To evaluate the best case scenario for FAE, we do not account for the additional time FAE spends in partitioning batches and deciding the placement of embeddings. As shown in Figure 5.8, Bagpipe achieves 3.4× speedups for the DLRM model and 3.7× speedups for W&D. As discussed in §5.1, during hot batch training FAE has similar cache synchronization overheads as Bagpipe, but when it switches to cold batches, it suffers additional embedding access overheads due to no prefetching.

Next, we compare Bagpipe with open source FB-Research training system [141], built over PyTorch and Caffe-2, is designed for DLRM models [129] but can be easily modified to support other embedding-based deep recommendation models like W&D [30]. Bagpipe provides 5.6× and

4.2× speedups over FB-Research training system. FB-Research system is slow due to spending almost 60% of the time on data loading, which has also been observed by prior works [220, 140] as well, leading to worse throughput compared to Bagpipe which offloads data-preprocessing to remote machines.

When compared against TorchRec, a recent open source system built over PyTorch [105] and FBGEMM [142] to facilitate training of recommendation of models, we observe that Bagpipe is around 2.1× faster for DLRM models and around 1.3× faster for W&D models. Unlike Bagpipe, TorchRec does not perform any caching or pre-fetching, and therefore fetches and writes back a large number of embeddings on the critical path. Bagpipe reduces and overlaps the amount of embedding-related communication on the critical path.

To compare with HET [119], a system that performs bounded asynchronous training as described in §5.1, we use the author-provided code implemented in C++ with Python bindings to evaluate HET and set the asynchrony bound to 100, as suggested by the authors for maximum speedup. We find that Bagpipe is around 2.3× faster than HET for DLRM and 1.6× faster for W&D. We observe that, despite performing asynchronous training, HET needs to fetch embeddings that are not available in the local cache from the parameter server on the critical path. With increase in batch size the number of cache misses increases as well, due to the long tail of accesses (discussed in . We also verify that our performance closely matches with those reported in the paper [119].

As the speedup of Bagpipe varies across models, we perform a detailed investigation to understand this. We observe TorchRec to be the best performing baseline as it efficiently overlaps different parts of the training pipeline and make better use of network bandwidth using the `all2all` primitive for embedding fetches. Thus, for the next set of experiments we compare Bagpipe with TorchRec.

Comparing Bagpipe on other models. In addition to W&D and DLRM, we also train the Deep&Cross Network (D&C) [192] and DeepFM models [58] with Bagpipe and TorchRec. D&C models contain an additional Cross Network component, which performs explicit feature crossing of sparse features to learn predictive features of bounded degree without manual feature engineering. DeepFM introduces a factorization module that learns up to 2-order feature interactions between sparse and dense features. Details of these models are available in Table 5.2. In Figure 5.9 we observe that performance gains provided by Bagpipe over TorchRec depends on the size and computation requirements of the dense portion of recommendation models, e.g. for W&D which only has around 131,000 dense parameters we observe that Bagpipe provides only a 1.2× speedup, while for DeepFM which has 33.8 million parameters Bagpipe provides a speedup of over 3.7×. We believe that this is due to the pipelining mechanism present in TorchRec where the authors overlap the embedding write-back with the synchronization of the dense model. As the model size increases, the bandwidth requirement for synchronization also increases and the synchronization of dense model and embedding write-backs ends up competing for the same set of network resources. Meanwhile, Bagpipe significantly reduces the amount of embedding synchronization due to caching. Further, delayed synchronization allows Bagpipe overlap forward pass of the next iteration. Thus, our analysis indicates that TorchRec, unlike Bagpipe, is heavily bottlenecked by the network bandwidth available. To verify this, we next run experiments on a different hardware.

Comparing Bagpipe on different hardware. To understand the performance of Bagpipe on different hardware setups, especially in terms of network bandwidth, we evaluate Bagpipe on *g5.8xlarge* (A10G GPU and 25 Gbps bandwidth) instances. In terms of compute, A10G performs similar to V100, but the bandwidth on *g5.8xlarge* is 25 Gbps compared to 10 Gbps on *p3.2xlarge*. We use the same hyper-parameters and model configurations as in previous sections. Figure 5.10 shows a comparison of per-iteration time between DLRM and DeepFM, for both Bagpipe and TorchRec. We observe that Bagpipe with DLRM model on *g5.8xlarge* trainers is around 1.9× faster than `p3.2xlarge` trainers. On other hand, the DLRM model with TorchRec on *g5.8xlarge* trainers is around 2.4× faster than *p3.2xlarge* trainers. Similarly for DeepFM, time for TorchRec reduces by 2.4× (1015ms to 414ms) when we switch from *p3.2xlarge* instances to *g5.8xlarge*. This confirms our hypothesis that for larger models TorchRec is bounded by bandwidth, while Bagpipe, because of caching and efficient pipelining of communication, makes better use of network resources. However, it is unclear yet, what fraction of the iteration time in Bagpipe is spent on network-bound embedding access and to understand this, we next compare Bagpipe to an *ideal* system which has no overhead of embedding accesses.

Comparing Bagpipe with an *ideal* system. Comparing Bagpipe to an *ideal* system will show how far Bagpipe is from completely alleviating embedding access overheads. To create such an ideal system, we prefetch all necessary embeddings to the GPU memory before starting training and switch off prefetch, cache sync, and cache eviction modules of Bagpipe. In Figure 5.14, we perform this comparison for DLRM and DeepFM models on both *p3.2xlarge* and *g5.8xlarge*. DLRM model on *p3.2xlarge* instance on *ideal* system takes around 30ms while, Bagpipe takes around 56ms. DLRM model on *g5.8xlarge* on the *ideal* system takes around 19ms while Bagpipe takes around 30ms. This shows that at lower bandwidths for DLRM, there are periods in the pipeline when model training is blocked on embedding operations. We also study the same effect with DeepFM, a larger model that provides more opportunities for Bagpipe to overlap embedding-related operations. For DeepFM we observe that on *p3.2xlarge* instances ideal takes around 236ms while Bagpipe takes around 253ms (overhead 17ms). On the high bandwidth *g5.8xlarge* instance *ideal* system takes around 116ms while

Bagpipe takes around 128ms (overhead of 12ms). These results indicate that Bagpipe gets within 10% of an *ideal* system with deeper models and has almost constant overhead for providing embeddings (around 12 to 20 ms) even at lower bandwidths.

Comparison on Different Datasets. We also analyse performance of Bagpipe on Avazu [76] and Criteo Terabyte [101] (largest publicly available dataset). Using eight *p3.2xlarge* instances as trainers, in Figure 5.11 we see that compared to TorchRec on DLRM model, Bagpipe is 1.9$\times$ to 2.4$\times$ faster. This shows that irrespective of the dataset Bagpipe provides a significant speedup over the best-performing baseline.

Convergence Comparison. Since Bagpipe ensures that embedding reads are not stale, it should have the same convergence properties as synchronous training using TorchRec. We verify this in Figure 5.12 where we see that Bagpipe's convergence is very close to TorchRec with minor differences arising from random initialization. For HET we observed that the convergence depends on the model complexity; for DLRM, HET's open source code [65] did not converge to the same loss as TorchRec, while we observed similar convergence as TorchRec for W&D, a smaller model (We have reported this issue to the HET authors). Overall, we see that Bagpipe retains the convergence of synchronous training while providing per-iteration speedups.

5.4.2 Scalability and Fault Tolerance

Scalability. In Figure 5.15a, we scale batch size and number of machines (up to 32 GPUs and batch size of 65,536). With increase in batch size the number of embeddings to fetch and synchronize increases. Despite increase in communication, Bagpipe scales around 1.4$\times$ for 2$\times$ increase in resources and work (320K samples/sec for 16 trainers vs 446K samples/sec for 32 trainers). In Figure 5.15b we scale just batch size, we observe that batch size 65,536 takes a very similar time as batch size 131,072. Because

Table 5.3: **Effect of increasing $\mathcal{L}$**: With increase in $\mathcal{L}$ the cache size required increases but improves throughput till $\mathcal{L}$ of 100.

Lookahead	Cache Size (MB)	Avg Time per Iteration (ms)
5	39.6	535.6
10	66.7	289.3
50	235.2	85.7
100	410.5	67.3
200	720.6	65.1
300	1003.3	65.6

with a higher batch size Bagpipe is able to overlap a bigger proportion of cache synchronization with the longer forward pass.

Fault Tolerance. In Figure 5.13 we observe that trainer in Bagpipe recover in less than a minute, compared to FB-research system which is close to 13 minutes. For the FB-Research system we make a best-case assumption that the framework can checkpoint the iteration just before failure, to avoid checkpointing at every iteration. Even in this case, FB-Research system takes around 13 minutes to recover since the amount of state on each trainer includes a large shard of the embeddings. Meanwhile, trainers in Bagpipe are able to recover in less than a minute and do not require checkpointing at every iteration (§5.2.4). Other systems do not discuss fault tolerance.

5.4.3 Sensitivity Analysis of Bagpipe

Next, we study the performance of Bagpipe with different configurations and also micro-benchmark components of Bagpipe. Unless stated otherwise, we use the same setup described §6.3, with a batch size of 16,384 on Criteo Kaggle dataset.

Overhead of Oracle Cacher. In Figure 5.16a, 5.16b we observe Oracle Cacher's overhead increases sub-linearly with increase in categorical features and batch size. However, the time per iteration is still significantly higher than the time taken to perform lookahead by Oracle Cacher. Since Oracle Cacher is overlaped with training, it only becomes a bottleneck if it's time exceeds that of trainer. Overall, we find that Oracle Cacher can almost dispatch *3.27 Million* examples per second. We find that this is sufficient to power the most optimized systems reported in prior work (e.g., 8 ZionEx nodes (128 A100 GPUs) processing up to 1.6 Million samples per second [126]). We benchmarked Oracle Cacher for other parameters like different $\mathcal{L}$ and observe constant throughput, i.e. complexity of Oracle Cacher does not depend on $\mathcal{L}$.

Effect of $\mathcal{L}$. In Table 5.3 we study how cache size required and throughout changes for different $\mathcal{L}$. As $\mathcal{L}$ increases, cache size required increases sub-linearly. This sub-linear behavior due to reuse of embeddings found in the previous batches during lookahead process. We also observe that throughput benefits from increasing $\mathcal{L}$ start plateauing beyond 100. This is because as the lookahead value goes over 200, we are keeping all the popular elements in the cache, and increasing $\mathcal{L}$ at this point does not affect communication much.

Effect of Access Pattern Skew. Unlike some prior systems [8], Bagpipe is designed to handle skew pattern changes. To study performance of Bagpipe when the skew pattern changes, we create an artificial dataset similar to Criteo Kaggle dataset with the same number of features and samples but with different skew patterns. We choose top 1% of embeddings and then create an exponential function such that cumulative probability of sampling from top 1% embeddings is equivalent to the chosen skew, e.g. top 1% of embeddings are responsible for 40% accesses. The remaining embeddings are sampled uniformly such that they lead to the remaining 60% of accesses. In Figure 5.17 we study how the iteration time changes as the embedding reuse of top 1% embeddings changes between 90% and 1%; e.g. the 40% bar reflects the runtime when 1% of embeddings are reused 40% of the time. We observe that due to optimizations present in Bagpipe like pre-fetching, LRPP and delayed synchronization even when the degree of skew changes from 90% skew to no skew, Bagpipe's per iteration time changes at most by 13%. On the other hand, FAE [8], which relies only on caching degrades by 7.2×. Next we vary the skew of embedding accesses using the popular Zipf [93] distribution. The α parameter in Zipf distribution determines the skew, with a higher α denoting higher skew. In Figure 5.18 we observe that even with a large change in skew (varying Zipf's parameter between 1 and 5), Bagpipe's throughput does not vary significantly. This shows Bagpipe is resistant to changes in skew.

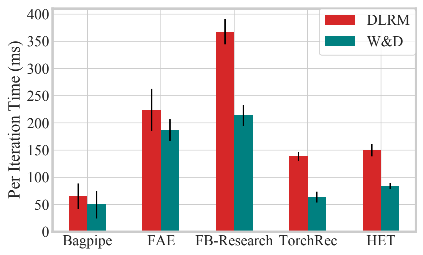


Figure 5.8: **Compare Bagpipe with Existing Systems**: We compare per iteration time of Bagpipe against existing FAE [8], FB-Research training system [141], TorchRec [118] and HET [119]. Bagpipe provides speedups betwen 1.2× and 5.6×.

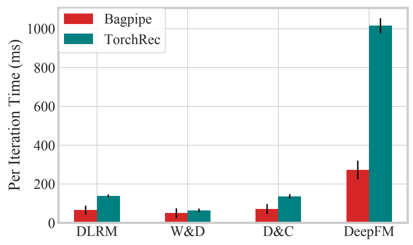


Figure 5.9: **Compare Bagpipe with different models**: We compare Bagpipe and TorchRec on four different models, DLRM [129], W&D [30], D&C [192], and DeepFM [58]. We observe speedups between 1.2× and 3.7×.

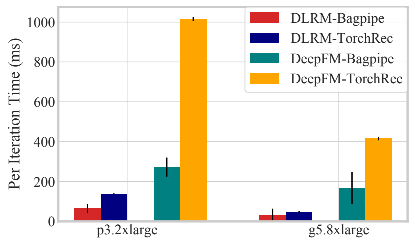


Figure 5.10: **Compare Bagpipe on different Hardware**: Speedup provided by Bagpipe over TorchRec on *p3.2xlarge* decreases from 3.7× to 2.5× on *g5.8xlarge* (high bandwidth) depicting that TorchRec is more constrained by bandwidth.

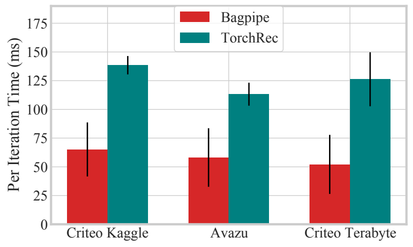


Figure 5.11: **Compare with Different Datasets**: Bagpipe consistently provides speedups between 1.9× to 2.4× across datasets.

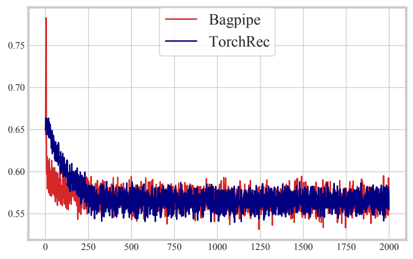


Figure 5.12: **Loss convergence for Bagpipe and TorchRec**: Convergence of Bagpipe and TorchRec is very similar, with slight differences due to random initialization.

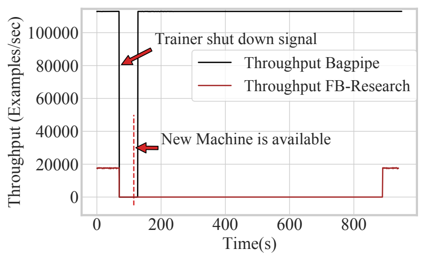
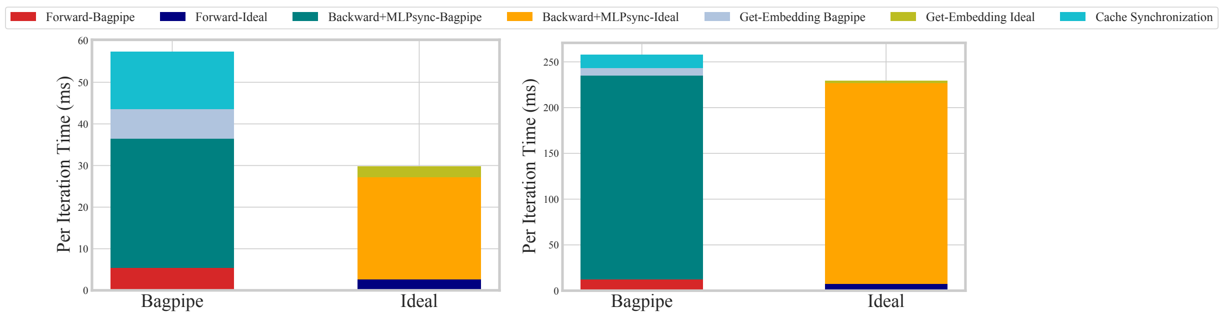
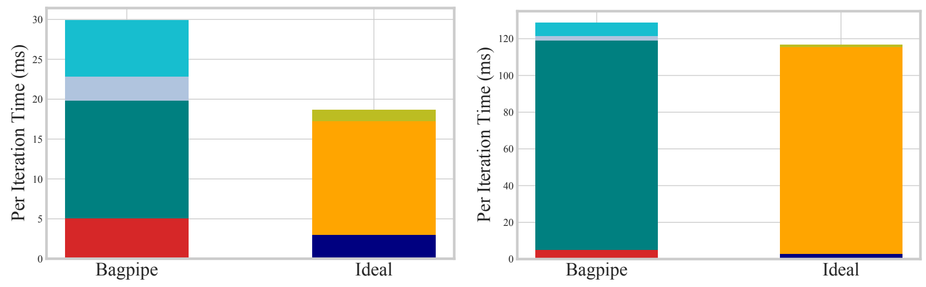


Figure 5.13: **Recovery from trainer failure**: Bagpipe requires less than 60 seconds to recover from a trainer failure compared to 13 minutes for FB-Research System.

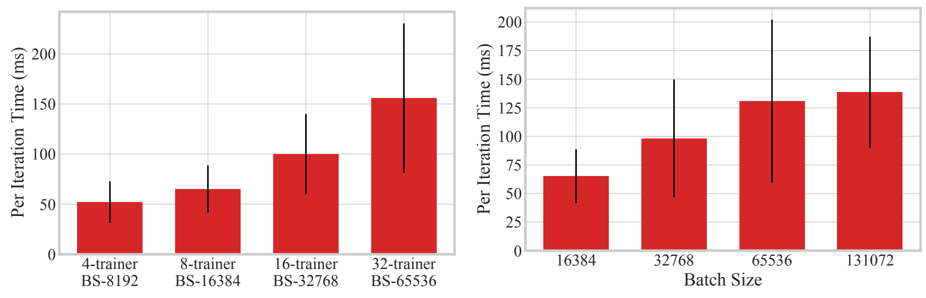


(a) DLRM: p3.2xlarge (b) DeepFM: p3.2xlarge



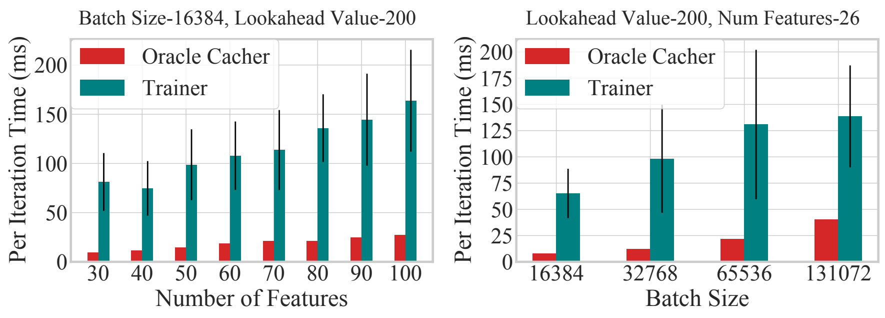
(c) DLRM: g5.8xlarge (d) DeepFM: g5.8xlarge

Figure 5.14: **Comparing with Ideal:** Comparing BAGPIPE with an *ideal* system which has no overhead for embedding fetch, we observe that system comes within 10% of time per iteration for large models where there is potential to overlap embedding accesses.



(a) **Increasing Trainers** (b) **Increasing Batch Size**

Figure 5.15: **Scalability of BAGPIPE: (left)** we increase the number of trainers such that batch size per machine is constant; BAGPIPE provides sublinear scalability due to increasing communication bottlenecks. **(right)** Increasing batch size with 8 trainers results in better throughput as we are able to better overlap communication.



(a) **Categorical Features** (b) **Batch Size**

Figure 5.16: **Latency of Oracle Cacher**: We observe that overall Oracle Cacher scales very well, it increases sub-linearly with the increase in the number of features and batch size. However, training time will always hide the latency of Oracle Cacher.

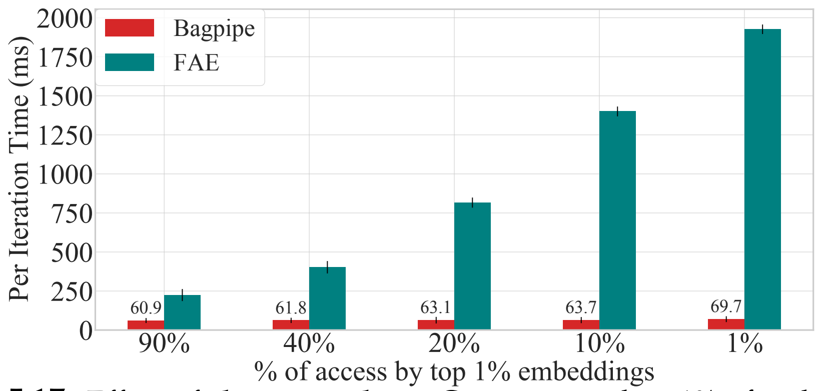


Figure 5.17: **Effect of change in skew:** Comparing when 1% of embeddings perform 90% of embedding accesses to just 1% of embedding access (no skew). Unlike FAE, Bagpipe's time only increases from 60.9ms to 69.7ms showing resistance to change in skew.

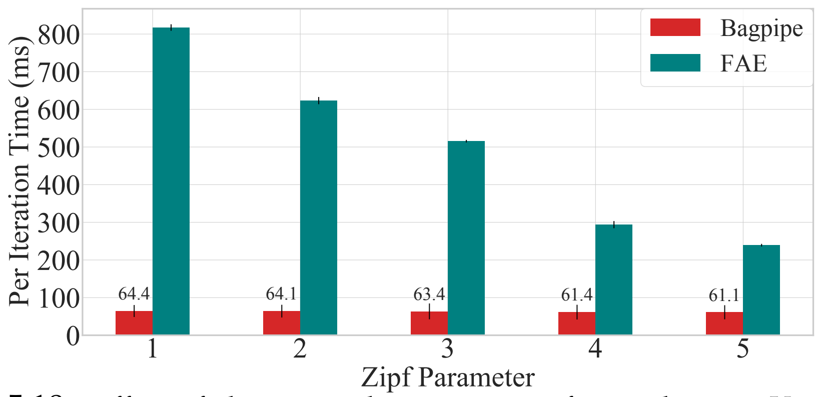


Figure 5.18: **Effect of change in skew using Zipf Distribution:** Varying the α parameter in Zipf distribution; a higher α indicates higher skew. Even with drastic increase in the skew, the time taken by BAGPIPE remains almost constant.

5.5 Conclusion

We presented Bagpipe, a new system that can accelerate the training of deep learning based recommendation models. Our gains are derived from better resource utilization and by overlapping computation with data movement. Our disaggregated architecture also allows independent scaling of resources and better fault tolerance, while retaining synchronous training semantics. Our experiments show that Bagpipe provides an end-to-end speedup of up to 5.6× over state-of-the-art baselines.

Part IV

Reducing Memory Bandwidth Requirement for LLM Inference.

6 CLUSTERED HEAD ATTENTION

In this chapter we introduce Clustered Head Attention a variant of Multi-Head Attention which reduces both memory bandwidth and compute requirement for Multi-Head Attention.

6.1 Preliminaries

LLMs have demonstrated remarkable performance on language modelling tasks ranging from question answering, text summarizing, language translation. However, such performance has been achieved by scaling models to trillions of parameters, and existing works [67, 176, 87] show that increasing the model size may lead to even higher model quality.

Inference on LLMs introduce several new challenges. Beyond just the quadratic computation cost of self-attention [183] with increasing context and large model sizes, LLMs also store intermediate Key (K) and Value (V) pairs for subsequent next word prediction. This K,V caching introduces additional memory related challenges as K,V cache size increases with increase in sequence length. The architecture of widely used LLMs like GPT [25] and LLaMa [176, 177] use Multi-Head Attention (MHA) [183]. MHA uses several attention heads to look at a sequence. As models grow bigger, the number of heads increases as well. For example, LLaMa-7B uses 32 attention heads in each layer, while LLaMa-65B uses 64 attention heads per layer [176]. The use of MHA exacerbates bottlenecks for serving LLMs. First, it increases compute pressure due to repeated application of the attention operation. Second, it increases the memory pressure due to requiring storage of Key (K), Value (V) caches that comes with the additional attention heads.

6.1.1 Inference in Decoder only Transformer

We first provide background on inference process for decoder only transformers like GPT [135, 25], LLaMa [176, 177] and the bottlenecks in performing inference. Further, we discussed several prior lines of work which have tried to tackle the inference bottlenecks for transformer based model.

A decoder-only transformer forms the building block of popular LLMs. A single decoder block consists of a self attention layer and a MLP. An input token is fed into the decoder block, to perform next-word prediction. The self attention block uses prior query (Q), key (K) and value (V) vectors associated with current token. These tokens are extracted by performing a linear projection with query, key and value weight matrices associated with a transformer.

To precisely define Multi-Head Attention (MHA), let H, T, d be positive integers, where H denotes number of heads, T denotes sequence length, d denotes model dimension. Let $x \in^{T\times d}$ be input to the MHA layer. For a single head h, then $\mathbf{K}^h = x\mathbf{W}_K^h$, $\mathbf{Q}^h = x\mathbf{W}_Q^h$ and $\mathbf{V}^h = x\mathbf{W}_V^h$ denote the corresponding key, query and value vector. The attention matrix for head h is calculated as follows:

$$A_h = \sigma(\frac{1}{\sqrt{d}} Q^h K^{hT})$$

Output of MHA is denoted by:

$$y = A_0V_0 \oplus A_1V_1 \oplus A_2V_2 \oplus \cdots \oplus A_HV_H$$

For performing inference, self attention needs access to the query, key and values associated with prior tokens. In order to avoid re-computation, inference serving systems cache the prior tokens in a sequence.

Compute cost required for multiple attention heads and memory capacity required for storing key and value vectors associated with each head during inference form two primary bottlenecks for LLM inference. In this work, we focus on reducing both memory and compute requirements via clustering multiple attention heads with similar output.

6.1.2 Related Work

Building Efficient Transformers. Improving efficiency of transformer models has been of major focus in recent years. Prior work can be broadly categorized in the following fields - (i) Hardware-software co-design [39, 38, 63, 64, 166, 49, 133, 190], (ii) Knowledge distillation [70, 85, 145, 194] (iii) Neural Architecture Search (NAS) [225, 94, 102] and (iv) Pruning [185, 114] and Quantization [51, 203, 91, 152**?** , 42, 43]. In this work our focus is on pruning , which we discuss next.

LLM Quantization. Recently several methods have been proposed to perform post training quantization allowing models to be quantized to a lower precision [51, 203, 43]. The goal of these methods is to perform quantization so as to minimize the error, CHAI is orthogonal to quantization based mechanisms as it depends on the insight of several attention heads focusing on the same tokens. The goal of quantization methods is to keep the same properties of original models, therefore we believe CHAI can be used to further accelerate post training quantized neural networks.

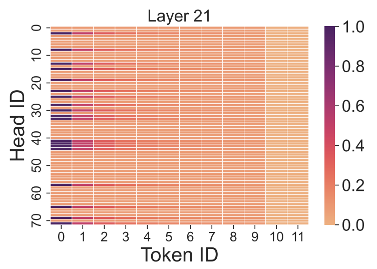
LLM Pruning. Pruning is a widely studied method to improve inference time by removing unused weights post training. Several prior works have looked at pruning for language models [29, 131, 27]. For example, oBERT is a second order method to reduce the number of weights [99]. Although these approaches can compress a model, they rarely yield inference speedups due to lack of hardware support for sparse operations on modern GPUs. To overcome the challenges, low rank decomposition methods [191, 189, 196], attention head pruning [120, 185], layer dropping [143, 48, 37] were proposed. However, these methods are infeasible for LLMs due to the use of iterative gradient calculations or fine-tuning leading to high resource requirements.

To overcome these issues, a recently proposed method, DejaVu [114], identifies portions of the model which are unused for a given context. To reduce the overhead of self-attention, DejaVu prunes attention heads which give *uniform weight across tokens*. We plot the activations for an exemplary sentence used by DejaVu for both OPT-66B and LLaMa-7B in Figure 6.1. We observe that while there are heads which give uniform weight to each token in OPT-66B model, there are no such heads in more parameter efficient models like LLaMa-7B, indicating that for smaller parameter efficient models like LLaMa DejaVu might not be applicable. The primary difference between OPT and LLaMa activation patterns could be attributed to the fact that LLaMa models are trained significantly longer and with more data.

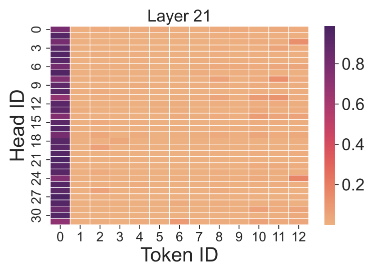
We observe that CHAI's insight about redundancy in the output of multiple heads in the attention holds across both OPT and LLaMa family of models. In our evaluation , we perform quantitative comparison between CHAI and DejaVu.

K,V Cache Compression. Prior works which have tried to reduce the K,V cache size [113, 218] by storing the K,V cache values for the most recent important tokens. However, they can not directly improve the latency of generating the next token, as they still perform the full transformer compute before finally deciding which K,V pairs should be stored. On the other hand, CHAI reduces not just the K,V cache size, it is also able to reduce the latency of next word prediction.

Speculative Decoding. Speculative decoding [104, 208, 202] is a popular method where a draft model is used to cheaply generate a sequence of draft tokens which can be efficiently verified by a target LLM. Speculative decoding can significantly reduce the latency of LLM serving, however it further exacerbates the compute and memory requirements as it requires additional resources to run both the draft and target model. CHAI on the



(a) **OPT-66B**: For several heads the activation scores are uniform, i.e. the heads given close to equal importance to each input token.



(b) **LLaMa-7B:** Heads in LLaMa-7B specifically pay attention to a specific token. However, multiple heads are attending to same token, in this case the first token.

Figure 6.1: **Activations for OPT-66B and LLaMa-7B for an exemplary sentence:** We observe that OPT-66B has several heads which give uniform attention scores to tokens whereas LLaMa-7B does not. However, both models have redundancies across heads, i.e. groups of heads are give similar attention to each token.

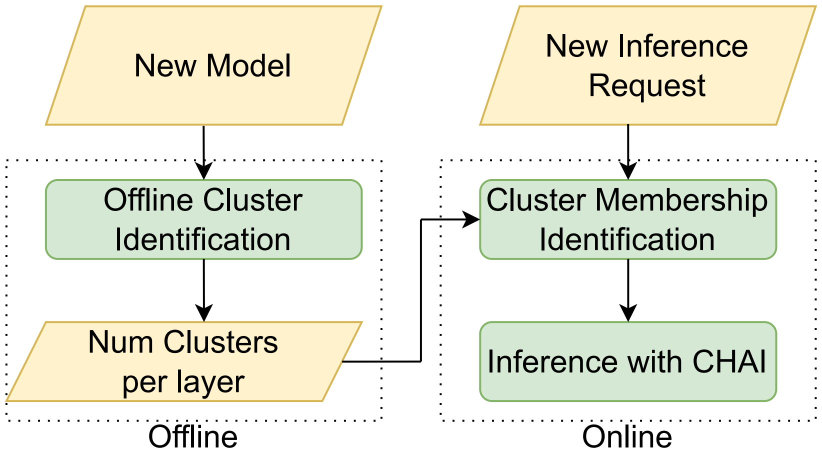
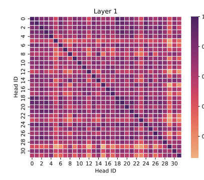
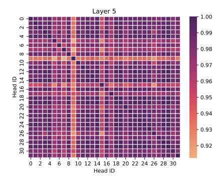
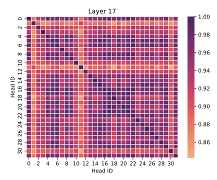


Figure 6.2: **CHAI Flow:** In the offline phase, we run clustering and perform elbow plot analysis for each new model. Then, for each new inference request we only perform cluster membership identification based on online performance.

other hand is focused on reducing the resource required for inference.


(a) Layer 1


(b) Layer 5


(c) Layer 17

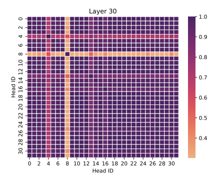
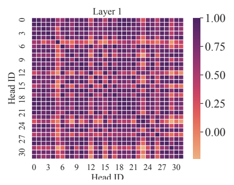
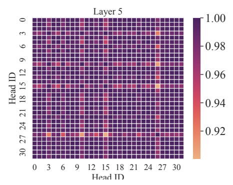
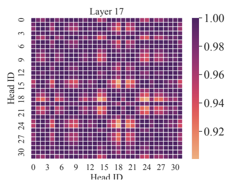

(d) Layer 30

Figure 6.3: **Average Correlation for 1024 Samples of C4 on LLaMa-7B:** The above figure shows two interesting observations. First, there exists high amount of correlation across several heads of attention. Second, the correlation is not uniform across layers, with later layers having higher correlation, i.e., first layer has very little correlation but correlation increases in later layers.


(a) Layer 1


(b) Layer 5


(c) Layer 17

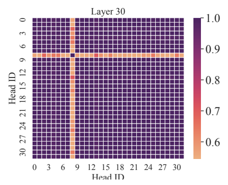

(d) Layer 30

Figure 6.4: **Correlation on a randomly selected single sample of LLaMa-7B.**

6.2 CHAI

Next, we describe CHAI. We first describe the key insights which have been used to build CHAI. Then, we detail CHAI's runtime pruning algorithm which is inspired by our insights and discuss how we perform inference using CHAI. Figure 6.2 provides a high level overview of inference using CHAI, which includes offline and online components.

6.2.1 Observations

Our primary insight stems from the observation that there is a high amount of correlation across the output of various attention heads in MHA, i.e. the output of several attention heads focuses on the same tokens. In Figure 6.3, we plot the average correlation across the 32 heads of LLaMa-7B for 1024 samples of the C4 [136] dataset for different layers and in Figure 6.4,

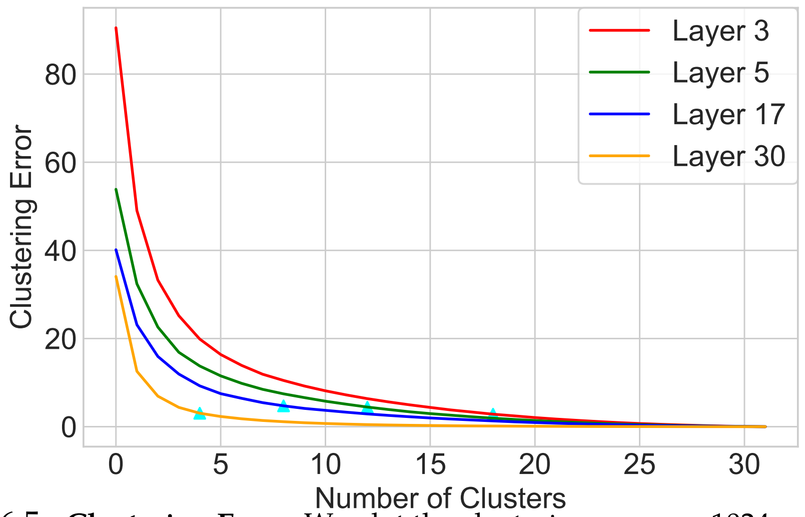


Figure 6.5: **Clustering Error:** We plot the clustering error on 1024 samples of C4-dataset. The markers represent the number of clusters we choose for a layer.

we plot correlation for a single sample of the dataset. These show us two insights - (i) Several heads output similar attention scores for each example and (ii) The amount of correlation increases in later layers, with heads in later layers with having higher correlation. This indicates that there is an opportunity to cluster attention heads with similar output and only run the self-attention operation for one of the representative attention heads within each cluster, thus reducing the amount of computation as well as the size of K,V cache.

Problem Formulation. Next, we formally define the problem of finding heads whose attention score is similar. Let H be the total number of attention heads, let $S = \{\langle K^1, Q^1\rangle, \langle K^2, Q^2\rangle, \langle K^3, Q^3\rangle, \cdot\cdot\cdot, \langle K^H, V^H\rangle\}$ be the set of Q, K pairs associated with each head h. Our goal is to find k subsets, $S_1 \subset S, S_2 \subset S, S_3 \subset S, \cdot\cdot\cdot S_k \subset S$ such that $< Q, K >$ pairs in each subset S_i produce similar output under function f. Where function f is the self attention operation, where $f(Q, K) = \sigma(QK^T)$. Further, we want $\cup_{i=1}^{k} S_i = S$.

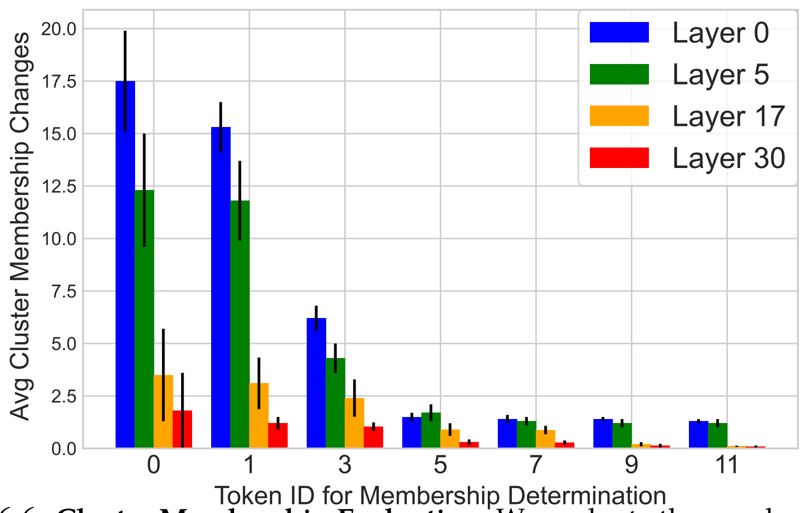


Figure 6.6: **Cluster Membership Evaluation:** We evaluate the number of times the cluster membership changes for performing next token prediction. We observed that if clustering is performed beyond the fifth token the number of times cluster membership changes is quite small.

Formally, we want to find S_i,

$$\forall < K^n, Q^n >, < K^m, Q^m > \in S_i,$$

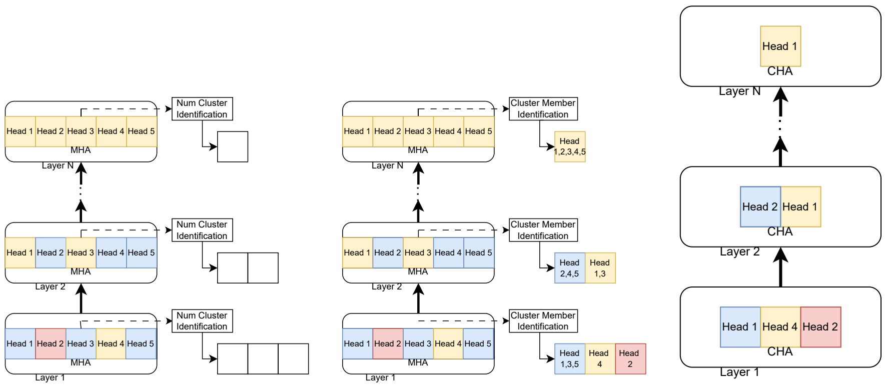
s.t.

$$f(K^n, Q^n) \approx f(K^m, Q^m)$$

Informally, we want subset of heads, where within each subset the self attention operation gives similar outcome.

In order to solve this problem we need to determine k which represents the number of such subsets, and the membership of such subset S_i. Our observations empirically demonstrate the existence of such a solution. We can potentially solve this problem using clustering, where determining the number of subsets translates to determining number of clusters and determining cluster membership becomes determination of cluster membership.

To observe memory and compute savings, we need an accurate and


Num Cluster Identification
Head 1 Head 2 Head 3 Head 4 Head 5
MHA
Layer N
Layer 2
Layer 1
Cluster Member Identification
Head 1,2,3,4,5
Head 2,4,5
Head 1,3
Head 1,3,5
Head 4
Head 2
CHA

(a) **Offline Cluster Identification:** For each new model we run an offline cluster identification phase. We collect the activations and perform Elbow-plot analysis to decide number of clusters.

(b) **Cluster Membership Identification:** For each new request, we initial run with multi-head attention for first five tokens. Using this we determine the number of clusters in each layer.

(c) **CHAI Inference:** Post cluster membership identification we substitute MHA with Clustered Head Attention.

Figure 6.7: **Schematic of CHAI** detailing three phases of the system.

efficient method to determine the number of clusters and their membership *without having access to activations*. Solving this forms a core contribution of our work.

6.2.2 Determination of Number of Clusters

Challenges. Figure 6.3 and Figure 6.4 indicate that the number of clusters varies widely per layer in a LLM. Specifically, the last few layers in the LLM exhibit a very low number of clusters (high redundancy), whereas the early layers demonstrate a high degree of variance across the output of heads resulting in large number of clusters. This observation suggests that the method used to determine number of clusters needs to make decisions for each layer independently. Additionally, widely used methods such as Elbow plot method [174] for determining number of clusters entail manual effort making cluster number determination impractical at inference time.

Design. To determine the number of clusters, we propose an offline strategy we run once for each model. In our case, we sample a small number of samples (1024) from the C4 [136] dataset and perform elbow-plot analysis by plotting clustering error (i.e. sum of squared distance from the closest cluster) as a function of number of clusters. Figure 6.5 shows the clustering error for LLaMa-7B for the samples selected. Based on the Elbow-plot analysis we choose the number of clusters when the error plateaus.

The offline analysis is performed once for each network by using the C4 [136] dataset. We do not change the number of clusters determined for a new dataset.

6.2.3 Determination of Cluster Membership

Challenges. Having determined number of clusters, we need to determine the membership of these clusters, i.e. which heads belong to which cluster in each layer. For Figure 6.3, 6.4 and 6.5, we perform clustering based on activations obtained by performing the forward pass. However, for each decoding step, performing clustering on output of self attention post forward pass will not yield any performance benefit as we will still be performing the original compute and using the full K,V cache. In order to utilize the insights observed in Section 6.2.1, we will need to decide the cluster members without having access to the output of the self attention.

Design. A simple strategy would have been keeping the cluster membership static across the tokens and independent of input context, e.g. we use the same cluster membership found during offline analysis with C4 data in the previous section. For evaluation purposes, we call this version of head selection **CHAI-static**.

However, we observed that the cluster membership does not remain static and varies based on context. When comparing Figure 6.4, which plots correlation for a single example, with Figure 6.3, which plots correlation for 1024 samples, we observe that the correlation across heads varies with varying context. Therefore, the correlation across the output of the heads depends on the context (input prompt), i.e. *a solution to determine the membership of each cluster has to account for context.* To understand the effects of accounting for context while clustering heads, we analysed the change in cluster membership changes and clustering with different context. In Figure 6.6, we observed an interesting phenomenon, after determining cluster membership by accounting for five tokens, the cluster membership does not change frequently. A direct outcome of this observation is that for each new sequence we can perform clustering based on the output of self-attention after the first five tokens. We observe that *activation from first five tokens of a new sequence are enough to accurately predict the cluster membership.* This dynamic version of head selection further allows us to improve accuracy over CHAI-static. Figure 6.7b shows an illustration of the membership identification step. Furthermore, evaluation results in Section 6.3 compare CHAI-static and CHAI performance.

6.2.4 Clustered Head Attention

Once we have decided which heads have similar attention output, we can than use Clustered Head Attention to combine key and query vectors for the heads.

6.2.5 Inference using CHAI

Next we, discuss the inference flow of CHAI, illustrated in detail in Figure 6.7. For each new model we first perform offline cluster identification (Figure 6.7a). Then for each new request, we determine the cluster membership using K-Means clustering once we have processed five tokens,

Table 6.1: Accuracy on OPT-66B

Method	PIQA	Hellaswag	Arc-Challenge	Arc-Easy	Boolq
MHA	78.4	71.1	41.6	64.7	65.4
DejaVu-50%	-0.25	-0.7	-0.6	-0.2	-4.0
CHAI-static	-1.35	-1.7	-0.7	-0.7	-0.7
CHAI	**-0.15**	**0.1**	**0.1**	**-0.1**	**-0.6**

using the observed activations (Figure 6.7b). After this step, we keep the clustered heads same throughout inference (Figure 6.7c).

There are two direct outcomes of CHAI's design. First, we directly reduce the amount of computation by removing redundant heads. Secondly, after a pre-determined token we fix the heads which are going to be pruned, this also allows us to remove the corresponding *Key* tokens associated, which significantly reduces the K,V cache size. Therefore, CHAI allows us to reduce both the inference compute as well as the size of the K,V cache required.

6.3 Evaluation

We experimentally verify the performance of CHAI and compare it to DejaVu [114] and SpAtten [190] on three different models of various sizes LLaMa-7B [176], LLaMa-33B and OPT-66B [216]. We evaluate the models on five commonly used NLP tasks: PIQA [22], HellaSwag [214], Arc-Challenge and Arc-Easy [33] and BoolQA [32].

6.3.1 Experimental Setup

All our experiments are performed on servers with NVIDIA V100 GPUs. For OPT-66B we used eight GPUs on a single node, for LLaMa-33B we used four GPUs, and for LLaMa-7B, we used a single GPU for inference. CHAI is built on top of Meta's xFormers [47].

6.3.2 Accuracy Evaluation

In our evaluation, we compare CHAI with Multi-Head Attention as baseline, static version of CHAI, as well two other state-of-the-art prior pruning methods; DejaVu and SpAtten. For DejaVu, we try different sparsity ratios, in order to try to match the accuracy number to MHA. We also compare CHAI to SpAtten, a method which removes unimportant tokens and heads.

In Table 6.1, we first verify that we are able to reproduce the performance numbers reported by DejaVu. To perform this, we took the OPT-66B and evaluated both DejaVu, CHAI and CHAI-static. We used DejaVu with 50% sparsity as reported by the authors. We used the author provided code to train their MLP predictor layers and incorporate their scheme in our setup. In Table 6.1, we observe that we were able to replicate results for OPT-66B. Furthermore, CHAI is also able to match the accuracy of MHA for OPT-66B.

Next, we compare CHAI, CHAI-static and DejaVu with the pre-trained MHA network, using LLaMa-7B on 5 different datasets. For DejaVu we used three configurations, 50% sparsity, 30% sparsity and 10% sparsity. In Table 6.2, we observe that when we use DejaVu with more 10% sparsity we see significant decrease in accuracy (by 18.6% for DejaVu-30%). On the other hand, our method based on our close analysis of the behaviour of layers of LLaMa-7B is able to recover accuracy. We observe a maximum accuracy degradation of 3.7% for CHAI. Similarly for LLaMa-33B using sparsity for more than 10% leads to significant accuracy drop, meanwhile CHAI closely matches the accuracy of the pre-trained model using MHA with maximum degradation in accuracy by 0.14%. This shows that CHAI is widely applicable across multiple datasets and models. We also want to highlight that we do not perform any dataset specific tuning.

Table 6.2: Accuracy on LLaMa-7B

Method	PIQA	HellaSwag	Arc-Challenge	Arc-Easy	BoolQ
MHA	79.8	76.1	47.5	72.8	76.0
DejaVu-10%	-3.9	-4.7	-5.78	-3.18	-7.4
DejaVu-30%	-13.3	-18.6	-18.75	-4.2	-20.2
DejaVu-50%	-24.6	-50.7	-19.35	-46.3	-21.6
SpAtten	-41.4	-42.5	-18.0	-40.2	-27.1
CHAI-static	-4.0	-4.3	-3.7	-2.5	-0.8
CHAI	**-2.0**	**-3.2**	**-0.5**	**0.3**	**0.1**

Table 6.3: Accuracy on LLaMa-33B

Method	PIQA	HellaSwag	Arc-Challenge	Arc-Easy	BoolQ
MHA	82.1	82.8	57.8	80.0	83.1
DejaVu-10%	-0.7	0.1	**-0.2**	-0.6	-0.2
DejaVu-30%	-9.3	-24.4	-17.91	-12.4	-12.2
DejaVu-50%	-27.6	-43.2	-24.6	-37.6	-21.2
SpAtten	-31.9	-44.1	-26.4	-40.3	-34.55
CHAI-static	-0.5	-0.2	-1.3	-3.7	-1.5
CHAI	**0**	**-0.14**	-0.21	**0.9**	**-0.04**

6.3.3 Memory Capacity Evaluation

CHAI reduces memory capacity requirements due to reduction in K,V cache size and minimal additional storage required to store the cluster map. In Figure 6.8, we show that for LLaMa-7B CHAI reduces the size of K,V cache by up to 21.4% compared to MHA. Even for comparatively small models like LLaMa-7B, the size of the K,V cache for a sequence length of 2048 is around 1.2 GB, while around 12 GB is used for the model weights. A reduction in K,V cache size can enable use of larger context length or serving more requests. We would also like to note that CHAI only removes the keys associated with redundant heads and keeps all the

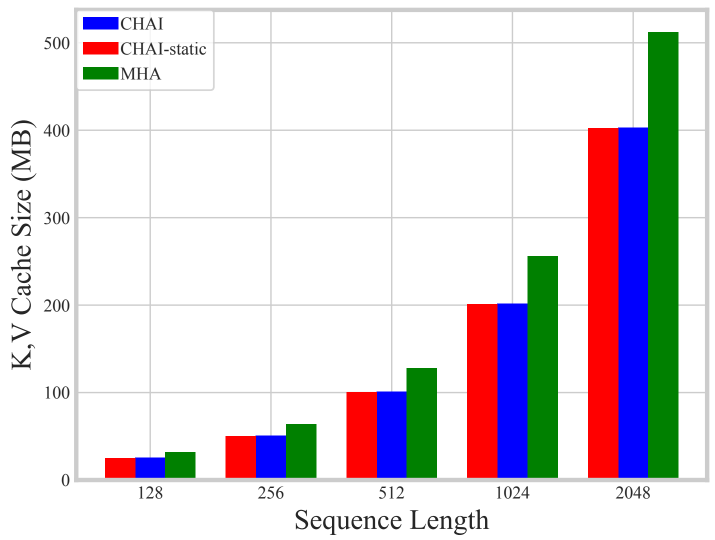


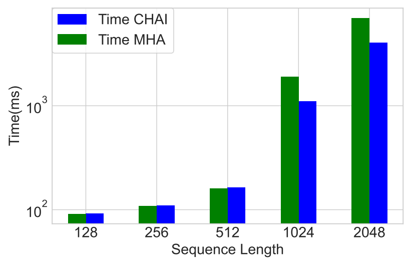
Figure 6.8: **Memory Savings**: We observed that for LLaMa-7B CHAI provides memory savings of up to 21.4%.

value vectors.

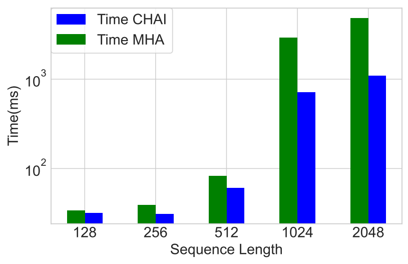
Memory overhead of CHAI. The only additional storage CHAI requires is storing the cluster map. Size of the map can be determined by $n\_layers \times (n\_heads + n\_clusters)$, for LLama7B the number of layers is 32, and number of heads is 32 and number of clusters vary from 28 in early layers to 4 in most of the later layers. Thus storing this map only requires a few hundreds of bytes. We would like to point out that similar methods like DejaVu require training MLP classifiers per layer to learn the sparsity pattern which need to be stored.

6.3.4 End-to-End Latency Evaluation

Next, we evaluate time to first token and time to next token comparing it with MHA. These are two standard metrics used for evaluation of an LLM. Time to first token evaluates the time for generating a first token given a new context. Time to first token accounts for generating K,V caches for all



(a) **Time to first token:** We observe speedups of up to 1.73× for sequence length of 2048.



(b) **Time to next token:** We observe a speedup of up to 5× for sequence length of 2048.

Figure 6.9: **Latency Analysis:** We observe that the speedups provided by CHAI increases as the sequence length becomes larger. Even for a comparatively small model like LLaMa-7B we observe speedups of up to 1.73× for a large sequence length.

the tokens in the context. Whereas time to next token evaluates the time for generating the next token, assuming the K,V caches for all internal tokens is available.

Time to first token. Next, in our experiments we compare the speedups provided by CHAI. In Figure 6.9-(a) for LLaMa-7B we show that our method provides speedup of up to 1.72× on a sequence length of 2048. The execution times represented in this figure accounts for the overhead of clustering in CHAI.

Time to next token. Another metric for evaluation of LLMs is time to next token. We do not account for the overhead of clustering in the case of time to next token. Our primary wins come from reducing compute and reducing memory bandwidth requirement for performing time to next token. Figure 6.9-(b) shows time to predict the next token for different sequence lengths. We observe that CHAI provides a speedup of over 5× for a sequence length of 2048.

Unfortunately, we are not able to compare times with DejaVu as the authors have not released the specialized kernels used for realizing the speedups on hardware [3], thus inhibiting a runtime comparison. However, we believe it is unlikely that at less than 10% sparsity which is needed by DejaVu to get comparable accuracy to MHA, it will yield high speedups [68]. We would like to highlight that because of performing dense computations, unlike DejaVu, CHAI does not need custom GPU kernels. Further, CHAI's speedup benefits are independent of the framework used, because irrespective of implementation, CHAI directly reduces the complexity of MHA.

Compute overhead of CHAI. Existing efficiency enhancing methods require some fine-tuning or modification to the architecture. While DejaVu [114] and SpAtten [190] are runtime methods they still require additional compute, e.g. DejaVu requires running MLP classifiers during inference. CHAI on the other hand only adds the computation overhead for determining the cluster membership. In our experiments we observed that clustering takes only 0.6 ms per request, which is about 0.008% of the inference latency.

6.3.5 Additional Experiments

Next we perform additional studies on our algorithm.

CHAI with Quantization. Next, we run experiments to understand how CHAI performs in conjunction with quantization. To perform this experiment we take an open source GPTQ [51] quantized model from HuggingFace [173] and run CHAI on the model. In Table 6.4, we show the performance of CHAI on a LLaMa-7B on the 4-bit quantized model. We observe that the maximum deviation from accuracy for the quantized model is 0.1%.

CHAI with Grouped Query Attention.

Grouped Query Attention [14] is a widely used method which shares single Key and Value vectors across multiple queries. This effectively reduces the K,V cache size. We perform preliminary studies to understand

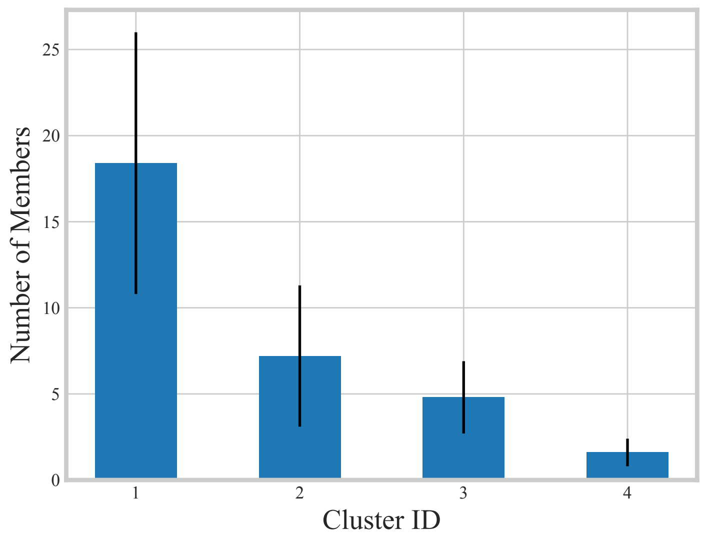


Figure 6.10: **Cluster Distribution**: We observe that number of heads within the cluster is quite skewed. We often observe one or two large clusters, while the remaining heads in the cluster.

Table 6.4: CHAI with Quantization

	HellaSwag	PIQA	BoolQ	Arc-Challenge	Arc-Easy
LLama-7B	76.1	79.8	76.0	47.5	72.8
LLama-7B-4bit-GPTQ	67.97	71.34	66.73	41.23	64.8
CHAI-LLama-7B-4bit-GPTQ	67.91	72.05	66.48	41.19	64.7

how CHAI can be used in conjunction with GQA. For these experiments we used models like LLama 2 - 70B that are pre-trained with GQA, we ran some preliminary experiments where we used CHAI to further reduce the number of K,Q pairs in grouped query attention. LLama-70B uses grouping factor of 8, i.e., 8 queries map to single key vector, we were able to reduce the number of K vectors on average by 1.8x and Q vectors by 3.7x. In Table 6.5. We show this reduction leads to negligible degradation in accuracy. This indicates that it is possible to use CHAI with GQA.

Pruning K, Q and V. In CHAI, we prune only the Key and Query portion of an attention head leaving the Value vector intact. Next, we study how accuracy changes if we remove the value vector as well. To perform this

Table 6.5: CHAI with GQA

	Hellaswag	PIQA	BoolQ
LLama2-70B-GQA	85.3	82.8	85.0
CHAI-LLama2-70B-GQA	85.2	82.73	85.0

Table 6.6: Pruning Both Q,K,V

	CHAI	CHAI-QKV	MHA
Arc-Challenge	47.0	41.29	47.5
PIQA	77.8	61.93	79.8

experiment we chose to reuse the value vector generated by the chosen head. In Table 6.6, we show how reusing the full head (Query, Key and Value vector) lead to additional loss in accuracy. This shows that for smaller networks like LLaMa it might be hard to remove the whole head in Multi-Head Attention.

Cluster Distribution. Figure 6.10 shows the distribution across clusters for Layer-18 on LLaMa-7B for different 1024 samples of C4 dataset. We observe that typically for LLMs majority of heads can be grouped into a single head.

6.4 Conclusion

In this work, we present CHAI, an efficient runtime method which identifies attention heads giving similar scores. Using this method we reduce overhead of Multi-Head Attention by clustering the correlated heads and computing attention scores only for heads which lead to disparate attention scores. Our evaluation shows that with minor accuracy loss system can speedup inference by up to 1.73×.

Part V

Conclusion and Future Work

7 CONCLUSION

This chapter reviews key contributions made in this dissertation. We first highlight the key problems studied in this dissertation. Next, we elucidate our findings and insights derived from our work. Next, we highlight learning from works presented in this dissertation and speculate how it can guide future works.

7.1 Summary and Contributions

The primary goal of this dissertation has been to study how data movement and synchronization bottlenecks lead to degraded performance. In this dissertation we looked at Gradient Compression in Part II. We studied how gradient compression can be further improved by varying the amount of compression during training. Next, we study several existing gradient compression methods. We observe that a majority of gradient compression methods do not provide any wall clock speedups despite massive reduction in amount of communication. We also provide a performance model which can help designers to come up with a compression scheme for a new deployment environment.

In Part III we propose Bagpipe, a large scale recommendation model training system. A key challenge in prior recommendation model training systems has been the remote embedding access overhead. Bagpipe uses the idea of lookahead and where it can look beyond the current training and figure out embedding access patterns. Based on this, it can overlap embedding access overheads with remote accesses. In Bagpipe we also introduce Logically Replicated and Physically Partitioned Caches, such caches from a logical point of view are replicated but have different contents in them. This allows us to minimize the control overhead as we do not have to care about partitioning and also reduces the amount of communication.

Finally in Part IV we study multi-head attention. Multi-head attention is a crucial component of Large Language Model. However, multi-head attention is heavily memory bandwidth bottleneck. We show that several heads in multi-head attention often produce redundant activations. Based on this observation we introduce Clustered Head Attention, which determines the redundant heads and clusters them to reduce both compute and memory bandwidth requirement.

7.2 Learnings

The projects presented in this dissertation highlight a few underlying themes which can be used to tackle problems in the area of machine learning systems. With Accordion [9], we observed that when building approximation algorithms we can further improve them by accounting for neural network training dynamics. Based on the insights in Accordion we have worked on additional gradient compression algorithms like in Pufferfish [189]. Next, the utility of gradient compression highlights that when working on system optimizations, we need to also focus on how it effects other parts of the application. We observed that several Gradient Compression algorithms by compressing gradients were actually making it incompatible with efficient communication primitives leading to suboptimal performance. When building new optimization we should always be mindful of how those optimizations can effect compatibility with other components of the application.

With Bagpipe [11] we highlight that in ML Systems there are often avenues to extract additional information. Using this additional information we can reformulate the data flow to allow us to efficiently overlap communication with computation thus improve hardware utilization.

We believe that these findings with minor modification can be used with newer upcoming models. For ex, we are using an extension of lookahead to improve state management while serving LLMs.

BIBLIOGRAPHY

[1] Massively scale your deep learning training with nccl 2.4. https://bit.ly/341nGfs. Accessed: August 31, 2023.

[2] tc - show / manipulate traffic control settings. https://man7.org/linux/man-pages/man8/tc.8.html, 2020. Accessed: May 25, 2021.

[3] Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. https://github.com/FMInference/DejaVu, 2024.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[5] J. Acharya, C. De Sa, D. Foster, and K. Sridharan. Distributed learning with sublinear communication. In *International Conference on Machine Learning*, pages 40–50. PMLR, 2019.

[6] A. Achille, M. Rovere, and S. Soatto. Critical learning periods in deep networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BkeStsCcKQ.

[7] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814. IEEE, 2021.

[8] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair. High-performance training by exploiting hot-embeddings in recommendation systems. *arXiv preprint arXiv:2103.00686*, 2021.

[9] S. Agarwal, H. Wang, K. Lee, S. Venkataraman, and D. Papailiopoulos. Adaptive gradient communication via critical learning regime identification. *Proceedings of Machine Learning and Systems*, 3:55–80, 2021.

[10] S. Agarwal, H. Wang, S. Venkataraman, and D. Papailiopoulos. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

[11] S. Agarwal, Z. Zhang, and S. Venkataraman. Bagpipe: Accelerating deep recommendation model training. *arXiv preprint arXiv:2202.12429*, 2022.

[12] S. Agarwal, B. Acun, B. Homer, M. Elhoushi, Y. Lee, S. Venkataraman, D. Papailiopoulos, and C.-J. Wu. Chai: Clustered head attention for efficient llm inference. *arXiv preprint arXiv:2403.08058*, 2024.

[13] S. Agarwal, A. Phanishayee, and S. Venkataraman. Blox: A modular toolkit for deep learning schedulers. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1093–1109, 2024.

[14] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[15] A. F. Aji and K. Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[16] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[17] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli. The convergence of sparsified gradient methods. In *NeurIPS*, 2018.

[18] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, 2021.

[19] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communication library (intercom). In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 357–364. IEEE, 1994.

[20] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

[21] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar. signsgd with majority vote is communication efficient and fault tolerant. In *International Conference on Learning Representations*, 2018.

[22] Y. Bisk, R. Zellers, J. Gao, Y. Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[23] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, H. B. McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 18*), pages 578–594, 2018.

[27] T. Chen, J. Frankle, S. Chang, S. Liu, Y. Zhang, Z. Wang, and M. Carbin. The lottery ticket hypothesis for pre-trained bert networks. *Advances in neural information processing systems*, 33:15834–15846, 2020.

[28] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, X.-H. Sun, and G. Chen. icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 220–232. IEEE, 2023.

[29] X. Chen, Y. Cheng, S. Wang, Z. Gan, Z. Wang, and J. Liu. Earlybert: Efficient bert training via early-bird lottery tickets. *arXiv preprint arXiv:2101.00063*, 2020.

[30] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[31] M. Cho, V. Muthusamy, B. Nemanich, and R. Puri. Gradzip: Gradient compression using alternating matrix factorization for large-scale deep learning.

[32] C. Clark, K. Lee, M.-W. Chang, T. Kwiatkowski, M. Collins, and K. Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[33] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[34] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Operating Systems Review*, 53(1):14–25, 2019.

[35] P. Covington, J. Adams, and E. Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[36] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

[37] S. Dai, H. Genc, R. Venkatesan, and B. Khailany. Efficient transformer inference with statically structured sparse attention. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[38] T. Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[39] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[40] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[42] T. Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

[43] T. Dettmers and L. Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR, 2023.

[44] A. Devarakonda, M. Naumov, and M. Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[45] N. Dryden, T. Moon, S. A. Jacobs, and B. Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[46] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram. {Check-N-Run}: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[47] facebookresearch. xformers - toolbox to accelerate research on transformers. `https://github.com/facebookresearch/xformers`, 2023. Accessed: December 12, 2023.

[48] A. Fan, E. Grave, and A. Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[49] C. Fang, A. Zhou, and Z. Wang. An algorithm–hardware co-optimized framework for accelerating n: M sparse transformers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30 (11):1573–1586, 2022.

[50] J. Fang, H. Fu, G. Yang, and C.-J. Hsieh. Redsync: reducing synchronization bandwidth for distributed deep learning training system. *Journal of Parallel and Distributed Computing*, 133:30–39, 2019.

[51] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[52] R. M. French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[53] V. Gandikota, D. Kane, R. K. Maity, and A. Mazumdar. vqsgd: Vector quantized stochastic gradient descent. In *International Conference on Artificial Intelligence and Statistics*, pages 2197–2205. PMLR, 2021.

[54] A. Golden, S. Hsia, F. Sun, B. Acun, B. Hosmer, Y. Lee, Z. DeVito, J. Johnson, G.-Y. Wei, D. Brooks, et al. Generative ai beyond llms: system implications of multi-modal generation. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 257–267. IEEE, 2024.

[55] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[56] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022.

[57] D. Grubic, L. Tam, D. Alistarh, and C. Zhang. Synchronous multi-GPU deep learning with low-precision communication: An experimental study. 2018.

[58] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: A factorization-machine based neural network for ctr prediction. In *IJCAI*, 2017.

[59] J. Guo, W. Liu, W. Wang, J. Han, R. Li, Y. Lu, and S. Hu. Accelerating distributed deep learning by adaptive gradient quantization. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1603–1607, 2020.

[60] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.

[61] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[62] gurobi. Gurobi optimization. `https://www.gurobi.com/`.

[63] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, et al. A^ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[64] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 692–705. IEEE, 2021.

[65] hetu. Source code for het. `https://github.com/Hsword/Hetu/tree/ef1959`.

[66] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[67] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[68] S. Hooker. The hardware lottery. *Communications of the ACM*, 64 (12):58–65, 2021.

[69] S. Horvath, C.-Y. Ho, L. Horvath, A. N. Sahu, M. Canini, and P. Richtarik. Natural compression for distributed deep learning. *arXiv preprint arXiv:1905.10988*, 2019.

[70] C.-Y. Hsieh, C.-L. Li, C.-K. Yeh, H. Nakhost, Y. Fujii, A. Ratner, R. Krishna, C.-Y. Lee, and T. Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

[71] C.-C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[72] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[73] Y. Huang, X. Wei, X. Wang, J. Yang, B.-Y. Su, S. Bharuka, D. Choudhary, Z. Jiang, H. Zheng, and J. Langman. Hierarchical training: Scaling deep recommendation models on large cpu clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 3050–3058, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. doi: 10.1145/3447548.3467084. URL `https://doi.org/10.1145/3447548.3467084`.

[74] hugectr. NVIDIA Merlin HugeCTR Framework. `https://developer.nvidia.com/nvidia-merlin/hugectr`.

[75] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[76] A. Inc. Avazu click logs. `https://www.kaggle.com/c/avazu-ctr-prediction/data`, 2013. Accessed: December 10, 2020.

[77] A. Isenko, R. Mayer, J. Jedele, and H.-A. Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1825–1839, 2022.

[78] T. Ishkhanov, M. Naumov, X. Chen, Y. Zhu, Y. Zhong, A. G. Azzolini, C. Sun, F. Jiang, A. Malevich, and L. Xiong. Time-based sequence model for personalization and recommendation systems. *arXiv preprint arXiv:2008.11922*, 2020.

[79] N. Ivkin, D. Rothchild, E. Ullah, I. Stoica, R. Arora, et al. Communication-efficient distributed sgd with sketching. In *NeurIPS*, 2019.

[80] S. Jastrzebski, Z. Kenton, N. Ballas, A. Fischer, Y. Bengio, and A. Storkey. On the relation between the sharpest directions of DNN loss and the SGD step length. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=SkgEaj05t7`.

[81] S. Jastrzebski, M. Szymczak, S. Fort, D. Arpit, J. Tabor, K. Cho*, and K. Geras*. The break-even point on optimization trajectories of deep neural networks. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=r1g87C4KwB`.

[82] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference* (*USENIX ATC 19*), pages 947–960, 2019.

[83] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018.

[84] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[85] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[86] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[87] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[88] S. P. Karimireddy, Q. Rebjock, S. Stich, and M. Jaggi. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pages 3252–3261, 2019.

[89] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[90] R. I. S. Khan, A. H. Yazdani, Y. Fu, A. K. Paul, B. Ji, X. Jian, Y. Cheng, and A. R. Butt. {SHADE}: Enable fundamental cacheability for distributed deep learning training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 135–152, 2023.

[91] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pages 5506–5518. PMLR, 2021.

[92] T. Kim, H. Kim, G.-I. Yu, and B.-G. Chun. Bpipe: memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*, pages 16639–16653. PMLR, 2023.

[93] G. Kingsley Zipf. *Selected studies of the principle of relative frequency in language*. Harvard university press, 1932.

[94] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[95] A. Koloskova, T. Lin, S. U. Stich, and M. Jaggi. Decentralized deep learning with arbitrary communication compression. *arXiv preprint arXiv:1907.09356*, 2019.

[96] A. Koloskova, S. Stich, and M. Jaggi. Decentralized stochastic optimization and gossip algorithms with compressed communication. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3478–3487. PMLR, 09–15 Jun 2019. URL `http://proceedings.mlr.press/v97/koloskova19a.html`.

[97] A. V. Kumar and M. Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, 2020.

[98] D. Kumaran, D. Hassabis, and J. L. McClelland. What learning systems do intelligent agents need? complementary learning systems theory updated. *Trends in cognitive sciences*, 20(7):512–534, 2016.

[99] E. Kurtic, D. Campos, T. Nguyen, E. Frantar, M. Kurtz, B. Fineran, M. Goin, and D. Alistarh. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*, 2022.

[100] C. Labs. Criteo kaggle logs. `https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/`, 2013. Accessed: December 10, 2020.

[101] C. Labs. Terabyte click logs. `https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/`, 2013. Accessed: December 10, 2020.

[102] F. Lagunas, E. Charlaix, V. Sanh, and A. M. Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

[103] J. Lee, S. Abu-El-Haija, B. Varadarajan, and A. Natsev. Collaborative deep metric learning for video understanding. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 481–490, 2018.

[104] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[105] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[106] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188. IEEE, 2018.

[107] Z. Li, H. Zhao, Q. Liu, Z. Huang, T. Mei, and E. Chen. Learning from history and present: Next-item recommendation via discriminatively exploiting user behaviors. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1734–1743, 2018.

[108] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1705.09056*, 2017.

[109] X. Lian, B. Yuan, X. Zhu, Y. Wang, Y. He, H. Wu, L. Sun, H. Lyu, C. Liu, X. Dong, et al. Persia: A hybrid system scaling deep learning based recommenders up to 100 trillion parameters. *arXiv preprint arXiv:2111.05897*, 2021.

[110] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[111] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[112] H. Ling, K. Kreis, D. Li, S. W. Kim, A. Torralba, and S. Fidler. Editgan: High-precision semantic image editing. *Advances in Neural Information Processing Systems*, 34:16331–16345, 2021.

[113] Z. Liu, A. Desai, F. Liao, W. Wang, V. Xie, Z. Xu, A. Kyrillidis, and A. Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *arXiv preprint arXiv:2305.17118*, 2023.

[114] Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[115] A. M Abdelmoniem, A. Elzanaty, M.-S. Alouini, and M. Canini. An efficient statistical-based gradient compression technique for distributed training systems. *Proceedings of Machine Learning and Systems*, 3, 2021.

[116] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[117] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, et al. Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2: 336–349, 2020.

[118] Meta. Torchrec. `https://github.com/pytorch/torchrec/`, 2022. Accessed: August 21, 2022.

[119] X. Miao, H. Zhang, Y. Shi, X. Nie, Z. Yang, Y. Tao, and B. Cui. Het: Scaling out huge embedding model training via cache-enabled distributed framework. *Proc. VLDB Endow.*, 15(2):312–320, 2022.

[120] P. Michel, O. Levy, and G. Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.

[121] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[122] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama, et al. Massively distributed sgd: Imagenet/resnet-50 training in a flash. *arXiv preprint arXiv:1811.05233*, 2018.

[123] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram. Analyzing and mitigating data stalls in dnn training. *arXiv preprint arXiv:2007.06775*, 2020.

[124] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 533–549, 2021.

[125] mpi. Mpi all to all. `https://www.rookiehpc.com/mpi/docs/mpi_alltoall.php`, 2019. Accessed: December 12, 2022.

[126] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. *arXiv preprint arXiv:2104.05158*, 2021.

[127] D. G. Murray, J. Simsa, A. Klimovic, and I. Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

[128] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[129] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[130] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[131] S. Prasanna, A. Rogers, and A. Rumshisky. When bert plays the lottery, all tickets are winning. *arXiv preprint arXiv:2005.00561*, 2020.

[132] H. Qi, E. R. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations*, 2017.

[133] Y. Qin, Y. Wang, D. Deng, Z. Zhao, X. Yang, L. Liu, S. Wei, Y. Hu, and S. Yin. Fact: Ffn-attention co-optimized transformer architecture with eager correlation prediction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, 2023.

[134] R. Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science*, pages 1–9. Springer, 2004.

[135] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[136] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[137] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-shot text-to-image generation, 2021.

[138] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. 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*, pages 3505–3506, 2020.

[139] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[140] F. Research. Facebook research dlrm. `https://github.com/facebookresearch/dlrm/issues/206`, 2019. Accessed: December 10, 2021.

[141] F. Research. Facebook research dlrm. `https://github.com/facebookresearch/dlrm`, 2019. Accessed: December 10, 2021.

[142] F. Research. Fbgemm. `http://github.com/facebookresearch/fbgemm`, 2019. Accessed: August 31, 2023.

[143] H. Sajjad, F. Dalvi, N. Durrani, and P. Nakov. On the effect of dropping layers of pre-trained transformer models. *Computer Speech & Language*, 77:101429, 2023.

[144] P. Sanders, J. Speck, and J. L. Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[145] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[146] S. Sarvotham, R. Riedi, and R. Baraniuk. Connection-level analysis and modeling of network traffic. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 99–103, 2001.

[147] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems*, 31(9):3400–3413, 2019.

[148] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek. Sparse binary compression: Towards distributed deep learning with minimal communication. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[149] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[150] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[151] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[152] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020.

[153] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[154] S. Shi, X. Chu, K. C. Cheung, and S. See. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772*, 2019.

[155] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu. A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2238–2247. IEEE, 2019.

[156] S. Shi, X. Zhou, S. Song, X. Wang, Z. Zhu, X. Huang, X. Jiang, F. Zhou, Z. Guo, L. Xie, et al. Towards scalable distributed training of deep learning on public cloud clusters. *Proceedings of Machine Learning and Systems*, 3, 2021.

[157] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[158] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[159] Y. Song and D. P. Kingma. How to train your energy-based models. *arXiv preprint arXiv:2101.03288*, 2021.

[160] S. U. Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

[161] S. U. Stich and S. P. Karimireddy. The error-feedback framework: Better rates for sgd with delayed gradients and compressed communication. *arXiv preprint arXiv:1909.05350*, 2019.

[162] S. U. Stich, J.-B. Cordonnier, and M. Jaggi. Sparsified sgd with memory. In *Advances in Neural Information Processing Systems*, pages 4447–4458, 2018.

[163] N. Strom. Scalable distributed DNN training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[164] C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune bert for text classification? In *China National Conference on Chinese Computational Linguistics*, pages 194–206. Springer, 2019.

[165] A. T. Suresh, X. Y. Felix, S. Kumar, and H. B. McMahan. Distributed mean estimation with limited communication. In *International Conference on Machine Learning*, pages 3329–3337. PMLR, 2017.

[166] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, et al. Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 830–844, 2021.

[167] Y. K. Tan, X. Xu, and Y. Liu. Improved recurrent neural networks for session-based recommendations. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 17–22, 2016.

[168] H. Tang, S. Gan, C. Zhang, T. Zhang, and J. Liu. Communication compression for decentralized training. In *NeurIPS*, 2018.

[169] H. Tang, X. Lian, M. Yan, C. Zhang, and J. Liu. d^2: Decentralized training over decentralized data. In *International Conference on Machine Learning*, pages 4848–4856. PMLR, 2018.

[170] H. Tang, C. Yu, X. Lian, T. Zhang, and J. Liu. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pages 6155–6165. PMLR, 2019.

[171] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[172] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[173] TheBloke. Llama-7b gptq. `https://huggingface.co/TheBloke/LLaMa-7B-GPTQ`, 2023. Accessed: December 12, 2023.

[174] R. L. Thorndike. Who belongs in the family? *Psychometrika*, 18(4): 267–276, 1953.

[175] P. Tillet, H.-T. Kung, and D. Cox. 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*, pages 10–19, 2019.

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

[177] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[178] T. X. Tuan and T. M. Phuong. 3d convolutional networks for session-based recommendation with content features. In *Proceedings of the eleventh ACM conference on recommender systems*, pages 138–146, 2017.

[179] B. Twardowski. Modelling contextual information in session-aware recommender systems with neural networks. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 273–276, 2016.

[180] Y. Ueno and R. Yokota. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-GRID)*, pages 430–439, 2019. doi: 10.1109/CCGRID.2019.00057.

[181] T. Um, B. Oh, B. Seo, M. Kweun, G. Kim, and W.-Y. Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proceedings of the VLDB Endowment*, 16(5): 1086–1099, 2023.

[182] A. Van Den Oord, S. Dieleman, and B. Schrauwen. Deep content-based music recommendation. In *Neural Information Processing Systems Conference (NIPS 2013)*, volume 26. Neural Information Processing Systems Foundation (NIPS), 2013.

[183] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[184] T. Vogels, S. P. Karimireddy, and M. Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, pages 14236–14245, 2019.

[185] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *arXiv preprint arXiv:1905.09418*, 2019.

[186] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 144–161, 2023.

[187] H. Wang, S. Sievert, S. Liu, Z. Charles, D. Papailiopoulos, and S. Wright. Atomo: Communication-efficient learning via atomic sparsification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9850–9861. Curran Associates, Inc., 2018.

[188] H. Wang, K. Sreenivasan, S. Rajput, H. Vishwakarma, S. Agarwal, J.-y. Sohn, K. Lee, and D. Papailiopoulos. Attack of the tails: Yes, you really can backdoor federated learning. *Advances in Neural Information Processing Systems*, 33:16070–16084, 2020.

[189] H. Wang, S. Agarwal, and D. Papailiopoulos. Pufferfish: Communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems*, 3, 2021.

[190] H. Wang, Z. Zhang, and S. Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[191] H. Wang, S. Agarwal, Y. Tanaka, E. Xing, D. Papailiopoulos, et al. Cuttlefish: Low-rank model training without all the tuning. *Proceedings of Machine Learning and Systems*, 5, 2023.

[192] R. Wang, B. Fu, G. Fu, and M. Wang. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*, pages 1–7. 2017.

[193] S. Wang, Y. Wang, J. Tang, K. Shu, S. Ranganath, and H. Liu. What your images reveal: Exploiting visual contents for point-of-interest recommendation. In *Proceedings of the 26th international conference on world wide web*, pages 391–400, 2017.

[194] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[195] X. Wang and Y. Wang. Improving content-based and hybrid music recommendation using deep learning. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 627–636, 2014.

[196] Z. Wang, J. Wohlwend, and T. Lei. Structured pruning of large language models. *arXiv preprint arXiv:1910.04732*, 2019.

[197] J. Wangni, J. Wang, J. Liu, and T. Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, pages 1299–1309, 2018.

[198] J. Wen, J. She, X. Li, and H. Mao. Visual background recommendation for dance performances using deep matrix factorization. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 14(1):1–19, 2018.

[199] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[200] L. Wesolowski, B. Acun, V. Andrei, A. Aziz, G. Dankel, C. Gregg, X. Meng, C. Meurillon, D. Sheahan, L. Tian, et al. Datacenter-scale analysis and optimization of gpu machine learning workloads. *IEEE Micro*, 41(5):101–112, 2021.

[201] J. Wu, W. Huang, J. Huang, and T. Zhang. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*, pages 5325–5333. PMLR, 2018.

[202] H. Xia, T. Ge, P. Wang, S.-Q. Chen, F. Wei, and Z. Sui. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3909–3925, 2023.

[203] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[204] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation, 2020. URL `http://hdl.handle.net/10754/662495`.

[205] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020.

[206] C.-Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*, pages 366–367. IEEE Computer Society, 1988.

[207] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[208] S. Yang, G. Lee, J. Cho, D. Papailiopoulos, and K. Lee. Predictive pipelined decoding: A compute-latency trade-off for exact llm decoding. *arXiv preprint arXiv:2307.05908*, 2023.

[209] C. Yin, B. Acun, C.-J. Wu, and X. Liu. Tt-rec: Tensor train compression for deep learning recommendation models. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 448–462, 2021. URL `https://proceedings.mlsys.org/paper/2021/file/979d472a84804b9f647bc185a877a8b5-Paper.pdf`.

[210] Y. You, I. Gitman, and B. Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 6, 2017.

[211] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[212] M. Yu, Z. Lin, K. Narra, S. Li, Y. Li, N. S. Kim, A. Schwing, M. Annavaram, and S. Avestimehr. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed cnn training. *arXiv preprint arXiv:1811.03617*, 2018.

[213] Y. Yu, J. Wu, and J. Huang. Exploring fast and communication-efficient algorithms in large-scale distributed networks. *arXiv preprint arXiv:1901.08924*, 2019.

[214] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[215] H. Zhang, J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International Conference on Machine Learning*, pages 4035–4043, 2017.

[216] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[217] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

[218] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, et al. H _2 o: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*, 2023.

[219] H. Zhao, Z. Yang, Y. Cheng, C. Tian, S. Ren, W. Xiao, M. Yuan, L. Chen, K. Liu, Y. Zhang, et al. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[220] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, et al. Understanding and codesigning the data ingestion pipeline for industry-scale recsys training. *arXiv preprint arXiv:2108.09373*, 2021.

[221] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, et al. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*, pages 1042–1057, 2022.

[222] M. Zhao, S. Pan, N. Agarwal, Z. Wen, D. Xu, A. Natarajan, P. Kumar, R. Tijoriwala, K. Asher, H. Wu, et al. {Tectonic-Shift}: A composite storage fabric for {Large-Scale}{ML} training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 433–449, 2023.

[223] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[224] S. Zheng, Z. Huang, and J. Kwok. Communication-efficient distributed blockwise momentum sgd with error-feedback. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 11450–11460. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/80c0e8c4457441901351e4abbcf8c75c-Paper.pdf`.

[225] Y. Zhou, N. Du, Y. Huang, D. Peng, C. Lan, D. Huang, S. Shakeri, D. So, A. M. Dai, Y. Lu, et al. Brainformers: Trading simplicity for efficiency. In *International Conference on Machine Learning*, pages 42531–42542. PMLR, 2023.