Sequential Decision Making for Large Scale Machine Learning

by

Rui Liu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Associate Professor Barzan Mozafari, Chair
Research Assistant Professor Reza Soroushmehr
Assistant Professor Lu Wang
Professor Lei Ying

Rui Liu

ruixliu@umich.edu

ORCID iD: 0000-0002-7647-4208

© Rui Liu 2023

ACKNOWLEDGMENTS

I would like to thank all the people who have supported me through my PhD. Without them, this dissertation would never have been possible. There are so many people to whom I owe my gratitude, and I will highlight a few:

I would like to thank my advisor, Professor Barzan Mozafari, for his tremendous support during my PhD. His continuous encouragement and assistance helped me expand my research areas, from multi-armed bandit, to distributed learning, to transformer models which are the primary driving forces for AI's recent successes. He could always give me insightful feedback and suggestions whenever I needed his help, no matter if it was about research or life in general. I am truly grateful to have had the honor of working with him for the past few years.

I am also thankful to Professors Reza Soroushmehr, Lu Wang and Lei Ying for serving on my dissertation committee. Their valuable feedback helped me improve my dissertation. I really appreciate their time and help.

During my PhD, I have spent several summers interning at Meta and Microsoft, where I had the chance to work on industrial scale machine learning systems. I learned a lot from my mentors and colleagues, broadening my eyesight in many ways.

I am also very lucky to have made many friends at the University of Michigan, from the database system group, AI lab and various other groups. In particular, a bunch of friends at the Key-to-Success group spent a lot of time discussing technical ideas over the past few years, along with many fun nights playing board games together, which I will never forget.

Finally, and most importantly, my family. Their unconditional love and support have always been my source of strength, keeping me optimistic for whatever may come in the future.

TABLE OF CONTENTS

Acknowledgments . . . . . **ii**

List of Figures . . . . . **vi**

List of Tables . . . . . **ix**

List of Appendices . . . . . **xi**

Abstract . . . . . **xii**

Chapter

1 Introduction **1**

1.1 Large Scale Machine Learning: Powerful but Expensive 1
1.2 Sequential Decision Making 2
1.3 Decision Types 3
1.4 Related Work 5
1.5 Contributions 6

Part I Deciding Coordinate Selection **10**

2 Best Arm Identification for Maximum Inner Product Search (MIPS) **11**

2.1 Motivation 11
2.2 Related Work 15
2.2.1 Existing Approaches to MIPS 15
2.2.2 Existing Approaches to MAB 16
2.3 A Bandit Approach to MIPS 17
2.4 Multi-Armed Bandit with Bounded Pulls (MAB-BP) 18
2.5 BOUNDEDME: An Algorithm for Solving MAB-BP 19
2.5.1 A Concentration Inequality for MAB-BP 19
2.5.2 The BOUNDEDME Algorithm 21
2.6 Evaluations 23
2.6.1 Characteristics of the BOUNDEDME Algorithm 24
2.6.2 BOUNDEDME vs. Other MIPS Algorithms on Synthetic Datasets 26
2.6.3 BOUNDEDME vs. Other MIPS Algorithms on Real-World Datasets . . . 26
2.7 Summary 27

3 Gradient Sparsification for Distributed Learning . . . 28

3.1 Motivation . . . 28
3.2 Related Work . . . 32
3.3 Joint Sparsification of Gradient Computation and Communication . . . 34
3.3.1 Bandit Method for Distribution Learning . . . 37
3.3.2 Large Batch Optimizer Variant with Layerwise Adaptive Learning Rates 40
3.3.3 Implementation Details . . . 41
3.4 Evaluations . . . 42
3.4.1 Setup . . . 43
3.4.2 Convergence Comparison . . . 44
3.4.3 Convergence Comparison on ImageNet . . . 45
3.5 Summary . . . 46

Part II Deciding Example Selection 47

4 Sample Efficient Transformer with Memory Replay . . . 48

4.1 Motivation . . . 48
4.2 Related Work . . . 50
4.3 Transformer with Memory Replay . . . 51
4.3.1 Model Architecture . . . 51
4.3.2 Joint Learning . . . 55
4.3.3 Memory Replay Weights . . . 57
4.4 Evaluation Setup . . . 58
4.4.1 Pre-training . . . 58
4.4.2 Fine-tuning . . . 59
4.5 Evaluation Results . . . 60
4.5.1 Pre-training Efficiency . . . 60
4.5.2 Alternative Strategies for Weight Initialization . . . 63
4.5.3 Runtime Cost . . . 65
4.6 Summary . . . 66

5 Adam with Bandit Sampling . . . 67

5.1 Motivation . . . 67
5.2 Related Work . . . 70
5.3 Preliminaries about Adam . . . 71
5.4 Adam with Bandit Sampling . . . 72
5.4.1 Adaptive Mini-Batch Selection . . . 72
5.4.2 Update of Distribution p^t . . . 74
5.4.3 Comparison with Uniform Sampling . . . 76
5.5 Evaluations . . . 78
5.5.1 Setup . . . 78
5.5.2 Convolutional Neural Networks . . . 79
5.5.3 Recurrent Neural Networks . . . 81
5.5.4 Recurrent Convolutional Neural Networks . . . 81

5.6 Summary . . . 82

6 Selective Classification with Curriculum Learning . . . 83

6.1 Motivation . . . 83
6.2 Related Work . . . 86
6.3 Background . . . 87
6.3.1 Selective Classification Problem Formulation . . . 87
6.3.2 Selective Neural Network Model . . . 89
6.4 Curriculum-inspired Training Method . . . 90
6.4.1 Mini-batch Construction . . . 91
6.4.2 Mini-batch Ordering . . . 92
6.4.3 Difficulty Score Estimation . . . 93
6.5 Evaluations . . . 95
6.5.1 Setup . . . 95
6.5.2 Selective Classification Results . . . 97
6.5.3 Ablation Study . . . 98
6.6 Summary . . . 99

Part III Deciding Token Routing 101

7 Communication Efficient Regularization for Mixture-of-Experts Transformers . . . 102

7.1 Motivation . . . 102
7.2 Background . . . 104
7.2.1 From Densely Activated Transformers to Sparsely Activated Transformers 104
7.2.2 All-to-all Communication Cost of MoE Models . . . 107
7.3 Gating Dropout as Communication-efficient Regularization . . . 108
7.3.1 Extension with Expert LayerDrop . . . 111
7.4 Evaluations . . . 112
7.4.1 Setup . . . 113
7.4.2 Results on WMT-10 Dataset . . . 115
7.4.3 Results on Web-50 Dataset . . . 116
7.4.4 Effect of Dropout Rate . . . 117
7.5 Related Work . . . 118
7.6 Summary . . . 118

8 Conclusions . . . 120

Bibliography . . . 123

Appendices . . . 144

LIST OF FIGURES

FIGURE

1.1 Dissertation structure with each part about one type of decision. 7

2.1 Correctness of BOUNDEDME's guarantees 24
2.2 Synthetic Gaussian dataset 24
2.3 Synthetic uniform dataset 25
2.4 Real-world datases 25

3.1 Convergence in terms of wall-clock time, confirming our method JOINTSPAR-LARS's faster convergence than baselines. 43
3.2 Speedup of JOINTSPAR-LARS against baseline methods, computed as the ratio of the wall-clock time spent by each baseline to that by JOINTSPAR when achieving the same testing loss. 44
3.3 Convergence comparison for training AlexNet (with 62 million parameters) on ImageNet dataset. JOINTSPAR-LARS converges faster than baseline methods. 45

4.1 The architecture diagram of our *Transformer with Memory Replay* (TMR). The generator produces corrupted examples, which are being saved to the fixed-size memory buffer. The discriminator samples examples from the memory buffer for training. High-quality examples are reused by the discriminator, thus making it more sample efficient to train the discriminator. 49
4.2 Forward pass of *Transformer with Memory Replay* (TMR). For simplicity, mini-batch size is set to 1 in the diagram to illustrate how an example is changed. We put $*$ right after a token to indicate this token has been replaced by the generator. Memory replay between the generator and discriminator helps the discriminator to learn more efficiently, by providing high-quality examples to it. The sentence "the weather today feels* so good*" is saved to the memory buffer, and a better-quality sentence "cats* fight over bones" which is sampled from the memory buffer is instead provided to the discriminator as input. 56
4.3 Pre-training efficiency comparison. Our models (i.e., TMR(loss_diff) and TMR(grad_norm)) have better scores than baseline ELECTRA when the pre-training has gone through enough iterations, because the memory replay needs time to accumulate enough high-quality examples. 62

4.4 Runtime efficiency comparison using small model on GLUE. TMR(loss_diff) has better runtime efficiency than the baseline ELECTRA, because TMR(loss_diff) achieves better scores after pretaining for the same amount of wall-clock time. 63
4.5 Comparison of alternative strategies for weight initialization. TMR(LSR) and TMR(LinUCB) are much worse than TMR(loss_diff) as the number of iterations increases. 65

5.1 Adam with Bandit Sampling. Our method ADAMBS generalizes Adam with the ability to adapt to different examples by using a bandit method to maintain a distribution p over all training examples. 68
5.2 CNN model on CIFAR10, CIFAR100 and Fashion MNIST 79
5.3 RNN model on MNIST and Fashion MNIST 80
5.4 RCNN model on IMDB 81

6.1 Diagram of our curriculum-inspired method for training the selective neural network model. The training examples are ordered by their difficulty scores as shown in the left-most box. Hard examples are at the top (red color) and easy examples are at the bottom (green color). At each iteration, the mini-batch is constructed by sampling from both the hard part and the easy part, with their ratio appropriately calibrated. As training process goes, we gradually narrow the difficulty gap between easy and hard examples, indicated by the boundaries of the easy and hard parts moving towards each other in the diagram. 94

7.1 The MoE model architecture. The FFN sub-layer in the original transformer architecture is replaced by the MoE sub-layer, which consists of experts. 105
7.2 The all-to-all communication illustration of an MoE model with two experts in a two-machine setting. Expert 0 and 1 reside at machine 0 and 1, respectively. Each machine has a local batch of 4 tokens to process. All-to-all communication is needed right before (and right after) MoE sub-layer. 105
7.3 The all-to-all communication cost. 106
7.4 The Gating Dropout illustration of an MoE model with two experts in a two-machine setting. Expert 0 and 1 reside at machine 0 and 1, respectively. Each machine has a local batch of 4 tokens to process. Gating Dropout allows us to send tokens to the local experts at the same machines with probability p (denoted by the dashed arrows). . . . 108
7.5 BLEU score vs. training time (in seconds) on the WMT-10 dataset. Both Gate-Drop and Gate-Expert-Drop reach the target BLEU (denoted as the black line) much faster than the baseline. 114
7.6 The effect of the dropout rate on throughput and BLEU Δ (i.e., difference w.r.t. the baseline's BLEU) for Gate-Expert-Drop on the WMT-10 dataset (measured after training for 35k steps). 116

B.1 Convergence in terms of the number of iterations, confirming that our method JOINTSPAR enjoys a similar convergence rate as most baselines. 158
B.2 Convergence in terms of wall-clock time, confirming our method JOINTSPAR's faster convergence than baselines. 159
B.3 (a) Communication time per iteration of each method, (b) total time per iteration, and (c) total time to communication time ratio. 160

B.4 Convergence rate for three different levels of sparsity budget s: low (0.1), medium (0.5), and high (0.9). Train loss is measured in (a), and test loss is measure in (b). . . . 161

C.1 Experiments comparing with SGD variants. 173
C.2 Error Plot of CNN models 173
C.3 Error Plot of RNN and RCNN models 173

D.1 Testing loss vs. the number of iterations for different models on different datasets. Our method convergences faster by achieving a lower testing loss for the same number of iterations. 176

LIST OF TABLES

TABLE

2.1 State of the art algorithms for MIPS. Here, n is the number of data vectors and N is the vectors' dimensionality. 14

3.1 Breakdown of computation time (t_{comp}) versus communication time (t_{comm}) per epoch (in seconds) for vanilla SGD versus gradient compression methods in large batch optimization. The results are obtained by training AlexNet on ImageNet dataset using 8 machines with P100 GPUs connected with a 100Gb/s InfiniBand fabric (batch size is set as a large value, i.e., 1024 for each machine). While gradient compression methods (e.g., TERNGRAD and GSPAR) reduce communication time by 2-3x, the computation time becomes the dominant part. Note that the slight increase in computation time of the gradient compression methods is due to time spent compressing and uncompressing gradients. 31
3.2 Notation summary. 33
3.3 Target accuracy when measuring the training time for speedup comparison 44

4.1 Hyperparameter values for small and base models. Generator is smaller than discriminator in terms of hidden size, FFN-size and number of attention heads. 60
4.2 GLUE benchmark details. For each task, we list the task description and the size of the training set. 61
4.3 Results on GLUE of the small models after pre-trained for 200k steps. 61
4.4 Runtime cost comparison. The value in the table is the wall clock time (in seconds) to finish 100 iterations. 66

5.1 Convergence rates under doubly heavy-tailed distribution. 78

6.1 Results of moderate-sized CNN model on CIFAR10. Testing classification error rates (in $\%$) are shown for different target coverage rates. The relative improvement of error rates of our method w.r.t. SelectiveNet is shown at the last column. 95
6.2 Results of moderate-sized CNN model on CIFAR100. Testing classification error rates (in $\%$) are shown for different target coverage rates. The relative improvement is shown at the last column. 96

6.3 Results of moderate-sized CNN model on SVHN. Testing classification error rates (in %) are shown for different target coverage rates. The relative improvement is shown at the last column. 96
6.4 Results of VGG model on CIFAR100. Testing classification error rates (in %) are shown for different target coverage rates. The relative improvement is shown at the last column. 97
6.5 Ablation study with moderate-sized CNN on CIFAR10. Testing classification error rates (in %) are shown below. 98

7.1 Relative throughput improvement of no-alltoall compared to baseline. 107
7.2 The results on WMT-10 dataset using models with Transformer-base architecture. The target BLEU score is the value achieved by the baseline method at convergence, i.e., 24.60. The training time to the target BLEU is in seconds. Hash-Layer uses a random hash function to replace the gating network. We use - in the table to indicate that Hash-Layer cannot reach the target BLEU. 112
7.3 The throughput results on Web-50 dataset measured in two different clusters. 114
7.4 The results on Web-50 dataset using models with Transformer-big architecture. **BLEU (avg)** is the average BLEU score for translating between English and other languages (both directions). **E**→**X** only considers translating from English to other languages. **X**→**E** only considers translating from other languages to English. And **(low)** is used to indicate that only low-resource languages are considered. 115

D.1 Ablation study with moderate-sized CNN on CIFAR10. Testing error rates (in %) are shown below. 176
D.2 The number of iterations to reach the target loss for training moderate-sized CNN on the CIFAR10 dataset. The target loss value is the lowest testing loss achieved by SELECTIVENET for each coverage rate. 177
D.3 Coverage violation of moderate-sized CNN on CIFAR10. Empirical Coverage (in %) are shown for different target coverage rates. Our method achieves smaller coverage violation (difference between target coverage and empirical coverage) than SELECTIVENET. 178

LIST OF APPENDICES

A Proofs for Chapter 2 . . . . . **144**

B Proofs for Chapter 3 . . . . . **148**

C Proofs for Chapter 5 . . . . . **162**

D More Results for Chapter 6 . . . . . **174**

ABSTRACT

Large scale machine learning lies at the core of many artificial intelligence's recent successes exemplified by AlphaGo, BERT, DALL-E, GitHub Copilot, AlphaCode, and ChatGPT. These large scale models are undoubtedly powerful but very expensive to train. To make them less expensive, we incorporate sequential decision making into machine learning model training. Sequential decision making has long been the focus of stand-alone fields (e.g., reinforcement learning and multi-armed bandit). We observe that sequential decision making problems also appear in the context of training machine learning models under several different settings. We show that carefully resolving these problems leads to improved training efficiency, thus reducing the training time. Depending on the specific training settings, the decisions are about how to (a) select coordinates of the vectors, (b) select examples from the training set, or (c) route tokens across different machines. In this dissertation, we consider a variety of training settings including recommendation systems, distributed learning, curriculum learning and transformer models. We design decision making strategies tailored for each training setting, and demonstrate reduced training times and often better accuracies through extensive experiments.

Chapter 1

Introduction

1.1 Large Scale Machine Learning: Powerful but Expensive

Over the last few years, machine learning models have become more and more powerful, achieving unprecedented or even super-human performance in many real-world applications. For example, powered by advances in machine learning, AlphaGo is the first computer program to beat world champions in the game of Go [210]. BERT achieves super-human performance in several natural language tasks such as question answering [53]. DALL-E is a revolutionary tool to generate realistic images in multiple styles based on text prompts [180]. GitHub Copilot relies on the Codex model to provide high-quality code completion services for over a dozen programming languages, including Python, JavaScript, PHP and Shell [32]. AlphaCode, another code generation system, can achieve an estimated rank within the top 54% among more than $5,000$ participants in challenging programming competitions [133]. Most recently, OpenAI released ChatGPT, which is a chatbot that can interact with humans in an unprecedented way, such as answer followup questions and admit its mistakes [1].

Even though these exciting applications span across many fields such as games, programming language generation, computer vision and natural language understanding, they have one thing in common: they all rely on **large scale machine learning** models. For example, the largest model used for AlphaCode contains 41 billion parameters [133]. BERT has 340 million parameters in

its earlier version [53], and has been further scaled up to 395 billion parameters by using the new Mixture-of-Experts (MoE) design [64]. The GPT-3 model, the underlying model for DALL-E and GitHub Copilot, contains 175 billion parameters [25]. ChatGPT is fine-tuned from a model from the GPT-3.5 series containing up to 175 billion parameters [1].

Actually, large scale machine learning models have been the foundation for the advances of machine learning, because it has been observed that as models become bigger and bigger, the performance (e.g., accuracy) is better and better. However, bigger models require more data and longer training time, thus resulting in a very high training cost. For example, it is estimated that it takes 168 days to train a GPT-3 model with 178 billion parameters using 256 NVIDIA A100 GPUs [163]. Training models for AlphaCode is reported to require hundreds of petaFLOPS days [133].

The goal of this dissertation is to *develop methods to reduce the running cost for large scale machine learning algorithms*. Specifically, we draw upon techniques from the area of sequential decision making. As we will elaborate later, we observe that sequential decision making appears in the context of training machine learning models under several different settings. By carefully resolving these sequential decision making problems, our methods improve the training procedures by reducing their computational complexity, thus leading to a shorter training time.

1.2 Sequential Decision Making

Sequential decision making has long been studied in stand-alone fields such as reinforcement learning [222]. They typically formalize the problem by imaging an agent interacting with its environment which the agent knows very little about at the very beginning. The goal is to decide the sequence of actions that the agent needs to take in order to achieve a pre-defined goal (e.g., exit a maze, win a game, collect as many scores as possible). In this dissertation, we do not consider the full scale setting of reinforcement learning, because its solutions are often sample greedy, thus very slow to converge. Rather, our methods are mostly based on a special setting called multi-armed

bandit [12, 26, 122], which is sufficient to characterize the model training procedures we aim to improve in this dissertation.

Multi-armed bandit can be understood by imaging a gambler (i.e., agent) in a casino in front of n different slot machines (i.e., environment). The agent knows little about these slot machines, except that he can choose one machine and pull its arm to get a reward. In other words, the agent has n different actions to choose from, each one corresponding to pulling one machine's arm. Over T rounds, the agent needs to achieve a goal by carefully choosing one action at each round. Broadly speaking, there are two alternative goals in the literature. The first goal is to determine the order to pull arms in order to identify the best arm with as few pulls as possible. Alternatively, the goal could be to determine the order to pull arms in order to accumulate as many rewards as possible.

Exploration–Exploitation Tradeoff. Any method that attempts to solve this multi-armed bandit problem for the second goal (i.e., accumulate as many rewards as possible) has to deal with the so-called exploration-exploitation tradeoff. Specifically, at each round, we could either go for exploration, that is, select other under-explored arms to get more information about them, or go for exploitation, that is, select the arm that is empirically the best based on already collected information. We need to balance this tradeoff, because if we always explore, we might be unlucky and end up selecting bad arms that return low rewards. On the other hand, if we always exploit, we might miss a better arm that looks bad temporarily due to lack of enough information.

1.3 Decision Types

We consider three types of decisions in this dissertation: (a) coordinate selection, (b) example selection and (c) token routing.

Coordinate selection. Given high-dimensional vectors, we may want to select some coordinates at each iteration in order to save the amount of computations on the vectors or reduce the size of vectors. In this case, **the decisions are about which coordinates to select**. For example, in distributed learning where multiple machines are used to collectively train a model, at each training

iteration, gradient vectors need to be sent from each worker machine over the communication links for the aggregation purpose. However, for a large scale model, the gradient vectors are high-dimensional, because the dimension of gradient vectors is the same as the number of trainable parameters in the model (e.g., 175 billion for GPT-3). Sending the full gradient vectors (e.g., 175 billion dimensional vectors for GPT-3) from each worker machine will take a large amount of communication time, sometimes even larger than the computation time on commodity systems with limited network bandwidth. An effective way to reduce the communication time here is to select some important coordinates and only send these coordinates over the communication link (i.e., gradient sparsification). Because the training procedure is typically iterative (e.g., using SGD for training), we are faced with this type of decision across multiple iterations. This is where sequential decision making comes into play, helping us make better decisions about which coordinates to select across many training iterations.

Example selection. Given a large set of training examples, we may want to select some examples from it at each iteration. This is motivated by the common practice of using a mini-batch (i.e., a small sample from the training set) to train deep neural networks at each iteration; this practice has been shown to result in lower computation time and better performance at convergence due to the stochastic nature of example selection. In this case, **the decisions are about which examples to select**. Exiting works use simple strategies to select examples (e.g., random selection by treating all examples equally, or importance-based sampling without the appropriate exploration–exploitation tradeoff). For example, we endow Adam with a new type of sampling based on multi-armed bandit, which we call *bandit sampling*. This sampling strategy will help Adam select important examples while taking into consideration the exploration–exploitation tradeoff. We also consider other settings where it is beneficial to carefully select examples in different ways (e.g., save some past examples for future use if the examples are generated on the fly, or follow an easy-to hard curriculum when the training is sensitive to example difficulty).

Token routing. As models become larger, routing strategies are introduced inside the model architecture so that not all model parameters are used for every input example, in order to control

the models' computational complexity [18, 64]. Essentially, each example or token (i.e., the vector representation of a word when the input is text) is routed through different parts of the model when necessary. We will use token here because the most successful model of this type is Mixture-of-Experts (MoE) transformer which is used for text data [64, 111]. When the model is too big to fit into one machine, different parts of the model is distributed among multiple machines. It means that each token needs to be routed across multiple machines. In this case, **the decisions are about how to route tokens across machines**. With an appropriate design for the routing strategy, our goal is to reduce the cross-machine communication cost without hurting the model's accuracy at convergence. Actually we will show that better model accuracy can be achieved because some routing strategies have a regularization effect during training.

1.4 Related Work

Coordinate selection has been studied in the context of coordinate descent methods which aim to update several coordinates of the entire parameter vector at each iteration [132, 192, 229, 230]. Most of these works focus on the convergence properties of coordinate descent methods in one-machine settings [82, 138, 168] or special multi-machine settings (e.g., feature-based data partition strategy [186] or primal-dual formulation [94]). We explicitly consider gradient sparsification in the general multi-machine settings where communication cost is reduced by designing sampling strategies on how to select coordinates of gradient vectors [5, 30, 123, 137, 184, 217, 231].

Example selection is inspired by the long-time efforts in the machine learning community to study how to take advantage of the difference among training examples, e.g., in many boosting algorithms [39, 143, 197, 198, 199]. The well-known AdaBoost algorithm [198] builds an ensemble of base classifiers iteratively, of which each base classifier is trained on the same set of training examples with adjusted weights. Because of the way AdaBoost adjusts weights on training examples, it can focus on examples that are difficult, thus decreasing the training error very quickly. In addition, it has been often observed in experiments that AdaBoost has very good generalization

ability, discussed and analyzed in several works [143, 191, 199]. Similar to AdaBoost, some of our methods also aim to improve training by iteratively adjusting example weights. However, the amount of available information is very different every time they adjust example weights. AdaBoost receives full information, in the sense that each training example needs to run through the up-to-date ensemble model to determine which examples are still misclassified. Our methods only receive partial information because we can only select a mini-batch at each iteration, bringing up the tradeoff between exploration (i.e., select other examples to get more information) and exploitation (i.e., select the empirically best examples based on already collected information), which is where multi-armed bandit comes to play. Curriculum learning [20, 83] is another optimization strategy that leverages a pre-trained teacher model to train the target model. The teacher model is critical in determining how to assign examples to mini-batches, except in Chapter 6 where we focus on the case when we do not have access to an extra teacher model. However, utilizing a teacher model is likely to further improve the performance of our method, which motivates our idea to use curriculum learning for selective classification as in Chapter 6.

Token routing is an active topic to improve MoE transformer training. Recent proposals include using hierarchical representation to learn the routing network [260], using pre-defined hash functions to decide routing strategies [189] and routing tokens to additional experts for student-teacher co-training [270]. In addition, token dropping methods that allow some tokens to skip the computation of certain network layers have been proposed for improving model inference [79, 109, 239] and training [88, 257]. Our token routing strategies in Chapter 7 also incorporate the token dropping idea to further speed up computation.

1.5 Contributions

Our intuition is that there are many settings of model training procedures, where making a sequence of decisions properly can reduce the computational complexity, thus accelerating the training procedures. Depending on the specific training procedures, the decisions could be (a) how

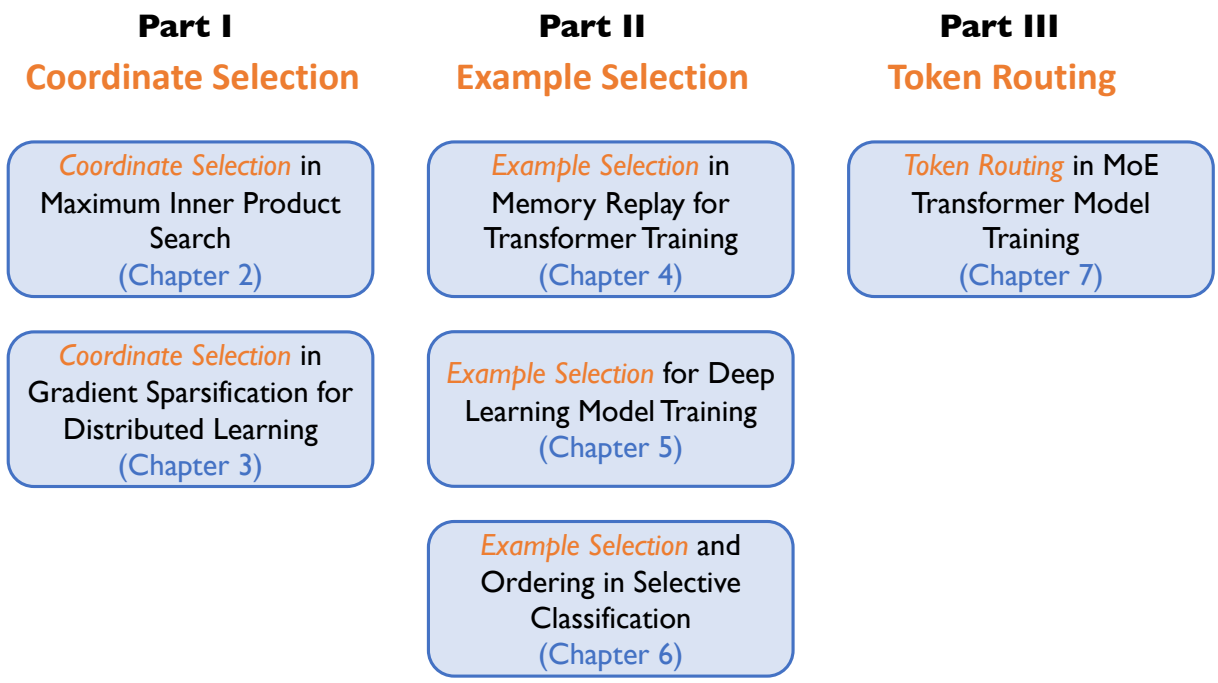


Figure 1.1: Dissertation structure with each part about one type of decision.

to select coordinates of vectors (e.g., bullet points (1) - (2) as below), or (b) how to select examples from the training set (e.g., (3) - (5) as below), or (c) how to route tokens across different machines (e.g., (6) as below). Therefore, the following chapters of this dissertation are organized into three parts. Chapters 2 and 3 comprise the first part where the decisions are about coordinate selection. Chapters 4 to 6 comprise the second part where the decisions are about example selection. The last part is Chapter 7 where the decisions are about token routing. This structure is also illustrated in Figure 1.1. In each chapter, our contributions are summarized as follows.

1. We aim to accelerate maximum inner product search (MIPS), which is a subproblem that appears in several machine learning algorithms and applications (e.g., recommendation systems). We cast MIPS into a best arm identification problem for which we need to decide which vector coordinates to compute at each round. (Chapter 2)

2. Gradient sparsification is a popular technique that reduces the communication cost in distributed training where multiple machines are used to collectively train a model. We propose a

new gradient sparsification method by casting it into a multi-armed bandit problem for which we need to decide which coordinates of the gradient vectors should be skipped for a lower communication cost when aggregating gradients. (Chapter 3)

3. We focus on transformer models which have shown state-of-the-art performance for many NLP tasks. We aim to improve the sample efficiency of training transformer models by leveraging the memory replay mechanism for which we need to decide which examples to save to and sample from the memory buffer. (Chapter 4)

4. We propose a generalization of Adam, a popular optimizer for training deep learning models. We aim to accelerate training deep learning models by enabling our new Adam optimizer to select training examples in each minibatch based on their importance in terms of model's convergence. (Chapter 5)

5. We consider training neural network models for selective classification, where the models have the reject option to abstain from predicting certain examples if not confident. We aim to improve selective neural network training by adapting the curriculum learning idea for which we need to decide how to select examples in each mini-batch and how to order these mini-batches throughout the training process. (Chapter 6)

6. We consider training Mixture-of-Experts (MoE) transformers, which have drawn great interest due to their outrageous scaling capability without significant increases in computational cost. MoE transformers have shown to achieve state-of-the-art performance in various Natural Language Processing (NLP) tasks such as machine translation. In MoE transformers, tokens need to be routed across different machines. This routing strategy often incurs a huge cross-machine communication cost. We aim to improve MoE transformer training by proposing a new routing strategy which reduces the cross-machine communication cost without sacrificing the model's accuracy. In fact, we will show that our method leads to better generalization performance due to its regularization effect which is similar to dropout. (Chapter 7)

Most materials in this dissertation have been published in top-tier machine learning and AI conferences. Specifically, Chapter 2 is published at Liu et al. [145]. Chapter 3 is published at Liu and Mozafari [141]. Chapter 4 is published at Liu and Mozafari [142]. Chapter 5 is published at Liu et al. [146]. And Chapter 7 is published at Liu et al. [147].

Part I

Deciding Coordinate Selection

Chapter 2

Best Arm Identification for Maximum Inner Product Search (MIPS)

2.1 Motivation

Many matrix factorization-based recommendation systems solve the problem of *Maximum Inner Product Search* (MIPS) [166, 209, 261] to figure out which items to recommend to each user [42, 118, 144]. It also appears as a key step in several other machine learning algorithms and applications, such as multi-class prediction [51, 95], structural SVM [100, 101], and vision applications [51]. When the MIPS problem becomes large (e.g., the number of users or items is huge in the real-world recommendation systems), fast solution to MIPS is desired. In this chapter, we study an efficient solution to MIPS by leveraging the multi-armed bandit idea.

The MIPS problem can be formally defined as follows: given a collection of n data vectors, $\mathcal{S} = \{v_1, v_2, \cdots, v_n\}$, where $v_i \in \mathbb{R}^N, 1 \leq i \leq n$, and a query vector $q \in \mathbb{R}^N$, the goal is to find $v^* \in \mathcal{S}$ that maximizes (or approximately maximizes) the inner product $q^T v^*$. In other words, MIPS is the following problem:

$$v^* = arg\max_{v \in \mathcal{S}} q^T v \tag{2.1}$$

The naïve linear search for solving MIPS requires $O(n \cdot N)$ time to exhaustively compute all n inner products, which can be daunting for massive datasets (large n) and/or high-dimensional data

(large N). This has led to significant interest in devising sub-linear time algorithms to solve the MIPS problem approximately, including the best paper award at NIPS'14 [209] and many other proposals [15, 166, 261].

Motivation I: Preprocessing Overhead. Despite their different merits, existing approximate techniques for MIPS all share a common pattern: they require a substantial time for preprocessing the data set $\mathcal{S}$, during which they construct a data structure that can then be used to answer queries more efficiently. For example, they construct a hash table [209], a sorted index [261], space partition trees [15], or other data structures [14]. Existing methods require various data structures to be built during preprocessing time. In fact, the preprocessing time is, in some cases, so large that it even exceeds the time complexity of the naïve approach for answering $\log n$ queries, i.e., $O(Nn \log n)$ [261]. We have summarized the preprocessing time of the previous techniques in Table 2.1. The rationale is that query times will be faster after the preprocessing step. The preprocessing time is therefore justified when there are many queries and the set $\mathcal{S}$ remains the same, i.e., once the preprocessing is done it would benefit many subsequent queries. However, there are many cases where either the number of queries is relatively small (or even 1), or the set $\mathcal{S}$ changes frequently, e.g., the approximate Linear Minimization Oracle (LMO) in Matching Pursuit and Frank-Wolfe optimization [93, 152]. In these scenarios, the preprocessing time is simply a burden. In this chapter, our first motivation is to design an approach to MIPS that will not require any preprocessing, while still achieving a query speed-up better than those approaches that do require it.

Motivation II: Suboptimality Bounds. As summarized in Table 2.1, specific parameters such as the number of hash functions or depth of partition trees are used to trade search accuracy with search efficiency. Various data structures with pre-specified parameters [14, 15, 209] are built before any query is given, which means that the trade-off is somewhat fixed for all queries. Thus, the computational cost for a given query is fixed. However, in many real-world scenarios, each query might have a different computational budget. [261] proposed a greedy method where the user could control the computational budget for each query. The computational budget can be viewed as an efficiency-accuracy knob. Nevertheless, the user can't get a solution with a guaranteed level of

optimality in general.[1] It is crucial for practitioners to have a knob with which the user can explicitly request and guarantee a certain level of optimality for each query [3, 4, 160], e.g., to know how accurate the solution would be if more computational budget were allowed [170]. Thus, our second motivation in this chapter is to design a MIPS algorithm that can bound (and directly control) the suboptimality of the returned answer, regardless of the data distribution. Specifically, for any $0 < \epsilon < 1$ and $0 < \delta < 1$ chosen by the user, the algorithm must be able to guarantee that, with probability at least $1 - \delta$, the returned solution $\hat{v}$ is ϵ-optimal with respect to optimal solution v^*, i.e., $\frac{1}{N} q^T v^* - \frac{1}{N} q^T \hat{v} < \epsilon$. This would help us to better understand the behavior of our algorithm theoretically, and would offer a flexible knob for trading off error and computational efficiency in practice.

Our Approach. Our approach to MIPS is inspired by the Multi-Armed Bandit (MAB) problem [12, 26, 61, 98]. MAB is a predominant model for characterizing the tradeoff between exploration and exploitation in decision-making settings. In MAB, there are n arms; each time we pull an arm, it returns a reward (e.g. a reward generated by sampling from a Guassian distribution). The true mean of an arm is defined as the mean of the distribution from which its rewards are sampled. The goal in MAB is to either (1) accumulate as much reward as possible, or (2) identify the best arm (i.e., the one with the highest true mean). In this chapter, our goal is the latter. We cast MIPS as a Best Arm Identification problem: we can treat each data vector as an arm, where pulling it means multiplying one of its coordinates with the corresponding coordinate from the query vector. We must then dynamically decide how many more floating-point multiplications to perform for each inner product, based on the partial results of all inner products thus far.

There are two different stopping conditions for the Best Arm Identification problem: fixed confidence and fixed budget. With fixed confidence, the MAB algorithm seeks to minimize the sample complexity—the number of pulls used—while guaranteeing that the returned arm is ϵ-optimal with probability at least $1 - \delta$ for any given $0 < \delta < 1, 0 < \epsilon < 1$. In the fixed budget setting, the MAB algorithm stops once it has used its budget in terms of the sample complexity,

[1] A more detailed treatment of the previous work is deferred to the Related Work section of this chapter.

Table 2.1: State of the art algorithms for MIPS. Here, n is the number of data vectors and N is the vectors' dimensionality.

Method	Preprocessing Time	Query Time	Theoretical Guarantees	Notes
BOUNDEDME (our method)	0	$O\left(\frac{n\sqrt{N}}{\epsilon}\sqrt{\log\left(\frac{1}{\delta}\right)}\right)$	Guaranteed to return an ϵ-optimal solution with probability at least $1-\delta$	User can choose any desired error $0<\epsilon<1$ and confidence $0<\delta<1$
GREEDY-MIPS [261]	$O(Nn\log n)$	$O(BN)$	No guarantees in general. (They guarantee optimality with high probability, only for uniformly distributed data and budget $B \geq O(N\log(n)n^{\frac{1}{N}})$	For non-uniform data, the results can be arbitrarily poor (e.g., if the largest coordinate of q^Tv is identical for all $v \in \mathcal{S}$, the output will be a random subset)
LSH-MIPS [166, 209]	$O(Nnab)$	$O(\frac{nN}{2^a}b)$	They guarantee to return the optimal vector v^* for query q with probability $1-\left(1-\left(1-\frac{\cos^{-1}(q^Tv^*)}{\pi}\right)^a\right)^b$ where a is the number of bits in each hyper LSH function and b is the number of hyper LSH functions	Since v^* is unknown *a priori*, the users cannot control the lower bound of this probability (e.g., if $q^Tv^*=-1$, this probability will always be 0 regardless of the values chosen for a and b)
RPT-MIPS [107]	$O(LNn\log n)$	$O(L\log n)$	They guarantee to return the optimal vector v^* for query q with probability upper bounded by some potential function depending on q, S and L, where L is the number of trees	Since q is unknown *a priori*, the users cannot control the lower bound of this probability
PCA-MIPS [15]	$O(N^2n)$	$O(\frac{nN}{2^d})$	None	d is the depth of the PCA tree

while seeking to return an arm whose true mean is as close as possible to that of the best arm. Although our **Motivation II** is inline with the fixed confidence setting, the existing MAB methods for fixed confidence are not effective for the MIPS problem. This is because the existing algorithms are designed for i.i.d. rewards drawn from some unknown distribution over an *infinite* population [12, 26], and thus require many pulls to achieve an accurate estimate of an arm's true mean. In MIPS, however, the useful number of pulls for any arm is upper bounded by N (the vectors's dimension). In other words, rewards are sampled *without replacement* from a discrete uniform distribution over a *finite* population. Therefore, by exploiting this structure, we should be able to significantly lower

the number of pulls. In this chapter, we introduce a new setting for Best Arm Identification problem with fixed confidence that suits the special structure of MIPS. We also propose an algorithm, called BOUNDEDME, inspired by the Median Elimination framework [60], using a tight statistical bound for sampling without replacement.

In summary, we make the following contributions:

- We identify two desirable motivations for the MIPS problem that are important in practice but overlooked by existing solutions (**Movitaion I** and **Motivation II**). We introduce a new MAB setting for Best Arm Identification, where the rewards for each arm are sampled from a large but finite list. We call this setting Multi-Arm Bandit with Bounded Pulls (MAB-BP).

- We propose a new algorithm for MAB-BP, called BOUNDEDME, which extends Median Elimination with a tight statistical bound. When applied to MIPS, BOUNDEDME enjoys a significantly lower sample complexity than all previous MAB methods developed for fixed confidence setting. More importantly, as a bandit approach, BOUNDEDME does not require any preprocessing (unlike previous MIPS solutions).

- Our extensive experiments on both synthetic and real-world datasets show that BOUNDEDME's query time is 5–$10\times$ faster than state-of-the-art MIPS algorithms, despite their use of preprocessing.

2.2 Related Work

2.2.1 Existing Approaches to MIPS

There are a number of sampling-based methods for MIPS. For example, SAMPLE-MIPS [38] is a scheme that samples $(i, j) \in \{1, \cdots, n\} \times \{1, \cdots, N\}$ with probability proportional to $v_i^{(j)} q^{(j)}$. However, it requires that all candidate and query vectors be nonnegative. DIAMOND-MSIPS [16] is another sampling-based approach which solves a similar problem, called *maximum squared inner product search (MSIPS)*. The goal in MSIPS is to find candidate vectors $v \in \mathcal{S}$ for which $(q^T v)^2$

is maximized. The solution to MSIPS, however, can be very different than that of MIPS, e.g., the former might return a v whose inner product with q is a large negative value.

Another popular approach is to reduce MIPS to the nearest neighbor search problem, which can then be solved using locality-sensitive hashing [166, 209], neighbor-sensitive hashing [169], PCA-trees [15], or K-Means approaches [14]. Nonetheless, all of these methods share a common pattern. Before answering any queries, they conduct a preprocessing on $\mathcal{S}$ to construct an approach-specific data structure, e.g., hash table in LSH-MIPS, space partition trees in PCA-MIPS, or cluster centroids in [14]. These data structures only contain information about the vectors in $\mathcal{S}$ and are independent of any query. Then, for each query, they use an efficient procedure on their preconstructed data structure to select a set of candidate vectors (i.e., a subset of $\mathcal{S}$). They perform an exact ranking on this candidate set to return the best vector. A recent approach, Randomized Partitioning Tree (RPT-MIPS) [107], builds a partitioning tree on top of an LSH scheme to solve the MIPS problem. RPT-MIPS guarantees the exact solution with a probability that depends on the vector set S and the given query. However, RPT-MIPS cannot directly control the quality of the returned vector (i.e., suboptimality bounds in **Motivation II**). Another approach is GREEDY-MIPS [261], which builds on the same algorithmic pattern, but also provides a budget for the size of the candidate set. This parameter is the only mechanism that implicitly controls the tradeoff between precision and query time. Table 2.1 summarizes the theoretical guarantees offered by some of the recent MIPS algorithms.

2.2.2 Existing Approaches to MAB

Best Arm Identification is a popular setting in MAB that aims to identify the best arm by dynamically deciding on how many times to pull each arm. In the fixed budget setting, the main idea behind algorithms such as Successive Halving [97, 104] and Successive Rejects [12] is to dynamically allocate the budget to the different arms in order to remove the bad arms in a round-by-round fashion until only one arm is left. Naturally, these methods tend to use up their budget in order to return an arm whose true mean is as close as possible to the optimal arm. The fixed budget setting

isn't suitable for our problem in this chapter. In the fixed confidence setting, algorithms share the same idea of dynamically pulling arms and removing the unpromising ones from consideration, such as Successive Elimination [61], Exponential Gap Elimination [104], LUCB [103], and Lil'UCB [98]. These algorithms are inline with our **Motivation II**, as they also seek to minimize sample complexity while guaranteeing that the returned arm is within a pre-specified proximity of the optimal arm. However, they cannot be directly applied to the MIPS problem because they assume the rewards are i.i.d. samples from some unknown distribution over an *infinite population*, whereas in MIPS, the rewards are sampled without replacement from a discrete uniform distribution over a *finite* list.

2.3 A Bandit Approach to MIPS

In this section, we show how the MIPS problem can be viewed as a Best Arm Identification problem with fixed confidence. As previously mentioned, the goal in MIPS is to solve the problem:

$$v^* = arg\max_{v \in \mathcal{S}} q^T v$$

where $\mathcal{S} = \{v_i | v_i \in \mathbb{R}^N, 1 \leq i \leq n\}$ is a collection of n data vectors and $q \in \mathbb{R}^N$ is a given query.

We cast MIPS as a bandit problem as follows. For every data vector $v_i \in \mathcal{S}$, we consider a corresponding arm a_i, whose reward has the following true mean $p_i = \frac{1}{N}\sum_{j=1}^{N} v_i^{(j)} q^{(j)} = \frac{v_i^T q}{N}$, where $v_i^{(j)}$ and $q^{(j)}$ are the j-th coordinates of v_i and q, respectively, for $1 \leq j \leq N$. When the arm a_i is pulled t times ($1 \leq t \leq N$), its rewards are generated by taking t i.i.d. samples *with* replacement from its reward list R_i, defined as the set $R_i = \{v_i^{(1)} q^{(1)}, v_i^{(2)} q^{(2)}, \cdots, v_i^{(N)} q^{(N)}\}$.

Each time the arm a_i is pulled, returning a reward corresponds to a floating-point operation in MIPS for multiplying one of the coordinates from v_i with its counterpart from q, i.e., computing $v_i^{(j)} q^{(j)}$ for some $1 \leq j \leq N$. The reward lists are initially unknown, but the more we pull an arm, the more we learn about its reward list. Our goal in MIPS is to find the arm with the highest true mean—the vector whose inner product with q is the highest—using as few floating point operations

as possible.

Unfortunately, in a traditional bandit setting, even if we pull the a_i arm N times, we still do not know the exact true mean of R_i, i.e., the exact inner product of v_i and q. This is because traditional bandit problems are designed for an unknown distribution over an infinite population, and hence rely on sampling *with* replacement. However, to solve MIPS, the rewards are drawn from a finite reward list—i.e., N coordinates. Thus, if we can exploit this structure and use sampling *without* replacement, we must be able to pull the arms significantly fewer times than in a traditional bandit. Further, once we have pulled the a_i arm N times, we should know the entire content of R_i, equivalent to the exact computation of the inner product between v_i and q. Next, we formally define this new bandit setting.

2.4 Multi-Armed Bandit with Bounded Pulls (MAB-BP)

We now formally introduce a new Multi-Armed Bandit setting, which we call *Multi-Armed Bandit with Bounded Pulls (MAB-BP)*. Assume a set of n arms $A = \{a_1, \cdots, a_n\}$. Each arm a_i is associated with a reward list $R_i = \{R_i^{(1)}, R_i^{(2)}, \cdots, R_i^{(N)}\}$, where N is the size of the reward list. Here, we assume $R_i^{(j)} \in [0, 1]$, but a similar analysis applies as long as the reward value is bounded. Every time an arm a_i is pulled, a reward is returned by sampling a value *without* replacement from its reward list R_i. Denote the true mean of reward for arm a_i as $p_i = \frac{1}{N}\sum_{j=1}^{N} R_i^{(j)}$. Thus, once an arm is pulled N times, the mean of the returned rewards is exactly equal to the true mean p_i. Our goal is the same as in a traditional Best Arm Identification: to identify an ϵ-optimal arm with probability at least $1 - \delta$ using as few pulls as possible, where ϵ and δ are provided by the user. We say that an arm $\hat{a}$ is an ϵ-optimal arm if $p_{a^*} - p_{\hat{a}} < \epsilon$, where a^* is the optimal arm.

It is easy to see that by choosing $R_i^{(j)} = v_i^{(j)} q^{(j)}$ for $1 \leq j \leq N$, one can cast MIPS as a MAB-BP problem. However, note that MAB-BP can be used to solve any problem of the form:

$$arg \max_{1 \leq i \leq n} \sum_{j=1}^{N} f(i, j)$$

where f can be an arbitrary function.

For MIPS, $f(i,j) = v_i^{(j)} q^{(j)}$. However, one can also use MAB-BP to solve the Nearest Neighbor Search (NNS) problem: given a collection of vectors $\mathcal{S} = \{v_1, \cdots, v_n\}$, where $v_i \in \mathbb{R}^N, 1 \leq i \leq n$, and a query vector $q \in \mathbb{R}^N$, the goal is to find $v^* \in \mathcal{S}$ that is closest to q, i.e., $\|q - v^*\|^2 = \sum_{j=1}^{N} (q^{(j)} - v^{*(j)})^2$ is minimized. In this case, $f(i,j) = -(q^{(j)} - v_i^{(j)})^2$.

2.5 BOUNDEDME: An Algorithm for Solving MAB-BP

Existing bandit algorithms are sub-optimal for MAB-BP. The fundamental reason is bandit algorithms have to estimate the minimum number of samples needed to obtain an estimate $\hat{p}$ of the true mean p that satisfies the given error ϵ and confidence δ requirements, i.e., $\mathbb{P}\left[\hat{p} - p \leq \epsilon\right] \geq 1 - \delta$.

The efficiency of a bandit algorithm depends on how accurately it can estimate the number of required samples, based on the reward values it has observed for each arm. This goal is achieved using concentration inequalities [24]. Since in traditional bandit the rewards are typically assumed to be sampled from a sub-Gaussian distribution over an infinite population [96], these algorithms often rely on Hoeffding's bound or the law of iterated logarithm (LIL) bound to determine the sample size for mean estimation of the reward distribution over an infinite population. However, as noted earlier, the reward values in MAB-BP are sampled without replacement and from a finite list. Thus, an algorithm that can exploit this additional information should be able to solve the MAB-BP problem more efficiently (i.e., with lower sample complexity).

Next, we derive a concentration inequality for sampling without replacement and then present our algorithm.

2.5.1 A Concentration Inequality for MAB-BP

We use the following corollary from [17].

Corollary 2.1 (Corollary 2.5 in [17])**.** *Let $\mathcal{X} = (x_1, x_2, \cdots, x_N)$ be a finite set of size $N > 1$ in $[a, b]$ with mean $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$, and $(X_1, \cdots, X_m)$ be a list of size $m < N$ sampled without*

replacement from $\mathcal{X}$. Then for any $m \leq N$, and any $\delta \in [0,1]$, it holds

$$\mathbb{P}\left[\frac{1}{m}\sum_{t=1}^{m} X_t - \mu \leq (b-a)\sqrt{\frac{\rho_m \log(1/\delta)}{2m}}\right] \geq 1-\delta \tag{2.2}$$

where ρ_m is defined as

$$\rho_m = min\left\{(1-\frac{m-1}{N}), (1-\frac{m}{N})(1+1/m)\right\} \tag{2.3}$$

Based on this corollary, we could get the following concentration inequality for sampling without replacement, as in the following lemma[2].

Lemma 2.1. *Let $\mathcal{X} = (x_1, x_2, \cdots, x_N)$ be a finite set of size $N > 1$ in $[a,b]$ with mean $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$, and $(X_1, \cdots, X_m)$ be a list of size $m < N$ sampled without replacement from $\mathcal{X}$. Then, for any given $0 < \epsilon < 1, 0 < \delta < 1$, if*

$$m = min\{\frac{u+1}{1+\frac{u}{N}}, \frac{u+\frac{u}{N}}{1+\frac{u}{N}}\} \tag{2.4}$$

where $u = \frac{\log(1/\delta)}{2}\frac{(b-a)^2}{\epsilon^2}$, then we have

$$\mathbb{P}\left[\frac{1}{m}\sum_{t=1}^{m} X_t - \mu \leq \epsilon\right] \geq 1-\delta \tag{2.5}$$

.

We can see that as the error bound ϵ approaches 0, the required sample size m will approach the finite set size N, but never exceed N. This matches our previous intuition that it is ineffective to pull one arm more than N times in MAB-BP. It is worth noting that a lemma similar to Lemma A.1 can be derived to show that $\mathbb{P}\left[\frac{1}{m}\sum_{t=1}^{m} X_t - \mu \geq -\epsilon\right] \geq 1-\delta$, based on a corollary similar to Corollary A.1.

[2]All omitted proofs can be found in the supplementary material.

2.5.2 The BOUNDEDME Algorithm

Our proposed algorithm, BOUNDEDME, is based on the median elimination strategy, but tailored to our MAB-BP setting. The basic idea of median elimination strategy is that, given a set of arms A, we pull each of these arms for a certain number of times to update their empirical means, discard the worst half in terms of their updated empirical means thus far, and repeat until only one arm remains. In Algorithm 2.1, we present BOUNDEDME for the more general case of identifying the top K arms with the highest true means of rewards. (The best arm identification is a special case, where K=1.) To be more specific about the difference between Algorithm 2.1 and the general median elimination strategy, the number of times that we pull each remaining arm is $t_l - t_{l-1}$ for the l-th iteration, where t_l is defined in the line 7 of Algorithm 2.1. Since we want to identify the top K arms, at the end of the l-th iteration, we discard $\left\lceil \frac{|S_l|-K}{2} \right\rceil$ arms with least empirical means thus far where $|S_l|$ is the number of remaining arms at the begining of l-th iteration, rather than discarding the worst half. In addition, we will stop when only K arms remain.

To simplify our notation, we enumerate the arms according to their true mean, i.e., $p_1 > p_2 > \cdots > p_n$. Let $T^* = \{1, 2, \cdots, K\}$ be the set of best K arms.[3] For any set T consisting of K arms, we say that T is ϵ-**optimal** if $\tilde{p}_{T^*} - \tilde{p}_T \leq \epsilon$, where $\tilde{p}_S$ is the K-th highest true mean among the arms in S. We also define the **suboptimality** of T to be $\tilde{p}_{T^*} - \tilde{p}_T$.

Given ϵ and δ provided by the user, BOUNDEDME's goal is to identify a set of K arms that is ϵ-optimal with probability at least $1 - \delta$, using as few pulls as possible.

We use the following function to simplify our presentation:

$$m(u) = min \left\{ \frac{u+1}{1+\frac{u}{N}}, \frac{u+\frac{u}{N}}{1+\frac{u}{N}} \right\} \tag{2.6}$$

We can see that $m(u) < N$ as long as $u > 0$. We have the following lemma.

Lemma 2.2. *For Algorithm 2.1, at any iteration l, we have* $\mathbb{P}\left[\tilde{p}_{S_l} \leq \tilde{p}_{S_{l+1}} + \epsilon_l\right] \geq 1 - \delta_l$.

Based on the above Lemma A.2, we could get the main theoretical property of Algorithm 2.1.

[3]We use the index i instead of a_i to simplify the notation.

Algorithm 2.1 BOUNDEDME Algorithm (for top-K)

input: $K \geq 1, \epsilon > 0, \delta > 0$, and a set of arms A
output: a set of K arms that is ϵ-optimal with probability $1 - \delta$

set $S_1 = A, \epsilon_1 = \frac{\epsilon}{4}, \delta_1 = \frac{\delta}{2}, l = 1$
set $t_0 = 0$
while $|S_l| > K$ **do**
    set $t_l = m\left(\frac{2}{\epsilon_l^2}\log\left(\frac{2(|S_l|-K)}{\delta_l\left(\left\lfloor\frac{|S_l|-K}{2}\right\rfloor+1\right)}\right)\right)$
    Pull every arm $a \in S_l$ for $t_l - t_{l-1}$ times, and let $\hat{p}_a^l$ denote its empirical mean since the begining of the algorithm
    Find the $\left\lceil\frac{|S_l|-K}{2}\right\rceil$-th value of $\hat{p}_a^l$ in ascending order, and denote it as $\bar{p}_l$
    $S_{l+1} = S_l \setminus \{a \in S_l : \hat{p}_a^l \leq \bar{p}_l\}$ (more precisely, remove $\left\lceil\frac{|S_l|-K}{2}\right\rceil$ arms with the least empirical means thus far)
    $\epsilon_{l+1} = \frac{3}{4}\epsilon_l, \delta_{l+1} = \frac{\delta_l}{2}, l = l + 1$
end while
return S_l

Theorem 2.1. *The* BOUNDEDME *algorithm (Algorithm 2.1) is guaranteed to return ϵ-optimal solution with probability at least $1 - \delta$.*

Note that BOUNDEDME is never slower than the naïve search, which has the $O(nN)$ time complexity:

Corollary 2.2. *For each arm, the number of times it is pulled by Algorithm 2.1 is upper-bounded by N.*

BOUNDEDME's time complexity is also lower than Median Elimination [60], which is $O\left(\frac{n}{\epsilon^2}\log(\frac{1}{\delta})\right)$:

Corollary 2.3. *The time complexity of Algorithm 2.1 is $O\left(\frac{n\sqrt{N}}{\epsilon}\sqrt{\log\left(\frac{1}{\delta}\right)}\right)$.*

Remark 2.1. *When Algorithm 2.1 is applied to the MIPS problem, the above bound indicates that the running time is sublinear in the dimension of vectors, but linear in the size of vector set $\mathcal{S}$. This implies that our approach is especially effective for very high-dimensional data. To mitigate the potential issue of linear dependence on the size of $\mathcal{S}$, we could exploit the geometric structure or similarity among vectors from $\mathcal{S}$. The tradeoff here is that this would now require some preprocessing. For example, we could find the convex hull of the set $\mathcal{S}$ first, and then only focus*

on the set of extreme points that form the convex hull, because the solution of the MIPS problem is guaranteed to always include at least one of these extreme points. Thus, when the number of extreme points is much smaller than the size of $\mathcal{S}$, our algorithm becomes sublinear in the size of $\mathcal{S}$.

2.6 Evaluations

Our evaluations aim to (1) empirically validate the theoretical guarantees of Theorem A.1 and (2) compare our method against the state-of-the-art.

Datasets. Since Theorem A.1 is a worst-case guarantee, we use an adversarily-generated synthetic dataset to verify its correctness. Then, we use both synthetic and real-world datasets to compare our algorithm with several state-of-the-art techniques. For each dataset, we used 10^4 vectors with 10^5 dimensions.

Baselines. We compared our method against the following state-of-the-art methods:

- LSH-MIPS [166, 209], which is a popular method for MIPS. We used the nearest neighbor transformation proposed in [15] and the LSH function, as suggested in [166]. We used the standard amplification procedure, i.e., the final result is an OR-construction of b hyper LSH hash functions and each hyper LSH function is an AND-construction of a random projections.

- GREEDY-MIPS [261], which is a recently proposed method that uses a budget B to control the time complexity of the query time.

- PCA-MIPS [15], which uses the depth of the PCA tree to control the time-precision tradeoff.

Comparison Metrics. We compare different algorithms by varying their parameters in order to explore their tradeoffs between precision and online speedup. Precision is defined as the fraction of true top K solutions in the returned top K solutions. Online speedup of an algorithm is defined as the query time required by the naïve (i.e., exhaustive) search divided by the query time of that algorithm. Recall that, unlike the baselines, our algorithm does not require any preprocessing.

However, we ignore the preprocessing time of the baselines in our comparisons, showing that our algorithm's online speedup is still superior despite the lack of any preprocessing.

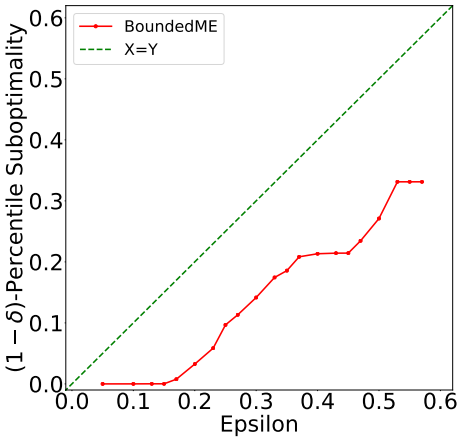
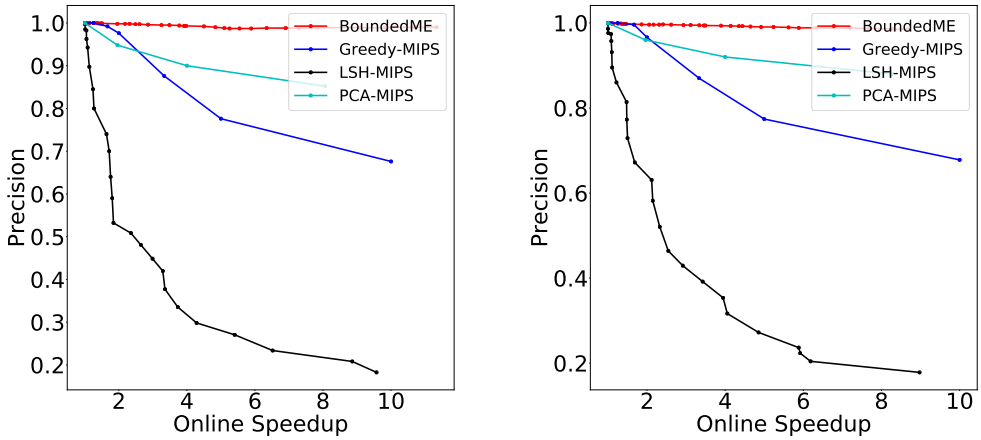


Figure 2.1: Correctness of BOUNDEDME's guarantees

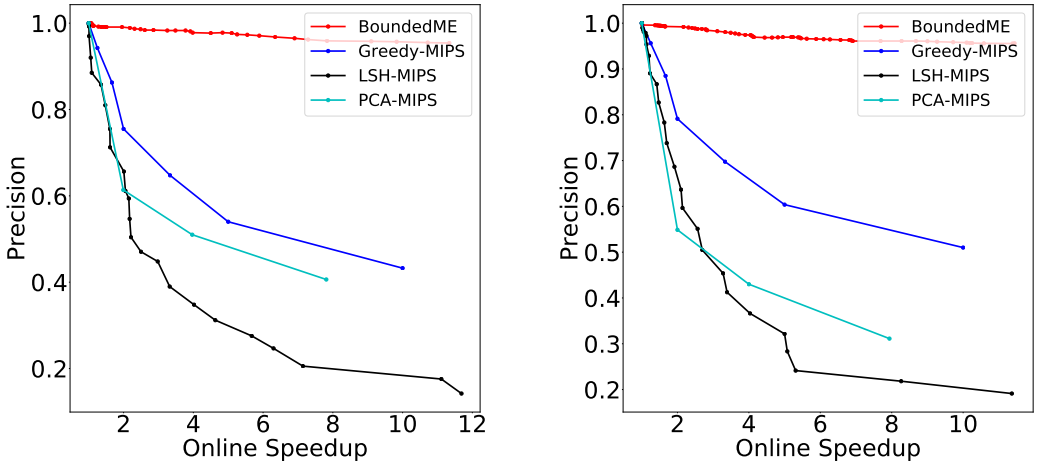


(a) top 5 solutions (b) top 10 solutions

Figure 2.2: Synthetic Gaussian dataset

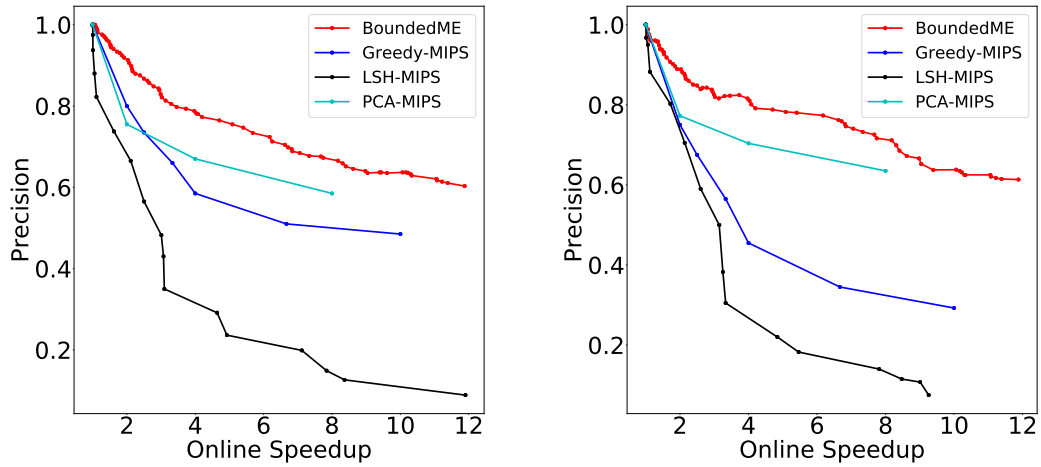
2.6.1 Characteristics of the BOUNDEDME Algorithm

Theorem A.1 provides a PAC bound for BOUNDEDME. In other words, with the δ and ϵ provided by the user, BOUNDEDME is guaranteed to return an ϵ-optimal solution with probability at least



(a) top 5 (b) top 10

Figure 2.3: Synthetic uniform dataset



(a) Netflix Dataset (b) Yahoo Dataset

Figure 2.4: Real-world datases

$1-\delta$. Note that this is a worst-case guarantee, and in most cases we expect the returned solution to be much better than that. Therefore, to empirically validate our worst-case guarantee, we design an adversarial dataset as follows (we will use other realistic datasets in later experiments).

To generate an adversarial dataset, we use 10^4 arm, each with a list of 10^5 reward values. For each arm a, we choose its true mean r_a uniformly at random from $[0, 1]$. Then, the rewards for that arm are generated with every reward being 1 with probability r_a and being 0 with probability $1 - r_a$. When

an arm is pulled—i.e., a sample is drawn from the rewards list without replacement—the rewards with value 1 are returned before those with value 0. This is to make the arms as indistinguishable as possible to the algorithm, thus causing an adversarial scenario.

In this experiment, we vary ϵ between 0 and 0.6. For each value of ϵ, we try all values of δ from the set $\{0.01, 0.05, 0.1, 0.2, 0.3\}$. For each pair of ϵ and δ, we run BOUNDEDME 20 times, each time on a different randomly generated adversarial dataset (as described above). We then measure the $(1-\delta)$-percentile of the list of suboptimalities for each specific pair of ϵ and δ. Figure 2.1 reports the average of these suboptimalities for each value of ϵ. Since the suboptimality is always less than its corresponding value of ϵ, it confirms that they are indeed smaller than their corresponding values of ϵ, i.e., validating Theorem A.1.

2.6.2 BOUNDEDME vs. Other MIPS Algorithms on Synthetic Datasets

We generate two synthetic datasets, where the vector values are drawn from Gaussian and uniform distributions, respectively. For BOUNDEDME, we varied $\epsilon, \delta \in [0, 1]$. For LSH-MIPS, we varied $a \in [1, 20]$ and $b \in [1, 50]$. For GREEDY-MIPS, we varied B from 10% to 100% of the dataset size. For PCA-MIPS, we varied the tree depth in $[0, 20]$. We run experiments for both the cases of returning the top 5 and 10 solutions. As shown in Figures 2.2 and 2.3, when the online speedup is small, all methods have very high precision. However, when online speedup becomes larger, the precisions achieved by other methods start to drop quickly, while BOUNDEDME can still maintain high precision. This demonstrates that BOUNDEDME outperforms these previous methods, despite its lack of preprocessing time.

2.6.3 BOUNDEDME vs. Other MIPS Algorithms on Real-World Datasets

We also compare BOUNDEDME against others on two real-world datasets, Netflix and Yahoo-Music used in [261]. We use the same setting as in [261] to compute the vector embeddings using matrix factorization. The other parameters are the same as previous subsection for the case of identifying the top 5 solutions. Again, as shown in Figure 2.4, the precision of other methods drops

more quickly than that of BOUNDEDME, implying the superior performance of BOUNDEDME over its counterparts.

2.7 Summary

In this chapter, we introduced a new bandit setting, Multi-Armed Bandit with Bounded Pulls (MAB-BP), where the rewards are sampled *without* replacement from a *finite* list. We showed that this setting can be used for solving important problems, such as Maximum Inner Product Search and Nearest Neighbor Search. We also proposed a new algorithm, BOUNDEDME, which extends the Median Elimination framework for MAB-BP settings. Using a new concentration inequality for finite lists, we derived BOUNDEDME's suboptimality guarantee and sample complexity. By applying BOUNDEDME to MIPS, we improved on state-of-the-art methods for MIPS by (1) avoiding their preprocessing step, and (2) offering a knob to the user to directly control the suboptimaltiy of the results. We also conducted extensive experiments on both synthetic and real-world datasets, showing significant speedups over state-of-the-art MIPS algorithms.

Chapter 3

Gradient Sparsification for Distributed Learning

3.1 Motivation

Distributed learning has drawn much attention as an important approach to scale up machine learning models because multiple machines are used to collectively train a model. However, due to the need for communication among these machines, distributed learning can suffer from huge communication cost. To alleviate this issue, gradient sparsification methods have been proposed to reduce the communication cost by only communicating some important coordinates of the gradient vectors. In this chapter, we propose an bandit-based gradient sparsification method.

Most distributed learning methods are focused on parallelizing *stochastic gradient descent* (SGD) algorithms [102, 201, 205]. This is because SGD (and its variants) are commonly used to train machine learning models, including deep learning models [56, 113, 221]. In SGD, let $f(x)$ be the loss function that we hope to minimize, where $x \in \mathbb{R}^D$ is the parameter vector of a machine learning model. At each iteration t, we have access to the unbiased gradient vector $g_t = g(x_t)$ such that $\mathbb{E}[g(x_t)] = \nabla f(x_t)$, where x_t is the parameter vector at current iteration. A standard SGD uses the following update rule to get x_{t+1} for the next iteraiton: $x_{t+1} = x_t - \alpha_t g_t$, where α_t is a scalar for step size.

A common method to parallelize SGD is to use several machines to collectively minimize f. Each machine has one partition of the whole training dataset, and maintains a local copy of the parameter vector x_t. At each iteration, it obtains a new gradient vector g_t according to its local parameter copy and data partition. After each iteration, all machines broadcast their local gradient vectors to their peers, so that gradient vectors can be aggregated to compute the new parameter vector x_{t+1}. This synchronization step of aggregating gradients can also be relaxed with asynchronous update methods [124, 140, 167, 182]. It has been observed that gradient communication in each iteration is a significant performance bottleneck in practical applications [34, 203, 218].

Existing Approaches. The main mechanism for alleviating the communication overhead is to compress the gradients before synchronously[1] broadcasting them to peer machines. There are two popular approaches to compressing the gradients: one is to quantize gradients (into lower precision representation) [8, 21, 48, 49, 50, 181, 246, 264, 268], and the other is to sparsify gradients by dropping some of their coordinates [5, 30, 123, 137, 154, 184, 217, 231]. However, both approaches must compute the gradients first and then apply their compression. Specifically, assume a gradient vector $g = [g_1, \cdots, g_d, \cdots, g_D]$, where the d-th coordinate g_d is the gradient value with respect to the d-th model parameter and D is the number of model parameters. The compressed gradient $Q(g)$ is usually obtained by compressing g according to a probability vector $p = [p_1, \cdots, p_d, \cdots, p_D]$. The variance of $Q(g)$ is defined as $\mathbb{E}\sum_{d=1}^{D}[Q(g)_d^2]$, and the sparsity of $Q(g)$ is defined as $\mathbb{E}\left[\|Q(g)\|_0\right]$, where $Q(g)_d$ is the d-th coordinate of $Q(g)$. Different methods differ in the way they choose p. For example, a popular technique, called TERNGRAD [246], chooses each p_d to be proportional to the absolute value of g_d, i.e., $p_d = \frac{|g_d|}{\|g\|_\infty}$, and defines $Q(g)$ as $Q(g)_d = \|g\|_\infty \operatorname{sign}(g_d) Z_d$, where Z_d is a Bernoulli random variable with $\mathbb{P}(Z_d = 1) = p_d = |g_d|/\|g\|_\infty$. Since $\|g\|_\infty$ is the same for all coordinates, TERNGRAD effectively quantizes each coordinate into one of the $\{-1, 0, 1\}$ values. For TERNGRAD, the variance of $Q(g)$ is $\|g\|_1\|g\|_\infty$, and the sparsity of $Q(g)$ is $\frac{\|g\|_1}{\|g\|_\infty}$, both of which are fixed for any given g. Other techniques, such as GSPAR [245], provide the flexibility to trade off between variance and sparsity by choosing p through solving a linear problem:

[1] State-of-the-art gradient compression often uses synchronous updates. We will also discuss asynchronous updates in Section 3.2.

$min_p \sum_{d=1}^{D} p_d$ s.t. $\sum_{d=1}^{D} \frac{g_d^2}{p_d} \leq (1+\epsilon) \sum_{d=1}^{D} g_d^2$, where ϵ is the parameter that allows the user to control this tradeoff. There are other techniques too, that apply SVD decomposition to g [240], consider multiple-level quantization [8], use additional memory to compensate for the sparsification error [217], or extend to decentralized settings where machines communicate only with their neighbors [223, 237].

These gradient compression methods are quite effective at compressing the size of the gradients, and thus reducing the *communication time* (i.e., the time spent on communicating the gradients with other machines over the network) in traditional distributed learning. Recently, there has been a surge in using large batch optimization to fully utilize the compute power, particularly for training deep neural networks on large scale datasets [75, 78, 108, 136, 212]. It has been shown that using larger batch sizes with synchronous SGD variants can significantly speed up the training [6, 78, 258, 259]. The *computation time* (i.e., time spent on computing the gradients that will be sent over the network) is no longer negligible due to large batch sizes. However, existing gradient compression methods cannot reduce the computation time. This is because the computation time is mainly determined by how the gradients are computed based on the batch of examples in each iteration. In fact, the computation time slightly increases due to the additional operations of compressing and uncompressing the gradients. This is why in many real life settings, the computation time is reported to be much larger than the communication time [6]. For example, as shown in Table 3.1, when training AlexNet [120] on the ImageNet dataset [52] with existing gradient compression methods (e.g., TERNGRAD and GSPAR), the computation time is at least twice the communication time. Therefore, to achieve the best of both worlds and reduce the total training time, we need to reduce the computation while retaining the benefits of gradient compression techniques.

Our Approach. In this chapter, we propose a new gradient compression method, JOINTSPAR, that aims to additionally reduce the computation time while keeping the goal of reducing the communication time. Our key insight is that our method can take advantage of the redundancy in gradient computation in existing methods. Specifically, all existing methods need to compute all coordinates in a gradient vector at each iteration before sparsifying it, even though some

Table 3.1: Breakdown of computation time (t_{comp}) versus communication time (t_{comm}) per epoch (in seconds) for vanilla SGD versus gradient compression methods in large batch optimization. The results are obtained by training AlexNet on ImageNet dataset using 8 machines with P100 GPUs connected with a 100Gb/s InfiniBand fabric (batch size is set as a large value, i.e., 1024 for each machine). While gradient compression methods (e.g., TERNGRAD and GSPAR) reduce communication time by 2-3x, the computation time becomes the dominant part. Note that the slight increase in computation time of the gradient compression methods is due to time spent compressing and uncompressing gradients.

Method	t_{comp}	t_{comm}	t_{comp}/t_{comm}
SGD	1021	1156	0.88
TERNGRAD	1205	446	2.70
GSPAR	1079	519	2.08

coordinates might be dropped by sparsification. The training time could be further reduced if only the coordinates that would be communicated were computed. JOINTSPAR integrates sparsity in the gradient computation, thereby avoiding redundant gradient computation and achieving joint sparsification of gradient computation and communication.

Specifically, rather than recomputing the probability vector at each iteration, we maintain a probability distribution p over all coordinates which is being updated across training iterations. The optimal solution to the probability distribution at each iteration requires us to compute the gradient value for each coordinate, which is exactly what we aim to avoid in the first place. We cast the problem of learning an optimal probability distribution as an adversarial multi-armed bandit problem. We use a multi-armed bandit method to update the distribution p, based on the partial information available at each iteration (i.e., the gradient values for coordinates that will not be dropped). The bandit method for learning p is embedded into the parallelized SGD that minimizes the loss function with respect to x, so that their iterations are synchronized. At each iteration, JOINTSPAR performs one update step for parallelized SGD with respect to x, and one update step for the bandit method with respect to p. The gradient coordinates are chosen according to the current value of p at the beginning of every iteration, and only the chosen coordinates are computed and communicated. This joint sparsification of gradient computation and communication greatly reduces the end-to-end training time of JOINTSPAR. In addition, JOINTSPAR also provides users with the flexibility to choose the sparsity budget. As with GSPAR, this can be used to trade off variance and

sparsity. We prove that JOINTSPAR achieves the same convergence rate in iteration numbers as existing state-of-the-art methods for communication-efficient distributed learning [21, 223]. We empirically verify that, compared to state-of-the-art methods, JOINTSPAR converges faster in terms of wall-clock time, because it requires less time per iteration.

In summary, we make the following contributions:

- We observe that, for large batch optimization, the computation time is no longer negligible, responsible for a significant fraction of their total training time. We propose new gradient compression method for large batch optimization, JOINTSPAR and its variant JOINTSPAR-LARS, which jointly sparsify both computation and communication of the gradients. Similar to state-of-the-art [240, 245], our methods provide users the flexibility to trade off variance and sparsity.

- We prove that JOINTSPAR has the same convergence rate in terms of *the number of iterations* as the state-of-the-art methods for SGD-based distributed learning [21, 223], namely $O(\frac{1}{\sqrt{T}})$, where T is the number of iterations. We also prove that when applied to recently proposed large batch optimizer with layerwise adaptive learning rates LARS, the variant JOINTSPAR-LARS enjoys the same convergence rate as the original LARS.

- Our extensive experiments on several benchmark datasets (e.g., MNIST, CIAF10, CIFAR100 and ImageNet) using various models (e.g., convolutional neural networks, and residual neural networks) show that our methods spend significantly less time per iteration, and converge faster than state-of-the-art methods in terms of *wall-clock time*.

3.2 Related Work

Training Efficiency. For efficient model training, especially in the single machine setting, weight compression techniques whereby the precision of the weights are reduced to save compute and memory overheads have been shown very effective [157, 193]. This is because memory usage

Table 3.2: Notation summary.

Symbol	Description	Symbol	Description
M	the total number of machines/processes	p_t^m	probability vector for machine m at iteration t
f	loss function	$p_{t,d}^m$	d-th coordinate of probability vector p_t^m
x	parameter vector	Z_t^m	selector vector for machine m at iteration t
$\mathcal{X}$	space of the parameter vector	S_t^m	active set for machine m at iteration t
D	number of blocks of the parameter vector	$Q(g_t^m)$	sparsified version of g_t^m
T	total number of training iterations	$Q(g_t^m)_d$	d-th block of $Q(g_t^m)$
g_t^m	full gradient vector for machine m at iteration t	s	sparsity budget, i.e., $\mathbb{E}\left[\|\|Q(g_t^m)\|\|_0\right]$
$g_{t,d}^m$	d-th block of gradient vector g_t^m	S_d	smoothness constant for x_d
$g_{t,[d]}^m$	zero vector with $g_{t,d}^m$ at d-th block	L	upper bound on $\|\|g_{t,[d]}^m\|\|, \forall t, d, m$

can be decreased by using fewer bits to store the same number of values, and compute time can also be reduced on processors that offer higher throughput for reduced precision arithmetic operations. For example, recent work has successfully trained deep neural networks using 8-bit floating point numbers without sacrificing accuracy [241]. Another line of work has shown that sampling techniques can be used to accelerate model training [106, 266]. For example, Adam with bandit sampling is proved to converge faster than the original Adam when the feature distribution in the training set is skewed (e.g., exponentially faster in terms of the size of the training set when the feature follows a doubly heavy-tailed distribution) [146]. The key idea is that, by sampling those important examples more often, the training process can be accelerated due to improved sample efficiency. Recent works [37, 142] have shown that improving sample efficiency is also useful for efficiently training transformer-based language models [53]. Our focus in this chapter is on efficient model training in a distributed setting through reducing both computation and communication costs.

Communication Efficiency. In the distributed setting where we use multiple machines to collectively train a model, frequent communication happens among these machines to exchange information that is local at each machine. The most studied type of communication is aggregating local gradients to get the global gradient which is needed for updating the model parameters at each training iteration, because distributed training in the data parallel fashion is widely adopted [130]. In this chapter, we also focus on this type of communication (i.e., aggregating local gradients). There has been significant progress on using various compression techniques to reduce the communication cost. The convergence rate of these methods under different assumptions has been studied in the literature [9, 33, 116, 137, 149, 224, 225, 236, 249, 254, 262, 267]. Some specialized systems have also

been developed that can support efficient communication for gradient aggregation based on various compression methods [65, 131, 185]. In addition to gradient compression methods, asynchronous updates provide an alternative means of alleviating the communication overhead [124, 134, 182]. Asynchronous updates are particularly useful in heterogeneous clusters, where some machines are considerably slower than others. These relatively slow machines are called stragglers. Asynchronous updates can introduce additional noise if there are too many stragglers. To avoid asynchronous noise and mitigate stragglers, an alternative solution is to use synchronous updates with backup workers [31] proposed to use. Although asynchronous updates could in principle be combined with gradient compression, state-of-the-art gradient compression methods still use synchronous updates [8, 21, 240, 246]. This is primarily to avoid the additional asynchronous noise, especially that stragglers are not a major concern. Thus, to be consistent with state-of-the-art gradient compression methods, we also focus on synchronous updates in the rest of this chapter. It is worth noting that there are other types of communication, such as passing neural activation values across machines due to model parallelism [81, 147, 207]. Other techniques such as a dropout variant called Gating Dropout [147] have been proposed to reduce the cost of other communication types.

3.3 Joint Sparsification of Gradient Computation and Communication

Consider the following nonconvex [2] optimization problem: $\min_{x \in \mathcal{X}} f(x)$, where $\mathcal{X}$ is the parameter space. The goal in distributed learning is to solve this problem using M machines/processes[3], where each machine stores a partition of the entire training dataset. Without loss of generality, assume parameter vector x can be decomposed into D blocks, i.e., $x = [x_1, \cdots, x_d, \cdots, x_D]$ where D is the total number of parameter blocks. The reason that this is without loss of generality is because, in neural network models, their layerwise architecture provides a natural block

[2] Due to space constraint, here we focus on nonconvex objective functions, which are the case for neural network models. Please refer to the appendix for our results on a tigher bound on convex objective functions.

[3] We use machine and process interchangeably in this chapter.

decomposition, i.e., all the paramaters from one layer form a block. For the rest of this chapter, we consider this layerwise formulation rather than the coordinate-based formulation, because (1) it is to be consistent with the recently proposed large batch optimizers with layerwise adaptive learning rates [258, 259] (see Section 3.3.2), (2) it allows efficient implementation on top of modern deep learning frameworks such as PyTorch (see Section 3.3.3). Suppose the full gradient of f with respect to x for machine m at iteration t is g_t^m, and $g_{t,d}^m$ is the gradient of f with respect to x_d. Denote $g_{t,[d]}^m = [0, \cdots, g_{t,d}^m, \cdots, 0]$, where all entries are zero except $g_{t,d}^m$ at the d-th block. Thus, $g_t^m = \sum_{d=1}^D g_{t,[d]}^m$.

Our goal in this chapter is to compress g_t^m by integrating sparsity in the gradient computation. To this end, we maintain a probability distribution $p_t^m = [p_{t,1}^m, \cdots, p_{t,d}^m, \cdots, p_{t,D}^m]$ over the gradient blocks. To sparsify (i.e., compress) the gradient, we keep $g_{t,d}^m$ with probability $p_{t,d}^m$, and drop it with probability $1 - p_{t,d}^m$. The sparsified gradient vector is then represented as $Q(g_t^m) = \sum_{d=1}^D \frac{g_{t,[d]}^m}{p_{t,d}^m} Z_{t,d}^m$, where $Z_{t,d}^m$ is a Bernoulli random variable with $\mathbb{P}(Z_{t,d}^m = 1) = p_{t,d}^m$, for all $1 \leq d \leq D$. We can verify that $Q(g_t^m)$ is unbiased: $\mathbb{E}[Q(g_t^m)_d] = \frac{g_{t,d}^m}{p_{t,d}^m} \mathbb{E}[Z_{t,d}^m] = \frac{g_{t,d}^m}{p_{t,d}^m} p_{t,d}^m = g_{t,d}^m$.

To avoid redundant gradient computations, at the beginning of each iteration, we instantiate the random variable $Z_{t,d}^m$ by setting $Z_{t,d}^m = 1$ with probability $p_{t,d}^m$ and $Z_{t,d}^m = 0$ with probability $1 - p_{t,d}^m$. Since we will drop any gradient block d for which $Z_{t,d}^m = 0$ during the communication, we also skip computing these dropped blocks. In other words, we only compute $g_{t,d}^m$ where $d \in S_t^m = \{d : Z_{t,d} = 1, \forall 1 \leq d \leq D\}$. We call the set S_t^m the active set as it indicates the gradient blocks that we will actively compute and communicate. The key notations used in this chapter are listed in Table 3.2. Because gradient computation accounts for a significant portion of the time spent in each iteration, computing only the necessary gradient blocks will be hugely advantageous. For our analysis, we make the following common assumptions [19, 47, 259].

Assumption 3.1. (1) The objective function f is S_d-smooth with respect to x_d; (2) The gradient $g_{t,[d]}^m$ is upper bounded by L, i.e., $L \geq \|g_{t,[d]}^m\|, \forall t, d, m$; (3) For all $x, y \in \mathcal{X}$, we have $B(x, y) \leq R^2$, where $B(x, y)$ is the Bregman divergence between x and y.

We use $S = [S_1, S_2, \cdots, S_D]$ to denote the D-dimensional vector of Lipschitz smoothness

constants. Let $S_\infty = \max_d S_d$. For simplicity, we also assume the function value at the initial point is upper bounded by R^2, i.e., $f(x_1) \leq R^2$. The probability distribution p_t^m will govern how many gradient blocks are computed and communicated. A good choice of p_t^m should place high probability values on those blocks that have the largest impact on the ultimate convergence of the model. Before we discuss how to determine $p_{t,d}^m$, let us assume that there is an oracle that provides some arbitrary value for $p_{t,d}^m$. We call this ideal method as ORACLESPAR. Its pseudocode is the same as Algorithm 3.2 except that line 9 should be replaced by using an oracle to get $p_{t+1,d}^m$. Then, we have the following theorem[4] regarding its convergence rate.

Theorem 3.1. *Under Assumption 3.1,* ORACLESPAR *achieves*

$$\begin{aligned} &\mathbb{E}\sum_{t=1}^{T}\sum_{d=1}^{D}\|f_d'(x_t)\|^2 \\ \leq &\frac{R^2}{\alpha_x} + \frac{S_\infty \alpha_x}{2M^2}\sum_{m=1}^{M}\sum_{t=1}^{T}\mathbb{E}\left[\sum_{d=1}^{D}\frac{\|g_{t,[d]}^m\|^2}{p_{t,d}^m}\right] \end{aligned} \tag{3.1}$$

where $f'(x_t) = [f_1'(x_t), \cdots, f_d'(x_t), \cdots, f_D'(x_t)]$ *is the gradient of* f *with respect to* x_t *and* α_x *is the step size.*

According to Theorem 3.1, the second term on the right hand side of Equation 3.1 depends on the values of probabilities $p_{t,d}^m$. For a fast convergence rate, we wish to choose the $p_{t,d}^m$ values such that the second term is minimized. Interestingly, the second term is the same as the gradient variance in existing methods [240, 245]. It can be shown that, for every iteration t, the optimal distribution $p_{t,[d]}^m$ is proportional to the gradient value of individual coordinate [7, 164]. Formally speaking, for any t, the optimal solution to the problem

$$arg \min_{\sum_{d=1}^{D} p_{t,d}=1} \sum_{d=1}^{D} \frac{\|g_{t,[d]}\|^2}{p_{t,d}} \tag{3.2}$$

is $p_{t,d} = \frac{\|g_{t,[d]}\|}{\sum_{d=1}^{D}\|g_{t,[d]}\|}, \forall d$. However, it is computationally prohibitive to get the optimal solution, because we need to compute the gradient values for all blocks (thus all coordinates), which is exactly

[4]All omitted proofs can be found in the appendix.

what we aim to avoid in the first place. Instead, we use a multi-armed bandit method [46, 156] to learn this distribution from the partial information that is available during training. Here, partial information is the gradient values for blocks in the active set S_t^m. The multi-armed bandit method maintains the distribution over all blocks, and keeps updating this distribution at every training iteration.

3.3.1 Bandit Method for Distribution Learning

Algorithm 3.1 Distribution update for p_t^m

Function: update $(p_t^m, S_t^m, \{g_{t,[d]}^m\}_{d \in S_t^m})$
  for $d \leftarrow 1$ **to** D **do**
    if $d \in S_t^m$ **then**
      $\tilde{l}_{t,d}^m = -\frac{\|g_{t,[d]}^m\|^2}{(p_{t,d}^m)^2} + \frac{L^2}{p_{min}^2}$;
    else
      $\tilde{l}_{t,d}^m = 0$;
    end if
    $w_{t,d}^m = p_{t,d}^m \exp(-\alpha_p \tilde{l}_{t,d}^m / p_{t,d}^m)$;
  end for
  $p_{t+1}^m = argmin_{q \in \mathcal{P}} D_{kl}(q \| w_t^m)$;
  Return: p_{t+1}^m

Our goal is to minimize $\sum_{t=1}^T \mathbb{E}\left[\sum_{d=1}^D \frac{\|g_{t,[d]}^m\|^2}{p_{t,d}^m}\right]$ by using a multi-armed bandit method on each machine m. We use a bandit method based on EXP3 [13] but extended to handle multiple actions at every iteration. The pseudocode is described in Algorithm 3.1. Note that L is assumed to be the upper bound on the gradient norm (see Assumption 3.1). In addition, we define the set $\mathcal{P}$ as $\mathcal{P} = \{p \in \mathbb{R}^D : \sum_{d=1}^D p_d = s, p_d \geq p_{min}, \forall 1 \leq d \leq D\}$, where s is the *sparsity budget* and p_{min} is a constant lower bound. Both s and p_{min} are parameters directly controlled by the user, where $0 < s \leq D$ and $0 < p_{min} \leq 1$. The sparsity budget s serves as a knob for the user to gradually

transition from no gradient computation (i.e., when $s = 0$) to full gradient computation (i.e., when $s = D$). In line 10, $D_{kl}(q\|w)$ is the Kullback-Leibler divergence between q and w.

To further illustrate the distribution update algorithm from the perspective of the bandit setting, the number of arms is D, where each arm corresponds to a parameter block. Selecting an active set S_t^m of blocks for which gradient computation and communication will be performed, is the same as choosing $|S_t^m|$ arms to pull. At iteration t, the loss of pulling the arm d is $\tilde{l}_{t,d}^m$, which is defined in line 4 of Algorithm 3.1. Each time we update the distribution, we only pull these arms from the set S_t^m, which are blocks for which gradient computation and communication will be performed. From the definition of $\tilde{l}_{t,d}^m$ as in line 4 of Algorithm 3.1, we can see that the loss $\tilde{l}_{t,d}^m$ is always nonnegative, and is inversely correlated with $\|g_{t,[d]}^m\|$. This implies that a block with small gradient norm will receive large loss value, resulting in its weight getting decreased. In other words, blocks with larger gradient norms are more likely to be selected in the active set. This aligns with existing gradient compression methods in the sense that blocks with small gradient norms tend to get dropped to reduce the communication cost. Using a sum tree structure to store these weights, we can efficiently update p_t^m in $O(|S_t^m| \log D)$ time [162]. With the logarithm dependence on D, the overhead of the bandit method is small even for a large neural network model.

Theorem 3.2. *Under Assumption 3.1, if we set the step size* $\alpha_p = \sqrt{\frac{2R^2 p_{min}^4}{DTL^4}}$*, Algorithm 3.1 achieves the following convergence rate:*

$$\mathbb{E}\sum_{t=1}^{T} l(p_t^m) - \min_{p\in\mathcal{P}} \mathbb{E}\sum_{t=1}^{T} l(p) \leq \frac{RL^2}{p_{min}^2}\sqrt{2DT} \tag{3.3}$$

where $l(p_t^m) = \sum_{d=1}^{D} \frac{\|g_{t,[d]}^m\|^2}{p_{t,d}^m}$.

Using a bandit method to keep updating the distribution p_t^m (which is initialized uniformly, i.e., $p_1^m = \frac{s}{D}$) as in Algorithm 3.1, we can now introduce our distributed learning algorithm that jointly sparsifies gradient computation and communication. We call our algorithm JOINTSPAR, which is presented in Algorithm 3.2.

Combining both Theorem 3.1 and 3.2, we can prove the convergence rate of JOINTSPAR, as

Algorithm 3.2 Our distributed learning method JOINTSPAR (for each worker m)

for $t = 1, \cdots, T$ **do**
  Instantiate the value for $Z_{t,d}^m$, where $Z_{t,d}^m$ is a Bernoulli random variable with $\mathbb{P}(Z_{t,d}^m = 1) = p_{t,d}^m$, for all $1 \leq d \leq D$;
  Determine active set $S_t^m = \{d : Z_{t,d} = 1, \forall 1 \leq d \leq D\}$;
  Compute gradient $g_{t,d}^m$ for all $d \in S_t^m$;
  Correct the gradients to make them unbiased with $\hat{g}_{t,d}^m = \frac{g_{t,d}^m}{p_{t,d}^m}$;
  Construct sparsified gradient $Q(g_t^m) = [q_1, \cdots, q_D]$, where $q_d = \hat{g}_{t,d}^m$ if $d \in S_t^m$, otherwise $q_d = 0$;
  Encode $Q(g_t^m)$ in sparse format and broadcast it to other peer machines;
  Receive $Q(g_t^{m'})$ from peer machines $1 \leq m' \leq M$, and update the parameter vector $x_{t+1} = x_t - \alpha_x \frac{1}{M} \sum_{m'=1}^{M} Q(g_t^{m'})$;
  $p_{t+1,d}^m = \text{update}\,(p_t^m, S_t^m, \{g_{t,[d]}^m\}_{d \in S_t^m})$;
end for

stated in Theorem 3.3.

Theorem 3.3. *Under Assumption 3.1,* JOINTSPAR *achieves*

$$\sum_{t=1}^{T} \sum_{d=1}^{D} \mathbb{E} \|f_d'(x_t)\|^2 \leq RLD \sqrt{\frac{S_\infty T}{2Ms}} + \frac{R}{LD} \sqrt{\frac{S_\infty s}{2MT}} \bar{R}^* + \frac{R^2 L}{p_{min}^2} \sqrt{\frac{S_\infty s}{MD}} \tag{3.4}$$

where

$$\bar{R}^* = \max_m \min_{p \in \mathcal{P}} \mathbb{E} \left[\sum_{t=1}^{T} \sum_{d=1}^{D} \frac{\|g_{t,[d]}^m\|^2}{p_d} \right]. \tag{3.5}$$

We now further bound Equation 3.5 to show that our convergence rate is $O(\frac{1}{\sqrt{T}})$. Since we have assumed the upper bound on gradient $L \geq \|g_{t,[d]}^m\|, \forall t, d, m$, we could get the following upper bound on $\bar{R}^*$.

Lemma 3.1. *Under Assumption 3.1, then we have*

$$\bar{R}^* = \max_m \min_{p \in \mathcal{P}} \mathbb{E} \left[\sum_{t=1}^{T} \sum_{d=1}^{D} \frac{\|g_{t,[d]}^m\|^2}{p_d} \right] \leq O(\frac{TL^2 D^2}{s}). \tag{3.6}$$

Theorem 3.4. *Under Assumption 3.1, our distributed learning method* JOINTSPAR *has the following convergence rate*

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{d=1}^{D} \mathbb{E} \|f_d'(x_t)\|^2 \leq O\left(RLD \sqrt{\frac{2S_\infty}{MsT}} \right) \tag{3.7}$$

where f is the convex objective function and s is the sparsity budget .

This theorem implies that the average convergence rate of JOINTSPAR is $O(\frac{1}{\sqrt{T}})$, which is the same as the state-of-the-art methods for distributed learning [21, 223]. As we increase the sparsity budget s, the right hand side of Equation 3.7 becomes smaller, implying a faster convergence rate. This is expected, as a larger value of s allows our method to communicate gradient vectors that are less sparsified, hence carrying more information at each iteration of the gradient communication. On the other hand, computing and communicating less sparsified gradient vectors will be more expensive in practice in terms of wall-clock time. This tradeoff can be directly controlled by the user, via setting the s value.

3.3.2 Large Batch Optimizer Variant with Layerwise Adaptive Learning Rates

Our method JOINTSPAR as shown in Algorithm 3.2 is based on synchronous SGD. Combined with some tricks such as linear scaling and warmup, synchronous SGD was able to drastically reduce the training time of ImageNet training with ResNet-50 from 29 hours to 1 hour using a large batch size (up to 8192) [78] . To further increase the batch size without compromising the test accuracy, layerwise adaptive learning rates have been recently introduced for large batch optimizers. The most prominent optimizers are LARS and LAMB [258, 259]. In this section, we dicuss a variant of JOINTSPAR by applying our bandit-based compression method to LARS and show that the same convergence rate as with the original LARS can still be achieved [5]. We call this variant JOINTSPAR-LARS. Its pseudocode is the same with Algorithm 3.2, except that the parameter update rule in line 8 becomes $x_{t+1,d} = x_{t,d} - \alpha_x \frac{\phi\|x_{t,d}\|}{\|m_{t,d}\|} m_{t,d}$ for $1 \leq d \leq D$. Here, momentum $m_{t,d}$ is updated by $m_{t,d} = \beta_1 m_{t-1,d} + (1-\beta_1)\left(\frac{1}{M}\sum_{m'=1}^{M} Q(g_t^{m'})\right)$, where initial momentum $m_0 = 0$ and β_1 is a hyperparameter taking value in $(0, 1)$. $\phi(\cdot)$ is a rescaling function. As with [259], we assume $\phi(z) = \min\{\max\{z, \alpha_l\}, \alpha_u\}$, where α_l and α_u are constant values. We have the following

[5] Due to space limit, we omit discussion on LAMB. But similar analysis can be extended to LAMB.

theorem regarding the convergence rate of JOINTSPAR-LARS[6].

Theorem 3.5. *Under Assumption 3.1, let* $\alpha_x = \sqrt{\frac{2(f(x_1)-f(x^*))}{\alpha_u^2 \|S\|_1 T}}$*, batch size* $B = T$*. Then for* x_t *generated using* JOINTSPAR-LARS*, we have the following bound*

$$\left(\mathbb{E}\left[\frac{1}{\sqrt{D}}\sum_{d=1}^{D}\|f'_d(x_a)\|\right]\right)^2 \leq O\left(\frac{(f(x_1)-f(x^*))\|S\|_1}{TD} + \frac{L^2D^2}{MsT}\right), \tag{3.8}$$

where $x^* = argmin_{x\in\mathcal{X}} f(x)$ *and* x_a *is an iterate uniformly randomly chosen from* $\{x_1, \cdots, x_T\}$.

This theorem implies that the convergence rate of JOINTSPAR-LARS is $O(\frac{1}{T})$ in term of *the number of iterations*, which is the same as the original LARS (see Theorem 2 in [259]). We empirically show that JOINTSPAR-LARS has a faster convergence rate than existing methods in terms of *wall-clock time*.

3.3.3 Implementation Details

We discuss how our methods are efficiently implemented on top of PyTorch. We treat each parameter tensor from the PyTorch framework as a parameter block in our implementation. This could also help us achieve highly efficient execution on GPU. Specifically, to skip gradient computation for a parameter block, we use the PyTorch API: `param_tensor.requires_grad_(False)` to turn off the gradient computation for its corresponding tensor `param_tensor`. For example, the `param_tensor` could be the kernel matrix of some convolutional layer, the bias vector at some linear layer and so on. This eliminates the need to break the matrix computation on GPUs while we still benefit from skipping some gradient computations. Note that the gradient is turned off (i.e., `requires_grad_(False)`) before *both* the forward pass as well as the backward pass. Because of the way auto-differentiation works in PyTorch [172], backpropagation will stop at some intermediate layer if the gradients for all previous layers have been turned off. This has often been the case across all our experiments, which certainly does save a lot more than half of the

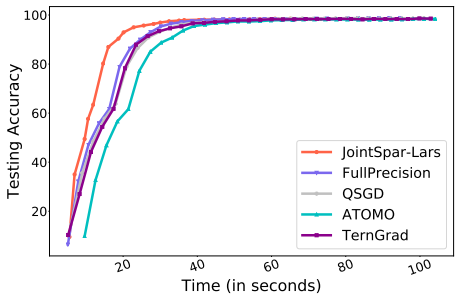
[6]As with [259], we analyze the case where $\beta_1 = 0$.

computation during backpropagation. However, turning off the gradient compuation on the whole weight matrix on linear layers could greatly slow down the training process. To gain finer-grained control over gradient computation for linear layers, we implemented a new `nn.Linear` layer that splits each column of its original weight matrix into an independent submatrix. In this way, the gradient computation of each column can be independently turned off.

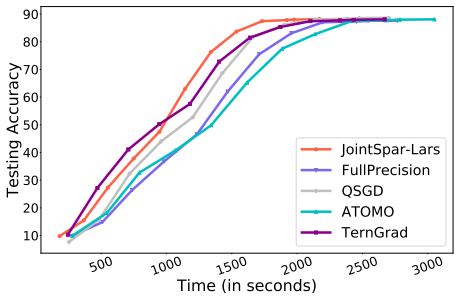
Modern deep learning frameworks, such as PyTorch, support overlapping communication and computation by sending gradients as soon as they are available [131]. This can accelerate training by avoiding the additional delay of waiting for all the gradients. To take advantage of the overlapping characteristic, many gradient compression methods are implemented in the layer-wise way, i.e., the compression is independently applied to the gradient vector of each layer [57]. Although our compression methods are not layer-wise, they can still enjoy the acceleration benefits provided by the overlapping characteristic. Simply speaking, this is because our methods still allow gradient vector of each layer to be sent as soon as it is computed. More specifically, at each iteration, our methods decide which layers should not send their gradient vectors even before the forward pass starts. We instantiate this decision in PyTorch via setting `param_tensor.requires_grad_(False)`. Then the procedure of the forward pass and backward pass starts with no involvement from our methods. In other words, each gradient vector is sent as soon as it is ready if overlapping communication and computation is supported by the framework.

3.4 Evaluations

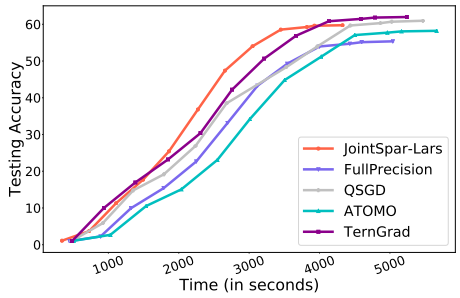
We conduct extensive experiments to evaluate the effectiveness and efficiency of our method. Due to space limit, we focus on JOINTSPAR-LARS because optimizers with layerwise adaptive learning rates are more effective in the large batch setting [258, 259]. More experiments on JOINTSPAR and other SGD-based compression methods can be found in the appendix.



(a) MNIST



(b) CIFAR10



(c) CIFAR100

Figure 3.1: Convergence in terms of wall-clock time, confirming our method JOINTSPAR-LARS's faster convergence than baselines.

3.4.1 Setup

Baselines. We compare our method, JOINTSPAR-LARS against the full precision version of LARS (no compression used) and the following state-of-the-art gradient compression methods for distributed learning: QSGD [8], TERNGRAD [246], and ATOMO [240]. All the gradient compression baseline methods are applied to LARS in our experiments. Note that all of these baseline compression methods must compute all coordinates in a gradient vector before compressing it to reduce communication cost. To the best of our knowledge, our method JOINTSPAR-LARS is the first to jointly sparsify gradient computation and communication.

Datasets and Implementations. We use several benchmark datasets in our experiments: MNIST, Fashion-MNIST, SVHN, CIFAR10, CIFAR100 and ImageNet. Different neural network models are trained on these datasets. We use PyTorch [173] to implement models and learning methods, and use mpi4py [45] as the communication framework in the distributed setting. In traditional distribution learning, the global batch size is usually fixed and evenly divided among multiple machines [8, 240]. In our experiments, however, we fix the local batch size at each machine, which

Table 3.3: Target accuracy when measuring the training time for speedup comparison

Dataset	MNIST	SVHN	Fashion-MNIST	CIFAR10	CIFAR100
Target Accuracy	90%	95%	91%	88%	55%

is the common practice in large batch optimization [78]. All experiments are run on a computer cluster with up to 16 nodes. Each node has 20 physical CPU cores with clock speed up to 4 GHz, and 4 NVIDIA P100 GPUs. Nodes are connected via a 100Gb/s InfiniBand fabric.



Figure 3.2: Speedup of JOINTSPAR-LARS against baseline methods, computed as the ratio of the wall-clock time spent by each baseline to that by JOINTSPAR when achieving the same testing loss.

3.4.2 Convergence Comparison

We compare the end-to-end convergence performance on different datasets and models trained with JOINTSPAR-LARS and all baselines, with respect to the number of epochs and wall-clock time. We use LeNet [125, 246] on MNIST, and ResNet-18 [84] on CIFAR10 and CIFAF100. We use 8 processes to collectively train each model and the local batch size is set to 1024. As with Wang et al. [240], for those methods that allow users to control the sparsity budget, we tune the sparsity budget to choose among $10\%, 30\%, 50\%, 80\%$ of the full gradient size. The convergence rates in terms of the wall-clock time are shown in Figure 3.1, where x-axis and y-axis are the wall-clock time and testing accuracy, respectively. In terms of wall-clock time, Figure 3.1 demonstrates that JOINTSPAR-LARS speeds up the model training significantly, thanks to spending less time per iteration than baselines.

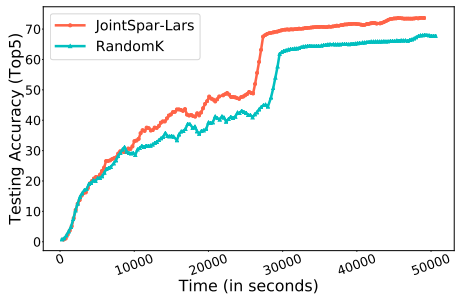


Figure 3.3: Convergence comparison for training AlexNet (with 62 million parameters) on ImageNet dataset. JOINTSPAR-LARS converges faster than baseline methods.

Speedup. To get a better understanding of our speedup against baselines, we compute the ratio of time spent by a baseline method vs. by our method when achieving the same target testing accuracy. We use the testing accuracy achieved by the full precision method at convergence as the target accuracy, as shown in Table 3.3. We report this ratio as speedup of our method against the corresponding baseline method. We use two more datasets: Fashion-MNIST and SVHN, and train the same convolutional neural network model as on MNIST. Figure 3.2 gives a detailed account of speedups of our method against different baseline methods on different datasets. Our method can achieve close to 2x speedup in most cases, and as high as 4x. Note that in some cases such as FullPrecision on SVHN, the speedup is as low as 1.1x. To further reduce the training time, our method could be combined with skipping the forward pass for the layers for which the gradient will be dropped during the communication. The idea of skipping certain layers of neural network models has been explored in prior papers, which can speed up model training without compromising the accuracy [62, 90]. We leave this idea to the future work.

3.4.3 Convergence Comparison on ImageNet

To verify the effectiveness of our method on large models, we also train AlexNet [120] on the ImagetNet dataset [52]. As demonstrated in [78], using large batch size (up to 8k) is very effective at reducing the overall training time on the ImageNet dataset without loss of accuracy. We set the local batch size to 1024 for each machine, and uses the same tricks (i.e., linear scaling and warmup)

as suggested in [78]. Other experimental settings are kept the same as in the previous subsection. From Figure 3.3, we observe that JOINTSPAR-LARS converges faster than the existing compression method RandomK, which randomly remove $K\%$ of the gradient coordinates. We still tune $K\%$ among $10\%, 30\%, 50\%, 80\%$ of the full gradient size. It demonstrates that our method maintains its benefits for large models.

3.5 Summary

We have observed that, in large batch optimization (e.g., large batch SGD and its variant LARS with layerwise adaptive learning rates), the per-machine computation cost is no longer negligible compared to the communication cost. We proposed new gradient sparsification methods for large batch optimization, JOINTSPAR (corresponding to large batch SGD) and its variant JOINTSPAR-LARS (corresponding to LARS), that only compute gradient coordinates that will be communicated. We formally proved that JOINTSPAR and JOINTSPAR-LARS still enjoy the same convergence rates (in terms of number of iterations) as their respective baseline methods. Extensive experiments demonstrated that our methods converge faster than existing methods.

Part II

Deciding Example Selection

Chapter 4

Sample Efficient Transformer with Memory Replay

4.1 Motivation

In this chapter, we consider the transformers models which have achieved state-of-the-art performance on various natural language processing (NLP) tasks, such as sentimental anaylysis, paraphrase detection, machine reading comprehension, text summarization, question answering and so on [25, 44, 53, 150]. The training of tranformers typically consists of two stages: pre-training and fine-tuning. Pre-training is the stage of training a generic model on an enormous corpus, such as Wikipedia to learn the representation inherent in understanding the natural language. Fine-tuning is the stage of training a task-specific model that is initialized with pre-trained parameters on the dataset for the specific downstream task for just a few epochs. Each downstream task has a separate fine-tuned model, even though it is initialized with the same pre-trained parameters. To obtain good generic representation, transformers are usually very large models with a huge number of parameters. For example, OpenAI recently released GPT-3, which contains 175 billion parameters [25]. Training these large-scale models is extremely compute-intensive and has very high sample complexity for the pre-training stage [37, 53]. To make transformers more sample efficient, Clark et al. [37] proposed a new transformer model, called ELECTRA, that consists of two modules: generator

and discriminator. Both generator and discriminator are transformers, although generator is usually smaller in size. The discriminator is trained to predict whether each token in the corrupted input was replaced by the generator. They showed that their model substantially outperforms previous models, such as BERT and XLNet given the same amount of data.

In this chapter, we go one step further by integrating the memory replay mechanism into ELECTRA, which we show can further improve the sample efficiency. Memory replay works by maintaining a fixed-size memory buffer that holds the most recent examples. It greatly improves the sample efficiency by enabling examples to be reused multiple times for training, rather than throwing away examples immediately after one-time usage. By controlling the strategy on how examples are managed in the memory buffer (e.g., how to assign weights to examples), memory replay can be customized according to specific needs. Memory replay mechanism has been successfully used in reinforcement learning [1] and GANs, because of its improvement on sample efficiency (i.e., requires less amount of samples to achieve the same accuracy) [63, 200, 244, 248].

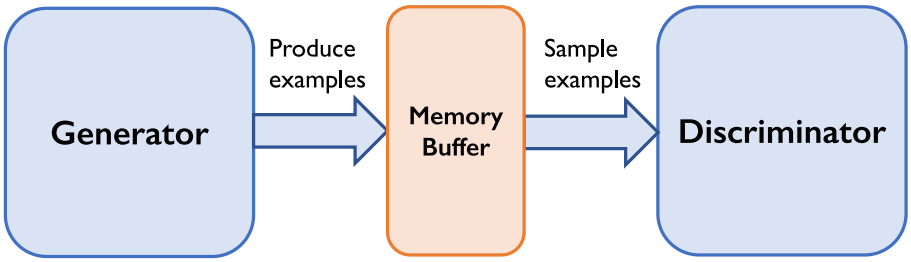


Figure 4.1: The architecture diagram of our *Transformer with Memory Replay* (TMR). The generator produces corrupted examples, which are being saved to the fixed-size memory buffer. The discriminator samples examples from the memory buffer for training. High-quality examples are reused by the discriminator, thus making it more sample efficient to train the discriminator.

Specifically, we keep the generator and discriminator from ELECTRA, but add a memory buffer between them. In ELECTRA, the generator produces corrupted text examples by trying to recover masked tokens, and discriminator takes the corrupted text examples as input and is trained to predict if each token is either original or replaced. Because the training objective of the generator is to mimic the original input sentences based on the masked ones, the generator will gradually drift

[1] The same concept is typically called *experience replay* in reinforcement learning literature.

away from its original purpose of providing random replacements to the input. Thus, the generator has the inevitable trend to produce lower and lower quality sentences as the generator is being optimized. We use memory replay to alleviate the above issue, which we refer to as *distribution drift* in this chapter. Our intuition is that, by saving examples produced by generator in the memory buffer, discriminator can reuse high-quality examples from the past, making it less sensitive to the distribution drift issue, and thus more sample efficient. We call this new transformer model *Transformer with Memory Replay* (TMR). Its architecture is illustrated in Figure 4.1. The detailed design of the memory replay would affect the performance of our new model. We will dicuss some design choices that lead to noticeable improvement on sample efficiency. Going beyond the design choices discussed in this chapter, we hope that our new model can be viewed as a general framework that provides an easy way to manipulate training examples via various memory replay designs, thus possibly achieving even better sample efficiency.

4.2 Related Work

Transformer-based Models. Before the transformer architecture came along, the dominant sequence models for NLP were based on complex recurrent neural networks, which are very hard to parallelize. In [233], the *transformer* architecture, solely based on attention mechanisms and easily parallelizable, was introduced to the NLP community, which has superior performance than traditional recurrrent neural networks. Ever since, transformer-based models have been the top performers in various NLP task competitions. For example, BERT [53] pre-trains a large transformer on unlabeled text corpora using the masked-language modeling task, achieving significant improvement on GLUE and SQuAD benchmark datasets. This demonstrates the power of combining transformers and the self-supervised pre-training strategy. MASS [214] and UniLM [54] extend BERT to the generation task by adding auto-regressive generative training objectives. Instead of masking out input tokens as in a masked-language modeling task for pre-training, XLNet [256] masks attention weights such that the input sequence is autoregressively generated in a random

order. ELECTRA corrupts the input by replacing some tokens with plausible alternatives sampled from a generator, which they showed is more sample efficient. It is also worth noting that there is significant effort dedicated to improving the transformer architecture to achieve computation efficiency [110, 115, 177, 226, 227] or to extend beyond the NLP domain [74, 89, 92, 105, 171].

Memory Replay. Memory replay (or experience replay) is critical to deep reinforcement learning to achieve super-human performance [135, 158, 200]. It has been shown to improve sample efficiency and stability by storing and reusing past transitions [63]. Some follow-up works have refined the basic memory replay mechanism [200] in various ways [10, 87, 139, 153, 220]. Several works have been trying to understand how memory replay works in the context of reinforcement learning. Liu and Zou [148] study the effects of replay buffer size and mini-batch size on learning performance. It has been reported that agent performance is sensitive to the number of environment steps taken per gradient step [66]. Sample efficiency can be improved by varying this ratio in combination with batch sizes [232]. In addition to reinforcement learning, Wu et al. [248] applies memory replay to GANs training in the task of learning new categories in a sequential fashion. They show that memory replay can prevent catastrophic forgetting [155], which is typically an issue in sequential learning.

4.3 Transformer with Memory Replay

4.3.1 Model Architecture

In this section, we describe the basic architecture of *Transformer with Memory Replay* (TMR). We keep the generator G and discriminator D from ELECTRA, but add a memory buffer between them. We describe each of them and explain the distribution drift issue which is the reason for using a memory buffer as follows.

Generator. The generator G is a transformer network [233] that maps a sequence of input tokens $\boldsymbol{x}^G = [x_1^G, \cdots, x_n^G]$ into a sequence of contextualized vector representations $h^G(\boldsymbol{x}^G) = [h_1^G, \cdots, h_n^G]$. The input token sequence is obtained from the original text token sequence $\boldsymbol{x} =$

$[x_1, \cdots, x_n]$ by masking out a random set of positions, i.e., replacing the original token with the `[MASK]` token. Typically, 15% of input tokens are masked out randomly [37, 53]. For any position t where the corresponding input token x_t^G is a `[MASK]` token, the generator outputs a probability for a particular token x with a softmax layer given by

$$p_G(x|\boldsymbol{x}^G) = \frac{\exp\left(e(x)^T h_t^G\right)}{\sum_{x'} \exp\left(e(x')^T h_t^G\right)} \tag{4.1}$$

where $e(x)$ is the embedding vector for token x. The generator is trained with the masked language modeling (MLM) task, i.e., it learns to predict the original tokens for the masked out positions. Denote the set of masked out positions as $\boldsymbol{m} = [m_1, \cdots, m_r]$. The loss function for the generator is

$$L_G(\boldsymbol{x}^G, \theta_G) = \mathbb{E}\left(\sum_{i \in \boldsymbol{m}} -\log p_G(x_i^G|\boldsymbol{x}^G)\right). \tag{4.2}$$

Discriminator. Similar to the generator, the discriminator G is also a transformer network mapping a sequence of input tokens $\boldsymbol{x}^M = [x_1^M, \cdots, x_n^M]$ into a sequence of contextualized vector representations $h^D(\boldsymbol{x}^M) = [h_1^D, \cdots, h_n^D]$. Although the generator typically has the same model architecture as the discriminator, but smaller in size, the parameter values from the discriminator are used to initialize the task-specific model during the fine-tuning stage. Existing work assumes the input to the discriminator comes directly from the output of the generator via the inference process, i.e., replace each `[MASK]` token by a token that is sampled according to $p_G(\cdot|\boldsymbol{x}^G)$ [37]. Instead, we use corrupted examples $\boldsymbol{x}^M$ sampled from the memory buffer as the input to the discriminator, in order to mitigate the distribution drift issue which we will elaborate shortly. For any position $t \in [1, n]$, the discriminator learns to predict whether the token x_t^M is *original* or *replaced*. A token is *original* if it matches the token id from the original text input. Otherwise, a token is considered *replaced*. The probability that token x_t^M is original is output by a sigmoid layer:

$$D(\boldsymbol{x}^M, t) = \text{sigmoid}\left(w^T h_t^D\right) \tag{4.3}$$

where w is the learnable parameter to the sigmoid layer. The loss function of discriminator is

$$L_D(\boldsymbol{x}^M, \theta_D) = \mathbb{E}\left(\sum_{t=1}^{n} -\mathbb{1}(x_t^M = x_t)\log D(\boldsymbol{x}^M, t) - \mathbb{1}(x_t^M \neq x_t)\log(1 - D(\boldsymbol{x}^M, t))\right). \tag{4.4}$$

Distribution Drift. We would like to point out the distribution drift issue of the generator, which is the key reason that motivates us to use the memory buffer. Specifically, to minimize the negative maximum likelihood loss L_G in Equation 4.2, the generator will tend to mimic the original input sentences and generate examples that highly resemble the input, as the training is making more and more progress. This behavior will cause the generator to drift away from its original purpose of providing random replacements to the input. Imagine an extreme scenario where the generator is highly optimized based on Equation 4.2, thus becoming perfectly capable of predicting the original token at each masked-out position. This generator would produce the highly similar (if not exactly the same) sentence as the original one. For example, if the original sentence $\boldsymbol{x}$ is "The individual images in a film are called frames" with tokens "image" and "film" being masked out, the sentence produced by a highly optimized generator $\boldsymbol{x}'^M$ would probably be "The individual images in a movie are called frames". When this sentence $\boldsymbol{x}'^M$ is received by the discriminator, all the tokens are considered as *original* except for the token "movie", because "movie" is the only token that is different from its corresponding one from the original sentence $\boldsymbol{x}$. The discriminator is supposed to learn reasonable language semantics by solving a two-class optimization problem based on these labeled tokens. However, there are two issues with $\boldsymbol{x}'^M$ that hinder the discriminator's learning progress:

1. The number of *replaced* tokens is much less than the number of `[MASK]` tokens (e.g., 50% less here), resulting in insufficient number of *replaced* tokens that would cause class imbalance problem for the discriminator;

2. The token "movie" that is considered as *replaced* is essentially noisy for the discriminator, because the sentence $\boldsymbol{x}'^M$ itself is completely acceptable, and it would be more reasonable to

consider "movie" as *original*.

The above example shows how a low-quality sentence can hurt the discriminator's learning progress.

On the other hand, an insufficiently optimized generator would probably produce an unacceptable sentence, e.g., "The individual coma in a version are called frames". The tokens "coma" and "version" are considered as *replaced* while other tokens are *original*. This is a better-quality sentence, because these labeled tokens imply that "coma" and "version" are not semantically related to other tokens from this sentence. Unfortunately, the generator has the inevitable trend to produce lower and lower quality sentences as it is being optimized. We call this phenomenon the *distribution drift* of the generator. Memory replay is a mechanism that remembers and reuses past examples by saving to and replaying from a memory buffer. It can be used to alleviate the distribution drift issue by replaying high-quality sentences from the memory buffer as input to the discriminator. It is worth noting that memory replay can also be treated as an importance sampling technique [106, 266], if we directly use it to hold the training set. This would enable other transformer models such as BERT to make use of memory replay. We do not consider this usage in this chapter because memory replay is especially effective in handling dynamic example stream rather than static example set [135, 200]. Next we discuss the memory buffer that is inserted between the generator and the discriminator.

Memory Buffer. From p_G, we create a corrupted example $\boldsymbol{x}^M$ by replacing the `[MASK]` token in $\boldsymbol{x}$ with a token randomly sampled according to $p_G(\cdot|\boldsymbol{x}^G)$ for all positions in $\boldsymbol{m}$. Any corrupted example $\boldsymbol{x}^M$ will be saved into the memory buffer. We assign a real-valued weight to each example in the memory buffer, which indicates the importance of each example. While the example importance is not directly accessible, we use different strategies to approximate it, which will be discussed in the next subsection. The memory buffer has a fixed size N. To be scalable when N is large, we use a sum tree to maintain the example weights. The memory buffer should support at least three operators: `add`, `update` and `sample`. The operator `add` is used to add a new corrupted example $\boldsymbol{x}^M$ into the memory buffer. The operator `update` is used when we want to update the weight for an example already in the memory buffer. The operator `sample` is used to sample some examples from the memory buffer according to the weight distribution, i.e., the probability that any example

is sampled is proportional to its weight. Any of these three operators has $O(\log N)$ in complexity, because of the need to traverse the sum tree from the root to a leaf node. The operator `sample` mentioned here is essentially a stochastic sampling method. In general, the probability of sampling example i is

$$P(i) = \frac{w_i^\alpha}{\sum_j w_j^\alpha} \tag{4.5}$$

where w_j is the weight for example j and the hyperparameter α determines how much prioritization is used. For example, $\alpha = 0$ corresponds to uniform sampling, and $\alpha = \infty$ corresponds to greedy sampling (i.e., sample examples with the largest weights). We illustrate more on how memory buffer works with some concrete examples in Figure 4.2.

4.3.2 Joint Learning

We learn the generator and discriminator jointly by minimizing the combined loss

$$\min_{\theta_G, \theta_D} \sum_{\boldsymbol{x}^G, \boldsymbol{x}^M} L_G(\boldsymbol{x}^G, \theta_G) + \lambda L_D(\boldsymbol{x}^M, \theta_D) \tag{4.6}$$

where λ is a scalar that balances the above two loss terms. We use the same optimizer, Adam with warmup, as in Clark et al. [37] to iteratively minimize the combined loss. Assume the mini-batch size is K.

Forward Pass. Specifically, during the forward pass of each iteration, we sample a mini-batch of original text token sequence $\{\boldsymbol{x}_{(k)}\}_{k=1}^K$. For each $\boldsymbol{x}_{(k)}$, a random set of positions is selected and the corresponding tokens are replaced by the `[MASK]` token. Thus, we get $\{\boldsymbol{x}_{(k)}^G\}_{k=1}^K$ by masking out random tokens from $\{\boldsymbol{x}_{(k)}\}_{k=1}^K$. $\{\boldsymbol{x}_{(k)}^G\}_{k=1}^K$ are provided to the generator as input, and generator loss $L_G(\boldsymbol{x}^G, \theta_G)$ is computed according to Equation 4.2. In the meantime, from the generator output $p_G(x|\boldsymbol{x}^G)$, as in Equation 4.1, corrupted examples $\{\boldsymbol{x}'^M_{(k)}\}_{k=1}^K$ are created by replacing the `[MASK]` token in $\boldsymbol{x}$ with a token randomly sampled according to $p_G(\cdot|\boldsymbol{x}^G)$. The corrupted examples $\{\boldsymbol{x}'^M_{(k)}\}_{k=1}^K$ are saved into the memory buffer once created. To get the input for

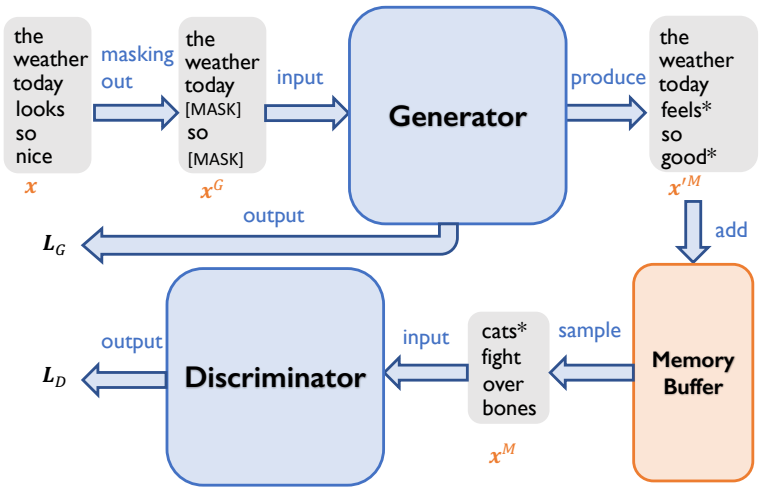


Figure 4.2: Forward pass of *Transformer with Memory Replay* (TMR). For simplicity, mini-batch size is set to 1 in the diagram to illustrate how an example is changed. We put $*$ right after a token to indicate this token has been replaced by the generator. Memory replay between the generator and discriminator helps the discriminator to learn more efficiently, by providing high-quality examples to it. The sentence "the weather today feels* so good*" is saved to the memory buffer, and a better-quality sentence "cats* fight over bones" which is sampled from the memory buffer is instead provided to the discriminator as input.

the discriminator, we sample a mini-batch of examples $\{\boldsymbol{x}_{(k)}^{M}\}_{k=1}^{K}$ from the memory buffer using the `sample` operator. The detailed design of the memory buffer (e.g., how to assign/update example weights) will affect which examples get sampled. Note that the sampled examples $\{\boldsymbol{x}_{(k)}^{M}\}_{k=1}^{K}$ are in general quite different from the examples $\{\boldsymbol{x}'^{M}_{(k)}\}_{k=1}^{K}$ that are just being saved to the memory buffer by the generator. The discriminator then computes the loss $L_D(\boldsymbol{x}^M, \theta_D)$ according to Equation 4.4. The forward pass is illustrated in Figure 4.2 with mini-batch size as 1.

Backward Pass. During the backward pass, gradient with respect to generator (or discriminator) parameters is computed from the generator (or discriminator) loss. As with [37], we don't back-propagate the discriminator loss through the generator, which is difficult because of the sampling operation. Adam with warmup is used to update the generator (or discriminator) parameters based on the generator (or discriminator) gradient.

4.3.3 Memory Replay Weights

Example weights play an important role in the memory replay mechanism. At each iteration, the generator produces a mini-batch of new corrupted examples $\{\boldsymbol{x'}^{M}_{(k)}\}^{K}_{k=1}$. The operator `add` is used to add these new examples into the memory buffer. When a new example is being added to the memory buffer, an initial weight is assigned to it. We then use the `sample` operator to sample a mini-batch of examples $\{\boldsymbol{x}^{M}_{(k)}\}^{K}_{k=1}$ from the memory buffer and provide them to the discriminator as input. The probability that an example is sampled is proportional to its weight. After forward and backward pass, we use the `update` operator to update the weights of the sampled examples based on feedback information from the discriminator. We hope to increase the weights of high-quality examples, because they are more beneficial to training the discriminator. In this way, these high-quality examples will be more likely to be sampled.

The strategy on how to assign and update example weights has great effect on which examples will be sampled as input to the discriminator, thus affecting its sample efficiency. For each new example added to the memory, we assign its initial weight as the current average of the weights. We also have tried other strategies to assign the initial weight, which are discussed in the experiments. Since the memory buffer has a fixed size, it will eventually become full as we add more and more examples to it. When we attempt to add a new example to a full memory, we evict the example with the lowest weight from the memory before adding the new example. The weight of each example indicates the importance of each example, which might change as the discriminator is being trained. To keep the example weight as up-to-date as possible, we update the weight of each sampled example at the end of each iteration. We discuss two different strategies on updating example weights.

Loss Difference. We keep track of the discriminator loss L_D, as in Equation 4.4 for each example every time it is sampled. The loss difference of the current time and the previous time is used as the new weight. Note that, since computing the loss difference requires that an example gets sampled at least twice, we will not update the weight for any new example until the second time it is sampled. In this case, we do not simply use the loss value as the new weight, because the loss value typically

decreases each time an example is sampled, so a smaller weight would not necessarily imply that it is less important. Instead, it typically means that this example has been recently sampled. Our experiments also show that using loss value as the weight results in bad performance.

Gradient Norm. Gradient norm has been shown to be the optimal weight for each example in importance sampling [266]. It demonstrates that per-example gradient norm can be a strong indicator for the example's importance. Each time an example is sampled, we update its weight using the discriminator gradient norm for this example. The drawback of using per-example gradient norm is that it incurs some overhead to compute the per-example gradient for each example in the mini-batch [77]. Recently, Katharopoulos and Fleuret [106] proposed an upper bound on gradient norm that can be computed efficiently. This upper bound has been shown to be a reasonable approximation on the gradient norm in training various neural network models [146]. We also use this upper bound as the new weight.

4.4 Evaluation Setup

4.4.1 Pre-training

We pre-train our model with two different sizes: a small model and a base model on English Wikipedia. The detailed hyperparameter values are listed in Table 4.1. As suggested by Clark et al. [37], in addition to sharing the embedding table between input and output tokens for the generator, we also share the embedding table beween generator and discriminator. We use Adam with warmup to pre-train the models. The detailed setup is the same as Clark et al. [37] if not stated otherwise. Specifically, we set $\epsilon = 1e-6$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The mini-batch size is 128 for the small model and 256 for the base model. The memory buffer size N is set to 1k in our experiments.

4.4.2 Fine-tuning

We use two commonly used datasets as the benchmark to evaluate performance: General Language Understanding Evaluation (GLUE) [238] and Stanford Question Answering Dataset (SQuAD) [179].

General Language Understanding Evaluation (GLUE). GLUE consists of eight tasks (i.e., MNLI, QQP, QNLI, SST, CoLA, STS, MRPC and RTE)[2], each of which corresponds to a specific type of NLP problem. Detailed information about the GLUE tasks is included in Table 4.2. As with Clark et al. [37], our evaluation metrics are Spearman correlation for STS, Matthews Correlation for CoLA, and accuracy for other GLUE tasks. The average of these scores is reported. Unless stated otherwise, results are on the dev set. For fine-tuning, we only need the discriminator's parameters to initialize task-specific models [37]. Specifically, we use the final hidden vector $h_1^D \in \mathbb{R}^H$ of the discriminator corresponding to the first input token (`[CLS]`) as the aggregate representation [53]. Note that H is the hidden size. The only new parameters introduced during fine-tuning are classification layer weights $W \in \mathbb{R}^{C \times H}$, where C is the number of labels. We use the standard cross-entropy loss for classification tasks: $-\log\left(\text{softmax}\left(h_1^D W^T\right)_{[class]}\right)$ where $[class]$ is the true class index.

Stanford Question Answering Dataset (SQuAD). The Stanford Question Answering Dataset (SQuAD v1.1) is a collection of 100k crowd-sourced question/answer pairs [179]. Given a question and a passage from Wikipedia containing the answer, the task of SQuAD is to predict the answer text span in the passage. As with the GLUE benchmark, for fine-tuning, we only need the discriminator's parameters to initialize task-specific models [37]. We introduce a start vector $S \in \mathbb{R}^H$ and an end vector $E \in \mathbb{R}^H$ to the task-specific model. The probability of location i being the start of the answer span is computed as a dot product between h_i^D and S, followed by a softmax over all locations in the paragraph:

$$P_i^S = \frac{e^{S \cdot h_i^D}}{\sum_j e^{S \cdot h_j^D}}. \tag{4.7}$$

Similarly, the probability of location i being the end of the answer span is computed as a dot product

[2] It is customary to exclude WNLI because it is difficult to beat even the majority classifier.

Table 4.1: Hyperparameter values for small and base models. Generator is smaller than discriminator in terms of hidden size, FFN-size and number of attention heads.

Hyperparameter	Small	Base
Generator:		
Number of layers	12	12
Hidden size	64	256
FFN inner hidden size	256	1024
Dropout	0.1	0.1
Number of attention heads	1	4
Attention head size	64	64
Attention dropout	0.1	0.1
Embedding size	128	768
Discriminator:		
Number of layers	12	12
Hidden size	256	768
FFN inner hidden size	1024	3072
Dropout	0.1	0.1
Number of attention heads	4	12
Attention head size	64	64
Attention dropout	0.1	0.1
Embedding size	128	768
Token Sequence:		
Mask percent	15	15
Max sequence length	128	512

between h_i^D and E, followed by a softmax over all locations in the paragraph:

$$P_i^E = \frac{e^{E \cdot h_i^D}}{\sum_j e^{E \cdot h_j^D}}. \tag{4.8}$$

The loss for fine-tuning is $-\log P^S_{[start]} - \log P^E_{[end]}$, where $[start]$ and $[end]$ are the correct start and end positions. The score of a candidate span from position i to position j is defined as $S \cdot h_i^D + E \cdot h_j^D$. The span with highest score where $i \leq j$ is used as the prediction.

4.5 Evaluation Results

4.5.1 Pre-training Efficiency

To show that memory replay can improve the sample efficiency, we compare our models (i.e., TMR(loss_diff) and TMR(grad_norm)) with baseline ELECTRA after being pre-trained with the

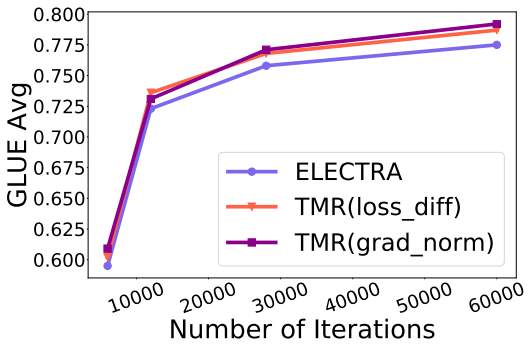
Table 4.2: GLUE benchmark details. For each task, we list the task description and the size of the training set.

Task	Description	Size
MNLI (Multi-genre Natural Language Inference)	predict whether the premise entails the hypothesis, contradicts the hypothesis, or neither, for a given premise sentence and a hypothesis sentence	393k
QQP (Quora Question Pairs)	determine whether a pair of questions are semantically equivalent	364k
QNLI (Question Natural Language Inference)	predict whether a context sentence contains the answer to a question sentence	108k
SST (Stanford Sentiment Treebank)	determine if the sentence is positive or negative in sentiment	67k
CoLA (Corpus of Linguistic Acceptability)	determine whether a given sentence is grammatical or not	8.5k
STS (Semantic Textual Similarity)	predict how semantically similar two sentences are on a 1-5 scale	67k
MRPC (Microsoft Research Paraphrase Corpus)	predict whether two sentences are semantically equivalent or not	3.7k
RTE (Recognizing Textual Entailment)	predict whether the premise entails the hypothesis or not, given a premise sentence and a hypothesis sentence	2.5k

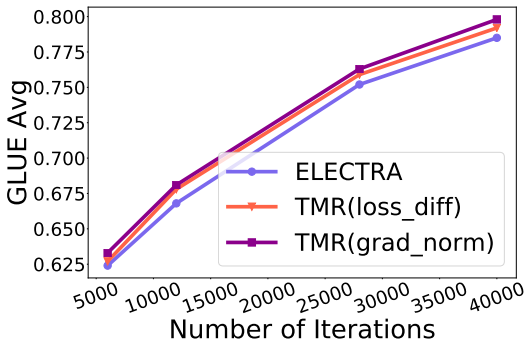
Table 4.3: Results on GLUE of the small models after pre-trained for 200k steps.

Model	MNLI	QQP	QNLI	SST	CoLA	STS	MRPC	RTE	Avg.
ELECTRA	0.730	0.881	0.836	0.862	0.733	0.832	0.809	0.584	0.783
TMR(loss_diff)	0.753	0.887	0.846	0.868	0.737	0.835	0.811	0.613	0.794

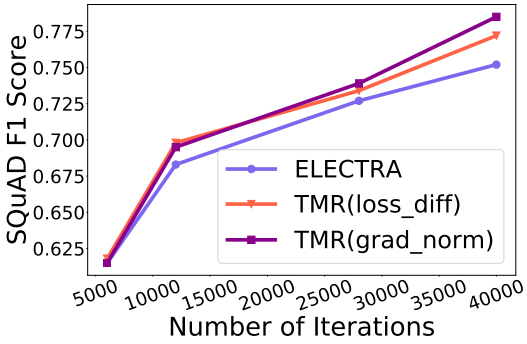
same number of iterations (i.e., also the same number of examples, because we use the same mini-batch size for different methods). TMR(loss_diff) and TMR(grad_norm) are our model *Transformer with Memory Replay* using loss difference and gradient norm to update example weight, respectively. Specifically, we pre-train different models to the same number of iterations using the same setting. We then fine-tune the models on GLUE and SQuAD to compare their performance. The results are illustrated in Figure 4.3. When the number of iterations is small, our models have almost the same performance on both GLUE and SQuAD benchmarks. As the number of iterations increases, we can see an obvious gap between our models and baseline. This is because, as more and more examples are processed, our models utilize the memory replay to remember more and more high-quality examples, thus boosting the discriminator's learning speed. On the other hand, the baseline ELECTRA uses whatever examples that are produced by the generator to train the discriminator.



(a) Small model on GLUE



(b) Base model on GLUE



(c) Base model on SQuAD

Figure 4.3: Pre-training efficiency comparison. Our models (i.e., TMR(loss_diff) and TMR(grad_norm)) have better scores than baseline ELECTRA when the pre-training has gone through enough iterations, because the memory replay needs time to accumulate enough high-quality examples.

As the training of the generator continues, the distribution of examples produced will also change due to the distribution drift issue, affecting the discriminator's training. We list the detailed results for each GLUE task after 200k steps in Table 4.3. As expected, after the same number of pre-training steps, our models TMR have better performance than baseline across most of the GLUE tasks. Note that in the original paper [37], the baseline ELECTRA is trained with much more iterations, getting slightly better GLUE scores than reported here. We do not train the models for too many iterations because the max number of iterations used in our experiments suffices to demonstrate the benefits of the proposed memory buffer mechanism (especially given the increasing concern on the effect of excessive energy consumption on the environment [202, 219]).

We also observe that TMR(grad_norm) often has better results than TMR(loss_diff). Theoretical analysis has been provided to show that gradient norm is the optimal weight for each example in

importance sampling [266]. Echoing this theoretical point, our observation empirically demonstrates that gradient norm is also a good choice for example weight in memory replay. Loss difference can be a bad approximation to gradient norm because model parameters might have changed significantly between adjacent visits to the same example. However, loss difference is a relatively cheap way to measure the example importance, which is especially beneficial in terms of the runtime efficiency as discussed right below.

Runtime Efficiency. We also report the runtime efficiency in terms of the wall-clock time needed for training. As shown in Figure 4.4, using loss difference to measure the example importance reduces the wall-clock time overhead of the memory replay, leading to better runtime efficiency.

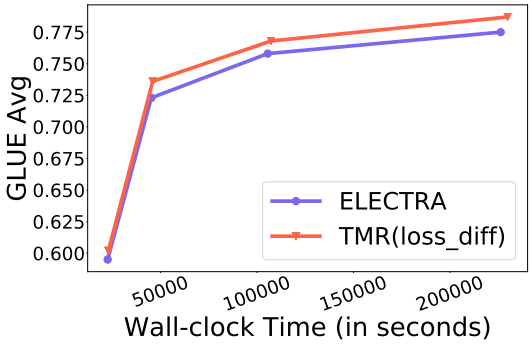


Figure 4.4: Runtime efficiency comparison using small model on GLUE. TMR(loss_diff) has better runtime efficiency than the baseline ELECTRA, because TMR(loss_diff) achieves better scores after pretaining for the same amount of wall-clock time.

4.5.2 Alternative Strategies for Weight Initialization

We discuss some alternative strategies that we have tried but have worse performance. We initialize the weight of a new example using the average of example weights in our previous experiments. This strategy will assign the same weight to new examples that arrive at the same iteration (i.e., these new examples are in the same mini-batch). The difference among examples is essentially ignored by this strategy. Intuitively, it might be better to assign weight to a new example using the weight of a similar example that is already present in the memory buffer. Since each example is represented by a sequence of token indices $\boldsymbol{x}^M = [x_1^M, x_2^M, \cdots, x_n^M]$, which can be viewed as the feature vector of this example, the similarity of two examples can be measured by

how similar their features are. There are two alternative strategies that take into consideration the similarity among examples: *Least Square Regression* and *Linear Upper Confidence Bound*. Before we dive into these two strategies, let us assume the memory buffer contains a set of examples and their corresponding weights, denoted as $\{\boldsymbol{x}^B_{(i)}, r_i\}_{i=1}^N$, where r_i is the weight of the example $\boldsymbol{x}^B_{(i)}$ and N is the capacity of the memory buffer.

Least Square Regression. To determine the weights for a new example, we solve a *Least Square Regression problem* (LSR) based on examples in the memory buffer and their weights:

$$\theta^* = arg\,\min_{\theta} \sum_{i=1}^{N} (\theta^T \boldsymbol{x}^B_{(i)} - r_i)^2. \tag{4.9}$$

Then, we assign $\theta^{*T}\boldsymbol{x}^M$ as the initial weight to the new example $\boldsymbol{x}^M$. Examples with different features will get different initial weights.

Linear Upper Confidence Bound. Multi-armed bandit problem [26, 60] provides a general framework for studying efficient sampling methods. For example, it has been used to speed up optimization methods for model training [146, 194], maximum inner product search [145], hyperparameter search [129], and model-related query execution [85]. We cast the memory replay into a multi-armed bandit problem, and rely on a bandit method to initialize and update the example weights. Specifically, we treat each example as an arm. In this way, sampling an example is equivalent to picking an arm to pull, and a high-quality example corresponds to an arm with high reward. Because each example is represented as its feature vector, there are an infinite number of possible examples, implying an infinite number of arms. To deal with an infinite number of arms, we resort to linear bandit, a special bandit setting where each arm is represented by a feature vector and reward is assumed to have a linear relationship with the arm feature vector. We use a popular linear bandit method called *Linear Upper Confidence Bound* (LinUCB), as described in Algorithm 1 of Chu et al. [36]. The Least Square Regression strategy will not change the inital weight of a new example until it gets sampled. LinUCB can essentially keep refining initial weights of new examples as more and more examples are sampled.

We did some experiments using LSR and LinUCB. The results are shown in Figure 4.5. We can see that both TMR(LSR) and TMR(LinUCB) are much worse than TMR(loss_diff). These results are a bit counter-intuitive, because the idea of assigning weight to a new example using the weight of a similar example is reasonable. We suspect the reason why neither TMR(LSR) nor TMR(LinUCB) work is that the linear relationship between feature vector and weight/reward, which is assumed by both, does not hold in our memory buffer. It would be a good future work direction to consider weight initialization strategies without linear relationship assumption.

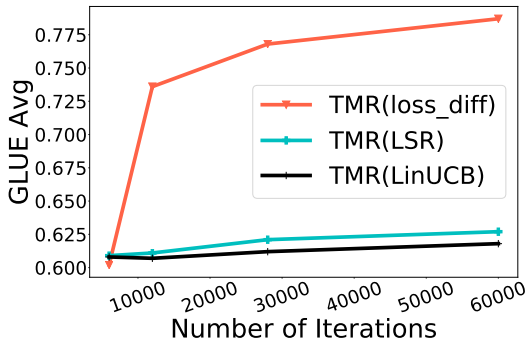


Figure 4.5: Comparison of alternative strategies for weight initialization. TMR(LSR) and TMR(LinUCB) are much worse than TMR(loss_diff) as the number of iterations increases.

4.5.3 Runtime Cost

We investigate the runtime cost for different models. In Table 4.4, we list the wall clock time to finish 100 iterations of training small models. The baseline ELECTRA model has the lowest time cost, which is expected, because our models have the overhead of maintaining the memory buffer. However, the time cost of TMR(loss_diff) is only slightly larger than that of ELECTRA. This implies that the overhead of maintaining the memory buffer can be very small, because computing loss difference will not incur additional cost. On the other hand, the time cost of TMR(grad_norm) is way larger than the baseline. This is because accurately computing per-example gradient norm requires us to feed each example of a mini-batch separately to the model for both forward and backward pass. This is usually very costly, because high parallelism provided by GPU cannot be fully utilized. TMR(grad_bound) uses the gradient upper bound from Katharopoulos and Fleuret [106] as

Table 4.4: Runtime cost comparison. The value in the table is the wall clock time (in seconds) to finish 100 iterations.

Model	Time	Model	Time
ELECTRA	377	TMR(grad_norm)	937
TMR(loss_diff)	383	TMR(grad_bound)	432

an approximation to gradient norm. It can significantly reduce the time cost of TMR(grad_norm).

4.6 Summary

In this chapter, we proposed a new transformer model, called *Transformer with Memory Replay* (TMR), for improving the sample efficiency. Our model integrates a memory replay mechanism into ELECTRA by adding a memory buffer between the generator and discriminator. Because the memory replay mechanism enables examples produced by the generator to be reused multiple times, the discriminator can be trained with high-quality examples, making it more sample efficient. We also introduced two different strategies on how to update example weights (i.e., use loss difference or gradient norm). Our experiments have shown that our models achieve higher scores than baseline when pre-trained for the same number of iterations (i.e., using the same amount of examples). We further demonstrated that the loss difference strategy leads to better runtime efficiency because of its low wall-clock time overhead.

Chapter 5

Adam with Bandit Sampling

5.1 Motivation

Stochastic gradient descent (SGD) is a popular optimization method, which iteratively updates the model parameters by moving them in the direction of the negative gradient of the loss evaluated on a mini-batch. However, standard SGD does not have the ability to use past gradients or adapt to individual parameters (a.k.a. features). Some variants of SGD, such as AdaGrad [56], RMSprop [228], AdaDelta [263], or Nadam [55], can exploit past gradients or adapt to individual features. Adam [112] combines the advantages of these SGD variants: it uses momentum on past gradients, but also computes adaptive learning rates for each individual parameter by estimating the first and second moments of the gradients. This adaptive behavior is quite beneficial as different parameters might be of different importance in terms of the convergence of the model training. In fact, by adapting to different parameters, Adam has been shown to outperform its competitors in various applications [112], and as such, has gained significant popularity.

However, another form of adaptivity that has proven beneficial in the context of basic SGD variants is adaptivity with respect to different examples in the training set [23, 28, 43, 106, 266]. For the first time, to the best of our knowledge, in this chapter we show that accounting for the varying importance of training examples can even improve Adam's performance and convergence rate. In other words, Adam has only considered the varying importance among parameters but not among

the training examples. Although some prior work exploits importance sampling to improve SGD's convergence rate [23, 28, 43, 106, 266], they often rely on some special properties of different models to estimate a sampling distribution over examples in advance. As a more general approach, we focus on learning the distribution during the training procedure, which is more adaptive. This is key to achieving faster convergence rate, because the sampling distribution typically changes from one iteration to the next, depending on the relationship between training examples and model parameters at each iteration.

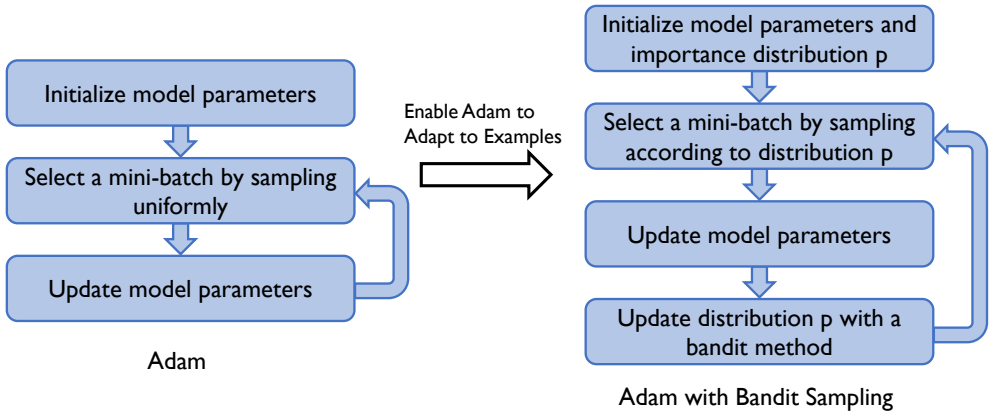


Figure 5.1: Adam with Bandit Sampling. Our method ADAMBS generalizes Adam with the ability to adapt to different examples by using a bandit method to maintain a distribution p over all training examples.

In this chapter, we propose a new general optimization method based on bandit sampling, called ADAMBS (Adam with Bandit Sampling), that endows Adam with the ability to adapt to different examples. To achieve this goal, we maintain a distribution over all examples, representing their relative importance to the overall convergence. In Adam, at each iteration, a mini-batch is selected *uniformly* at random from training examples. In contrast, we select our mini-batch according to our maintained distribution. We then use this mini-batch to compute the first and second moments of the gradient and update the model parameters, in the same way as Adam. While seemingly simple, this process introduces another challenge: how to efficiently obtain this distribution and update it at each iteration? Ideally, to obtain the optimal distribution, at each iteration one would need to compute the gradients for all training examples. However, in Adam, we only have access to gradients of the examples in the selected mini-batch. Since we only have access to partial feedback, we use a

multi-armed bandit method to address this challenge. Our idea is illustrated in Figure 5.1.

Specifically, we cast the process of learning an optimal distribution as an adversarial multi-armed bandit problem. We use a multi-armed bandit method to update the distribution over all of the training examples, based on the partial information gained from the mini-batch at each iteration. We use the EXP3 method [13], but extend it to allow for sampling multiple actions at each step. The original EXP3 method only samples one action at a time, and collects the partial feedback by observing the loss incurred by that single action. In contrast, in optimization frameworks such as Adam, we need to sample a mini-batch, which typically contains more than one example. We thus need a bandit method that samples multiple actions and observes the loss incurred by each of those actions. In this chapter, we extend EXP3 to use feedback from multiple actions and update its distribution accordingly. Although ideas similar to bandit sampling have been applied to some SGD variants and coordinate descent methods [162, 195, 196], extending this idea to Adam is not straightforward, as Adam is considerably more complex, due to its momentum mechanism and parameter adaptivity. To the best of our knowledge, we are the first to propose and analyze the improvement of using bandit sampling for Adam. Maintaining and updating distribution over all training examples incur some computational overhead. We will show an efficient way to update this distribution, which has time complexity logarithmic with respect to the total number of training examples. With this efficient update, the per-iteration cost is dominated by gradient computation, whose time complexity depends on the mini-batch size.

To endow Adam with the adaptive ability to different examples while keeping its original structure, we interleave our bandit method with Adam's original parameter update, except that we select the mini-batch according to our maintained distribution. ADAMBS therefore adapts to both different parameters and different examples. We provide a theoretical analysis of this new method showing that our bandit sampling does indeed improve Adam's convergence rate. Through an extensive empirical study across various optimization tasks and datasets, we also show that this new method yields significant speedups in practice as well.

5.2 Related Work

Importance sampling methods. Importance sampling for convex optimization problems has been extensively studied over the last few years. [187] proposed a generalized coordinate descent algorithm that samples coordinate sets to optimize the algorithm's convergence rate. More recent works [164, 266] discuss the variance of the gradient estimates of stochastic gradient descent and show that the optimal sampling distribution is proportional to the per-sample gradient norm. [162] proposed an adaptive sampling method for both block coordinate descent and stochastic gradient descent. For coordinate descent, the parameters are partitioned into several prefixed blocks; for stochastic gradient descent, the training examples are partitioned into several prefixed batches. However, it is difficult to determine an effective way to partition blocks of parameters or batches of examples. [106] proposed to sample a big batch at every iteration to compute a distribution based on gradient norms of these examples from the big batch, followed by a mini-batch that is sampled from the big batch for parameter update. However, it is unclear how much speedup their method can achieve in terms of theoretical convergence rate.

Other sample selection methods. Several strategies have been proposed to carefully select mini-batches in order to improve on training deep learning models. Locality-Sensitive Hashing (LSH) has been used to improve the convergence rate of SGD by adaptively selecting examples [29]. It is worth noting that a recent paper [183] points out that Adam's rapid decay of the learning rate using the exponential moving averages of squared past gradients essentially limits the reliance of the update to only the past few gradients. This prevents Adam from convergence in some cases. They proposed a variant of Adam, called AMSGRAD, with long term memory of past gradients. In the main text of this chapter, we remain focused on the original Adam. Similar analysis could be carried over to AMSGRAD, which is discussed in appendix. We note that Adam with the learning rate warmup (another variant of Adam) is the common practice in training transformer models for NLP tasks [37, 53]. However, due to the lack of well-studied theoretical analysis of this variant in the literature, we still base our dicussion on the original Adam.

5.3 Preliminaries about Adam

We consider the following convex optimization problem: $\min_{\theta \in \mathcal{X}} f(\theta)$ where f is a scalar-valued objective function that needs to be minimized, and $\theta \in \mathbb{R}^d$ is the parameter of the model. Let the gradient of f with respect to θ be denoted as G. Assuming the training dataset is of size n, we have $G = \frac{1}{n} \sum_{i=1}^{n} g_i$, where g_i is the gradient computed with only the i-th example. Furthermore, at each iteration t, we select a mini-batch of examples from the whole training set. We denote the realization of f with respect to the mini-batch selected at iteration t as f_t, and the gradient of f_t with respect to θ as G_t. Depending on the sampling strategy of a mini-batch, in some cases, G_t could be a biased estimate of G, meaning $\mathbb{E}[G_t] \neq G$. However, an unbiased estimate is required to update model parameters in stochastic optimization such as SGD and Adam. In such cases, we need to get an unbiased estimate $\hat{G}_t$, ensuring that $\mathbb{E}[\hat{G}_t] = G$. When a mini-batch is selected by sampling uniformly from all of the training examples, we have $\mathbb{E}[G_t] = G$, thus allowing $\hat{G}_t$ to simply be G_t.

Adam [112] selects every mini-batch by uniform sampling and updates exponential moving averages of the gradient m_t and the squared gradient v_t with hyperparameters $\beta_1, \beta_2 \in [0, 1)$, which control the exponential decay rates of these moving averages: $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \hat{G}_t$, $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \hat{G}_t^2$ where $\hat{G}_t^2$ indicates the element-wise square of $\hat{G}_t$. The moving averages m_t and v_t are estimates of the 1st moment (the mean) and 2nd raw moment (the uncentered variance) of the gradient. These moment estimates are biased toward zero and are then corrected, resulting in bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$: $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$, $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ where β_1^t and β_2^t are β_1 and β_2 raised to the power t, respectively. Next, the parameter is updated according to $\theta_t \leftarrow \theta_{t-1} - \alpha_\theta \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$ where ϵ is a small value, to avoid division by zero.

A flexible framework to analyze iterative optimization methods such as Adam is the online learning framework. In this online setup, at each iteration t, the optimization algorithm picks a point $\theta_t \in \mathbb{R}^d$. A loss function f_t is then revealed based on the seleted mini-batch, and the algorithm incurs loss $f_t(\theta_t)$. At the end of T iterations, the algorithm's regret is given by $R(T) = \sum_{t=1}^{T} f_t(\theta_t) - \min_{\theta \in \mathcal{X}} \sum_{t=1}^{T} f_t(\theta)$. In order for any optimization method to converge, it is necessary to ensure that $R(T) = o(T)$. For Adam, the convergence rate is summarized in the Theorem

4.1 from [112]. Under further assumptions as in Corollary 4.2 from [112], it can be shown that $R(T) = o(T)$. In this chapter, we propose to endow Adam with bandit samping which could further improve the convergence rate under some assumptions.

5.4 Adam with Bandit Sampling

5.4.1 Adaptive Mini-Batch Selection

Suppose there are n training examples. At iteration t, a mini-batch of size K is selected by sampling with replacement from all of the training examples, according to a distribution $p^t = \{p_1^t, p_2^t, \cdots, p_n^t\}$. Here, p^t represents the relative importance of each example during the model training procedure. Denote the indices of examples selected in the mini-batch as the set $I^t = \{I_1^t, I_2^t, \cdots, I_K^t\}$. Assume the gradient computed with respect to the only example I_k^t is $g_{I_k^t}$. Its unbiased estimate is $\hat{g}_{I_k^t} = \frac{g_{I_k^t}}{np_{I_k^t}}$. We can easily verify that $\hat{g}_{I_k^t}$ is unbiased because $\mathbb{E}_{p^t}\left[\hat{g}_{I_k^t}\right] = \frac{1}{n}\sum_{i=1}^n g_i = G$. Therefore, we define the unbiased gradient estimate $\hat{G}_t$ according to batch I^t at iteration t as

$$\hat{G}_t = \frac{1}{K}\sum_{k=1}^{K} \hat{g}_{I_k^t}. \tag{5.1}$$

Similarly, we can verify that $\mathbb{E}_{p^t}\left[\hat{G}_t\right] = \frac{1}{n}\sum_{i=1}^n g_i = G$.

It is worth noting that $\hat{G}_t$ defined for ADAMBS is different than that of Adam. This is because the sampling strategy is different, and appropriate bias-correction is necessary here. We use $\hat{G}_t$, defined in Equation 5.1, to update first moment estimate and second moment estimate in each iteration. Specifically, let m_t and v_t be the first and second moment estimates at iteration t, respectively. Then we update them in the following way

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot \hat{G}_t, \;\; v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot \hat{G}_t^2 \tag{5.2}$$

where $\beta_1, \beta_2 \in [0, 1)$ are hyperparameters that control the exponential decay rates of these moving averages. Our new method ADAMBS is described in Algorithm 5.1. Details about $update$ function

Algorithm 5.1 ADAMBS, our proposed Adam with bandit sampling. Note that β_1, β_2 are hyper-parameters controling exponential decay rates of first and second moment estimates, respectively. β_1^t, β_2^t denote β_1, β_2 raised to power t, respectively. α_θ is the learning rate for model parameter update. $\hat{G}_t^2$ is the element-wise square of $\hat{G}_t$.

1: Initialize model parameter θ_0;
2: Initialize first moment estimate $m_0 \leftarrow 0$, and second moment estimate $v_0 \leftarrow 0$;
3: Initialize distribution $p_j^0 \leftarrow \frac{1}{n}, \forall 1 \leq j \leq n$, and set iteration index $t = 0$;
4: **while** θ_t not converged **do**
5: $\quad t \leftarrow t + 1$;
6: $\quad$ select a mini-batch I^t by sampling with replacement from p^{t-1};
7: $\quad$ compute unbiased gradient estimate $\hat{G}_t$;
8: $\quad m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \hat{G}_t, \quad v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \hat{G}_t^2$;
9: $\quad \hat{m}_t \leftarrow m_t / (1 - \beta_1^t), \quad \hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$;
10: $\quad \theta_t \leftarrow \theta_{t-1} - \alpha_\theta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$;
11: $\quad p^t \leftarrow update(p^{t-1}, I^t, \{g_{I_k^t}\}_{k=1}^K)$;
12: **end while**

in line 11 are given in the next subsection.

Our method mantains a fine-grained probability distribution over all examples. This provides more flexibility in choosing mini-batches than prior work that uses coarse-grained probability distribution over pre-fixed mini-batches [162], because it is generally hard to decide how to partition the batches for pre-fixed mini-batches. If the training set is partitioned randomly, any mini-batch is likely to contain some important examples and some unimportant examples, making any two mini-batches equally good. In this case, prioritizing one mini-batch over another will not bring any advantage. It requires a fair amount of time on preprocessing the training dataset to partition the dataset in a good way, especially when the dataset is large. Some might argue that we could simply set the batch size to one in [162]. While the issue of batch partitioning does not exist anymore, this would significantly hurt the convergence rate because only one example is processed at each iteration. In contrast, ADAMBS does not require pre-partitioning mini-batches. At every iteration, a new mini-batch is formed dynamically by sampling from the whole dataset according to distribution p^{t-1}. Here, the distribution p^{t-1} is learned so that important examples can be selected into one mini-batch with high probability. Thus, it is more likely to get a mini-batch with all important examples, which could significantly boost the training performance.

We analyze the convergence of ADAMBS in Algorithm 5.1 using the same online learning framework [269] that is used by Adam. The following theorem [1] holds, regarding the convergence rate of ADAMBS.

Theorem 5.1. *Assume that the gradient estimate $\hat{G}_t$ is bounded, $\|\hat{G}_t\|_2 \leq G, \|\hat{G}_t\|_\infty \leq G_\infty$, and distance between any θ_t is bounded, $\|\theta_n - \theta_m\|_2 \leq D, \|\theta_n - \theta_m\|_\infty \leq D_\infty$ for any $m, n \in \{1, \cdots, T\}$, and $\beta_1, \beta_2 \in [0, 1)$ satisfy $\frac{\beta_1^2}{\sqrt{\beta_2}} < 1$. Let $\alpha_\theta = \frac{\alpha}{\sqrt{t}}$, and $\beta_{1,t} = \beta_1 \lambda^{t-1}, \lambda \in (0, 1)$, and $\gamma = \frac{\beta_1^2}{\sqrt{\beta_2}}$.* ADAMBS *achieves the following convergence rate, for all $T \geq 1$,*

$$R(T) \leq \rho_1 d\sqrt{T} + \sqrt{d}\rho_2 \sqrt{\frac{1}{n^2 K} \sum_{t=1}^{T} \mathbb{E}\left[\sum_{j=1}^{n} \frac{\|g_j^t\|^2}{p_j^t}\right]} + \rho_3 \tag{5.3}$$

where d is the dimension of parameter space and

$$\rho_1 = \frac{D^2 G_\infty}{2\alpha(1-\beta_1)}, \quad \rho_2 = \frac{\alpha(1+\beta_1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2}, \quad \rho_3 = \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\gamma)^2}. \tag{5.4}$$

5.4.2 Update of Distribution p^t

Algorithm 5.2 The distribution update rule for p^t.

1: **Function**: $update(p^{t-1}, I^t, \{g_{I_k^t}\}_{k=1}^K)$

2: Compute loss $l_{t,j} = -\frac{\|g_j^t\|^2}{(p_j^t)^2} + \frac{L^2}{p_{min}^2}$ if $j \in I^t$; otherwise, $l_{t,j} = 0$;

3: Compute an unbiased gradient estimate $\hat{h}_{t,j} = \frac{l_{t,j} \sum_{k=1}^{K} \mathbb{1}(j = I_k^t)}{K p_j^t}, \forall 1 \leq j \leq n$;

4: $w_j^t = p_j^{t-1} \exp(-\alpha_p \hat{h}_{t,j}), \forall 1 \leq j \leq n$;

5: $p^t = \arg\min_{q \in \mathcal{P}} D_{kl}(q \| w^t)$;

6: **Return:** p^t

From Equation 5.3, we can see that p^t will affect the convergence rate. We wish to choose p^t that could lead to a faster convergence rate. We derive how to update p^t by minimizing the right side of Equation 5.3. Specifically, we want to minimize $\sum_{t=1}^{T} \mathbb{E}\left[\sum_{j=1}^{n} \frac{\|g_j^t\|^2}{p_j^t}\right]$. It can be shown that for every iteration t, the optimal distribution p^t is proportional to the gradient norm

[1] All omitted proofs can be found in the appendix.

of the individual example [7, 164]. Formally speaking, for any t, the optimal solution to the problem $\arg\min_{\sum_{i=1}^{n} p_i^t=1} \sum_{j=1}^{n} \frac{\|g_j^t\|^2}{p_j}$ is $p_j^t = \frac{\|g_j^t\|}{\sum_{i=1}^{n} \|g_i^t\|}, \forall j$. It is computationally prohibitive to get the optimal solution, because we need to compute the gradient norm for every example at each iteration. Instead, we use a multi-armed bandit method to learn this distribution from the partial information that is available during training. The multi-armed bandit method maintains the distribution over all examples, and keeps updating this distribution at every training iteration. The partial information that we have at every iteration is the gradients of examples in the mini-batch. We use a bandit method based on EXP3 [13] but extended to handle multiple actions at every iteration. The pseudocode is described in Algorithm 5.2.

To further illustrate our distribution update algorithm from the perspective of the bandit setting, the number of arms is n, where each arm corresponds to each training example, and the loss of pulling the arm j is $l_{t,j}$ which is defined at line 2. We denote L as the upper bound on the per-example gradient norm, i.e., $\|g_j^t\| \leq L, \forall t, j$. Similar upper bound is commonly used in related literature [112, 162], because of the popular gradient clipping trick [265]. Each time we update the distribution, we only pull these arms from the set I^t, which is the mini-batch of training examples at current iteration. In line 5, $D_{kl}(q\|w^t)$ is the KL divergence between q and w^t, and the set $\mathcal{P}$ is defined as $\{p \in \mathbb{R}^n : \sum_{j=1}^{n} p_j = 1, p_j \geq p_{min} \forall j\}$, where p_{min} is a constant. From the definition of $l_{t,j}$ as in line 2 from Algorithm 5.2, we can see that the loss $l_{t,j}$ is always nonnegative, and is inversely correlated with the gradient norm $\|g_j^t\|$. This implies that an example with small gradient norm will receive large loss value, resulting in its weight getting decreased. Thus, examples with large gradient norms will be sampled with higher probabilities in subsequent iterations. Using a segment tree structure to store the weights for all the examples, we are able to update p^{t-1} in $O(K \log n)$ time [162]. With the efficient way to update distribution, the per-iteration cost is dominated by the computation of gradients, especially for large models in deep learning. For simplicity, our theoretical analysis is still focused on convergence rate with respect to the number of iterations. In experiments, we demonstrate that our method can achieve a faster convergence rate with respect to time.

Let $B_\phi(x, y) = \phi(x) - \phi(y) - \langle \nabla\phi(y), x - y \rangle$ be the Bregman divergence associated with

ϕ, where $\phi(p) = \sum_{j=1}^{n} p_j \log p_j$. By choosing this form of Bregman divergence, we can now study our EXP3-based distribution update algorithm under the general framework of online mirror descent with bandit feedback [27]. We have the following lemma regarding the convergence rate of Algorithm 5.2.

Lemma 5.1. *Assume that $\|g_j^t\| \leq L, p_j^t \geq p_{min}$ for all t and j, and $B_\phi(p, p') \leq R^2$ for any $p, p' \in \mathcal{P}$, if we set $\alpha_p = \sqrt{\frac{2R^2 p_{min}^4}{nTL^4}}$, the update in Algorithm 5.2 implies*

$$\mathbb{E}\sum_{t=1}^{T}\sum_{j=1}^{n}\frac{\|g_j^t\|^2}{p_j^t} - \min_{p\in\mathcal{P}}\mathbb{E}\sum_{t=1}^{T}\sum_{j=1}^{n}\frac{\|g_j^t\|^2}{p_j} \leq \frac{RL^2}{p_{min}^2}\sqrt{2nT}. \tag{5.5}$$

Combining Theorem 5.1 and Lemma 5.1, we have the following theorem regarding the convergence of ADAMBS.

Theorem 5.2. *Under assumptions from both Theorem 5.1 and Lemma 5.1,* ADAMBS *with the distribution update rule in Algorithm 5.2 achieves the following convergence rate*

$$R(T) \leq \rho_1 d\sqrt{T} + \rho_2\frac{\sqrt{d}}{n\sqrt{K}}\sqrt{M} + \frac{\rho_2 L\sqrt{R}}{p_{min}}\frac{\sqrt{d}}{n\sqrt{K}}(2nT)^{1/4} + \rho_3 \tag{5.6}$$

where $M = \min_p \sum_{t=1}^{T}\mathbb{E}\left[\sum_{j=1}^{n}\frac{\|g_j^t\|^2}{p_j}\right]$, and ρ_1, ρ_2, ρ_3 are defined in Equation 5.4.

Obviously, we have that, for ADAMBS, $R(T) = o(T)$, ensuring that it can converge.

5.4.3 Comparison with Uniform Sampling

We now study a setting where we can further bound Equation 5.6 to show that the convergence rate of our ADAMBS is provably better than that of Adam, which uses uniform sampling. For simplicity, we consider a neural network with one hidden layer for a binary classification problem. The hidden layer contains one neuron with ReLU activation, and output layer also contains one neuron with sigmoid activation. Cross-entropy loss is used for this binary classification problem. The total loss of this neural network model can be written in the following way $f(\theta) = \frac{1}{n}\sum_{j=1}^{n} f_j(\theta) = -\frac{1}{n}\sum_{j=1}^{n} y_j \log\sigma(ReLU(z_j^T\theta))$ where $\sigma(\cdot)$ is the sigmoid activation function, $ReLU(\cdot)$ is the ReLU activation function, $y_j \in \{-1, +1\}$ and z_j are the label and the feature vector for j-th example, respectively. Denote that $\theta^* = \arg\min f(\theta)$. Here,

$g_j = \partial f_j(\theta) = -y_j(1 - \sigma(ReLU(z_j^T\theta)))\mathbb{1}(z_j^T\theta > 0)z_j$. It implies that $\|g_j\|^2 \leq \|z_j\|^2 = \sum_{i=1}^d z_{j,i}^2$. We further asssume that feature vector follows doubly heavy-tailed distribution, which means that, $z_{j,i} \in \{-1, 0, +1\}$ and $\mathbb{P}(|z_{j,i}| = 1) = \beta_3 i^{-\gamma} j^{-\gamma}$, where $\gamma \geq 2$.

Lemma 5.2. *If the examples are sampled with uniform distribution, i.e.* $p_j = 1/n, \forall 1 \leq j \leq n$, *assuming that feature vector follows doubly heavy-tailed distribution, for the aforementioned neural network model, we have* $\mathbb{E}\sum_{j=1}^n \frac{\|g_j\|^2}{p_j} = \beta_3 n \log n \log d$.

Following Lemma 5.2, we have

Theorem 5.3. *Assuming that feature vector follows doubly heavy-tailed distribution, for the aforementioned neural network model, original Adam achieves the following rate*

$$R(T) \leq O(d\sqrt{T}) + O(\frac{\sqrt{d\log d}}{\sqrt{K}}\frac{\sqrt{n\log n}}{n}\sqrt{T}). \tag{5.7}$$

On the other hand, we have

Lemma 5.3. *Assuming that feature vector follows doubly heavy-tailed distribution, for the aforementioned neural network model, we have*

$$\min_{p_j \geq p_{min}} \sum_{j=1}^n \frac{\|g_j\|^2}{p_j} = O(\log d \log^2 n). \tag{5.8}$$

By plugging Lemma 5.3 into Theorem 5.2, we have

Theorem 5.4. *Assuming that feature vector follows doubly heavy-tailed distribution, for the aforementioned neural network model,* ADAMBS *achieves the following rate*

$$R(T) \leq O(d\sqrt{T}) + O(\frac{\sqrt{d\log d}}{\sqrt{K}}\frac{\sqrt{\log^2 n}}{n}\sqrt{T}). \tag{5.9}$$

Comparing the second terms at Equations 5.7 and 5.9, we see that our ADAMBS converges faster than Adam. The convergence rates are summarized in Table 5.1. In this table, we also compare against adaptive sampling methods from [162]. They maintain a distribution over prefixed batches. Adam with adaptive sampling over prefixed batches is called ADAM-APT, and Adam with unifom sampling over prefixed batches is called ADAM-UNI. We can see that our method ADAMBS achieves

Table 5.1: Convergence rates under doubly heavy-tailed distribution.

Algorithm	Convergence Rate	Algorithm	Convergence Rate
ADAMBS	$O(d\sqrt{T}) + O(\frac{\sqrt{d\log d}}{\sqrt{K}}\frac{\sqrt{\log^2 n}}{n}\sqrt{T})$	ADAM-APT	$O(d\sqrt{T}) + O(\sqrt{d\log d}\frac{\sqrt{\log^2 n}}{n}\sqrt{T})$
Adam	$O(d\sqrt{T}) + O(\frac{\sqrt{d\log d}}{\sqrt{K}}\frac{\sqrt{n\log n}}{n}\sqrt{T})$	ADAM-UNI	$O(d\sqrt{T}) + O(\sqrt{d\log d}\frac{\sqrt{n\log n}}{n}\sqrt{T})$

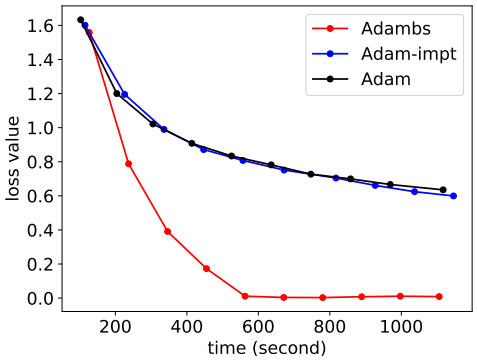
faster convergence rate than the others. Depending on constants not shown in the big-O notation, it's also possible that the convergence rate is dominated by the first term $O(d\sqrt{T})$, which makes our improvement marginal. We also rely on the experiments in the next section to demonstrate our method's faster convergence rate in practice.

5.5 Evaluations

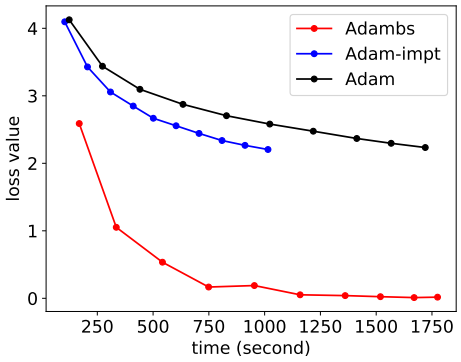
5.5.1 Setup

To empirically evaluate the proposed method, we investigate different popular deep learning models. We use the same parameter initialization when comparing different optimization methods. In total, 5 datasets are used: MNIST, Fashion MNIST, CIFAR10, CIFAR100 and IMDB. We run experiments on these datasets because they are benchmark datasets commonly used to compare optimization methods for deep learning [106, 112, 183]. It is worth noting that the importance sampling method proposed in [106] could also be applied to Adam. In addition, they proposed an efficient way to upper bound the per-example gradient norm for neural networks to compute the distribution for importance sampling. This could also be beneficial to our method, because the upper bound of gradient norm could be used in place of the gradient norm itself to update our distribution. In the experiments, we compare our method against Adam and Adam with importance sampling (as described in [106], which we call ADAM-IMPT). To be fair, we use the upper bound in the place of per-example gradient norm in our method. All the previous analysis also holds if an upper bound of gradient norm is used, because similar to Theorem 1, it will still upper bound $R(T)$. Experiments are conducted using Keras [35] with TensorFlow [2] based on the code from [106]. To see if our method could accelerate the training procedure, we plot the curves of training loss value vs. wall

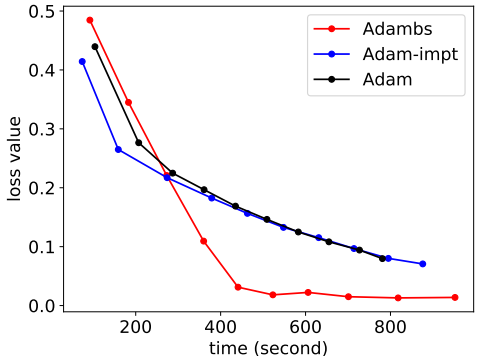
clock time for these three methods [2]. The curves are based on average results obtained by repeating each experiment five times with different random seeds. To see the difference in the performance of these optimization method, we use the same values for hyperparameters. Specifically, β_1 and β_2 are common hyperparameters to Adam, ADAM-IMPT and ADAMBS, which are chosen to be 0.9 and 0.999, respectively.



(a) CIFAR10



(b) CIFAR100



(c) Fashion MNIST

Figure 5.2: CNN model on CIFAR10, CIFAR100 and Fashion MNIST

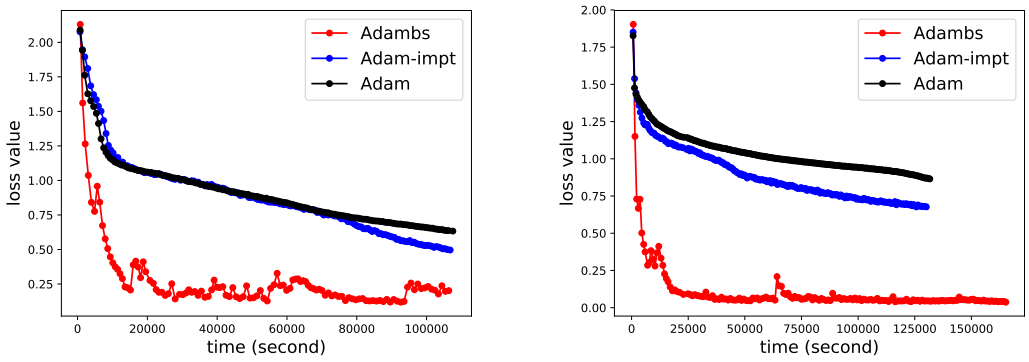
5.5.2 Convolutional Neural Networks

Convolutional neural networks (CNN) with several layers of convolution, pooling, and non-linear units have shown considerable success in computer vision tasks. We train CNN models on

[2]Please see the appendix for curves of training error rate vs. wall clock time.

three different datasets: CIFAR10, CIFAR100 and Fashion MNIST. CIFAR10 and CIFAR100 are labeled subsets of the 80 million tiny images dataset [119]. CIFAR10 consists of $60,000$ color images of size 32×32 in 10 classes with $6,000$ images per class, whereas CIFAR100 consists of $60,000$ color images of size 32×32 in 100 classes with 600 images per class. Fashion MNIST dataset [252] is similar to MNIST dataset except that images are in 10 fashion categories.

For CIFAR10 and CIFAR100, our CNN architecture has 4 layers of 3×3 convolution filters and max pooling with size 2×2. Dropout with dropping probability 0.25, is applied to the 2nd and 4th convolutional layers. This is then followed by a fully connected layer of 512 hidden units. For Fashion MNIST, since the dataset is simpler, we use a simpler CNN model. It contains 2 layers of 3×3 filters and max pooling with size 2×2 is applied to the 2nd convolutional layer, which is followed by a fully connected layer of 128 hidden units. The mini-batch size is set to 128, and learning rate is set to 0.001 for all methods on all three datasets. All three methods are used to train CNN models for 10 epochs and the results are shown in Figure 5.2. For CIFAR10 and CIFAR100, we can see that our method ADAMBS achieved loss value lower than others very quickly, within 1 or 2 epochs. For Fashion MNIST, our method ADAMBS is worse than others at the very begining, but keeps decreasing the loss value at a faster rate. After around 300 seconds, ADAMBS is able to achieve lower loss value than others.



(a) MNIST (b) Fashion MNIST

Figure 5.3: RNN model on MNIST and Fashion MNIST

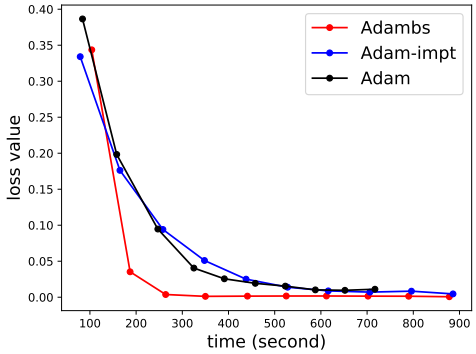


Figure 5.4: RCNN model on IMDB

5.5.3 Recurrent Neural Networks

Recurrent neural networks, such as LSTM and GRU, are popular models to learn from sequential data. To showcase the generality of our method, we use ADAMBS to accelerate the training of RNN models in image classification problems, where image pixels are fed to RNN as a sequence. Specifically, we train an LSTM with dimension 100 in the hidden space, and ReLU as recurrent activation function, which is followed by a fully connected layer to predict image class. We use two datasets: MNIST and Fashion MNIST. The batch size is set as 32, the learning rate is set as 10^{-6}, and the maximum number of epochs is set as 200 for all methods on both datasets. The results are shown in Figure 5.3. ADAMBS was able to quickly achieve lower loss value than the others.

5.5.4 Recurrent Convolutional Neural Networks

Recently, it has been shown that RNN combined with CNN can achieve good performance on some NLP tasks [121]. This new architecture is called recurrent convolutional neural network (RCNN). We train an RCNN model for the sentiment classification task on an IMDB movie review dataset. It contains $25,000$ movie reviews from IMDB, labeled by sentiment (positive or negative). Reviews are encoded by a sequence of word indexes. On this dataset, we train an RCNN model, which consists of a convolutional layer with filter of size 5, and a max pooling layer of size 4, followed by an LSTM and a fully connected layer. We set batch size to 30 and learning rate to 0.001,

and run all methods for 10 epochs. The result is shown in Figure 5.4. We can see that all methods converge to the same loss value, but ADAMBS arrives at convergence much faster than the others.

5.6 Summary

In this chapter, we have presented an efficient method for accelerating the training of Adam by endowing it with bandit sampling. Our new method, ADAMBS, is able to adapt to different examples in the training set, complementing Adam's adaptive ability for different parameters. A distribution is maintained over all examples and represents their relative importance. Learning this distribution could be viewed as an adversarial bandit problem, because only partial information is available at each iteration. We use a multi-armed bandit approach to learn this distribution, which is interleaved with the original parameter update by Adam. We provided a theoretical analysis to show that our method can improve the convergence rate of Adam in some settings. Our experiments further demonstrate that ADAMBS is effective in reducing the training time for several tasks with different deep learning models.

Chapter 6

Selective Classification with Curriculum Learning

6.1 Motivation

In *selective classification*, the goal is to design a predictive model that is allowed to abstain from making a prediction whenever it is not sufficiently confident. A model with this reject option is called a *selective model*. In other words, a selective model will reject certain examples as appropriate, and provide predictions only for accepted examples. In many real-life scenarios[1], such as medical diagnosis [117], legal documents translation [234], robotics and self-driving cars [73], selective models are used to minimize the risk of wrong predictions on the hard examples by abstaining from providing any predictions and possibly seeking human intervention.

In this chapter, we focus on selective neural network models, which are essentially neural network models with the reject option. These models have been shown to achieve impressive results [71, 72, 151]. Specifically, [72] proposed a neural network model, SELECTIVENET, that allows end-to-end optimization of selective models. SELECTIVENET contains a main body block followed by three heads: one for minimizing the error rate among the accepted examples, one for selecting the examples for acceptance or rejection, and one for the auxiliary task of minimizing the

[1]These are further elaborated in Section D.1 in the appendix.

error rate on all examples. These three heads are illustrated later in Figure 6.1. The final goal of this model is to minimize the error rate among the accepted examples while satisfying a coverage constraint in terms of the least percentage of examples that need to be accepted. The coverage constraint is imposed to avoid the trivial solution of rejecting all examples to get a 0% error rate. Ideally, the model should reject hard examples and accept easy ones to lower its overall error rate. While it is clear that difficulty scores are helpful, they are typically unknown in most settings. Therefore, to leverage difficulty scores we must overcome two challenges: (1) how to obtain the difficulty scores as accurately as possible and (2) how to best utilize them in a selective neural network model.

Recent curriculum learning techniques have investigated how to use example difficulty scores to improve neural network models' performance [83, 251]. To the best of our knowledge, these techniques only consider the typical classification setting where the error rate on all examples should be minimized. Curriculum learning techniques often use a scoring function to estimate the difficulty scores. There are different approaches to constructing a scoring function from a reference model, such as (1) confidence score [83], (2) learned epoch/iteration [251], and (3) estimated c-score [99]. Prior work has shown that these scoring functions are highly correlated with one another and lead to similar performance [251]. Existing curriculum learning techniques use the estimated difficulty scores only to decide the order of examples being exposed to the model during the training process. Typically, they expose easy examples to the model during the early phase of the training process, and gradually transition to the hard examples as the training progresses. In this chapter, we consider selective classification, which has a more relaxed goal: minimize the error rate on *the accepted examples* for a given coverage constraint. Difficulty scores are even more useful in selective classification, because the selection of accepted examples has a direct impact on the model's final performance (i.e., a lower error rate can be achieved by rejecting hard examples and accepting easy ones). This inspires us to design a new method for training selective neural network models.

Contributions. In this chapter, we propose a new method for training selective neural network

models inspired by curriculum learning. Our curriculum-inspired method has two benefits. First, existing training methods ignore the coverage constraint when constructing the mini-batch for each iteration, which we show will introduce misguiding noise to the training progress of selective neural network models. Taking advantage of the estimated difficulty scores, our method calibrates the ratio of easy and hard examples in each mini-batch to match the desirable value due to the coverage constraint. Second, our method also adopts the curriculum idea of increasing difficulty levels gradually over the training process, which has been demonstrated to improve neural network training [83, 242]. However, instead of relying on existing vanilla curriculum techniques that exploit difficulty ordering at the example level, our method is tailored to selective neural network training by exploiting difficulty ordering at the mini-batch level. We will show that this change is necessary because of the need to match coverage constraint as just explained.

To summarize, we make the following contributions:

1. To the best of our knowledge, we are the first to investigate the benefits of leveraging example difficulty scores to improve selective neural network model training.

2. We design a new curriculum-inspired method with two benefits for training selective neural networks: (a) calibrating the ratio of easy and hard examples in each mini-batch, which has been ignored by existing methods (Section 6.4.1); (b) adopting a curriculum idea which is tailored to selective neural network training by exploiting difficulty ordering at the mini-batch level (Section 6.4.2).

3. We conduct extensive experiments demonstrating that our method improves the converged error rate (up to 13% lower) of selective neural network models compared to the state-of-the-art (Section 6.5.2). We also show that our method is better than alternative designs using vanilla techniques from existing curriculum learning literature (Section 6.5.3).

6.2 Related Work

Selective classification. Prior work on selective classification primarily focuses on adding the reject option to classical learning algorithms such as SVM [67], nearest neighbors [86], boosting [40] and online learning methods [41]. In particular, [71] applies selective classification techniques in the context of deep neural networks (DNNs). They show how to construct a selective classifier given a trained neural network model. They decide whether or not to reject each example based on a confidence score. They rely on two techniques for extracting confidence scores from a neural network model: Softmax Response (SR), and Monte-Carlo dropout (MC-dropout). SR is the maximal activation in the softmax layer for a classification model, which is used as the confidence score. MC-dropout estimates the confidence score based on the statistics of numerous forward passes through the network with the dropout applied. Unfortunately, MC-dropout requires hundreds of forward passes for each example, incurring a massive computational overhead. More recently, [72] proposes a selective neural network that jointly learns a predictive function and a selection function. This model is trained end-to-end, resulting in a selective model that is optimized over the covered domain. They show empirically that this selective neural network outperforms previous methods based on SR or MC-dropout. In addition, inspired by portfolio theory, [151] propose a new loss function for selective classification based on the doubling rate of gambling. Self-adaptive training [91] that calibrates training process by model predictions can also be applied to selective classification. In [69], the selective classification is formulated into optimising a collection of class-wise decoupled one-sided empirical risks. However, none of the prior work in selective classification has exploited estimated difficulty scores as in curriculum learning, which is what we propose for the first time in this chapter.

Curriculum learning. Inspired by how human students are gradually introduced to concepts with increasing complexity, curriculum learning techniques train neural networks by feeding easier examples earlier in the training process and gradually transitioning to more difficult examples. There is a huge surge of interests in applying curriculum learning to different domains in recent years, e.g., image classification [80], machine translation [174], language model pre-training [128],

healthcare prediction [58] and graph learning [76]. Despite the differences among domains, these techniques follow a general pattern where a sequence of training criteria are presented to the model over T training steps and the training criteria affect how examples are selected at each step [215, 242]. Typically, example selection works in the following way: the mini-batch at each iteration is uniformly sampled from a working subset, which starts with a few easiest examples, and is being appended with harder and harder examples after every several epochs until all the training examples are added. Our curriculum-inspired method is also based on the above example selection strategy. Note that *anti-curriculum* method uses the opposite strategy by starting from the hardest examples and transitioning to easier examples. We do not consider the anti-curriculum strategy because it has been shown to have worse performance than normal curriculum strategy in recent studies [83, 251]. Similarly, another type of strategies for example selection that is different from curriculum learning is example weighting [106, 208]. However, some studies [215, 242] have shown that example weighting methods are more prone to noise and outliers in the datasets, because they tend to select the hardest examples which might actually be noise and outliers. In this chapter, we focus on curriculum learning due to its suitability for handling complex real-world scenarios where selective classification is usually applied.

6.3 Background

In this section we describe the problem formulation for selective classification, the selective neural network model and a baseline method for training it.

6.3.1 Selective Classification Problem Formulation

Here, we consider the multi-class classification setting. Let $\mathcal{X}$ be the input feature space, $\mathcal{Y}$ the label space and $P(\mathcal{X}, \mathcal{Y})$ a distribution over $\mathcal{X} \times \mathcal{Y}$. A predictive model is a function $f : \mathcal{X} \to \mathcal{Y}$. Given a loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$, the risk of the model f is $\mathbb{E}_{P(\mathcal{X},\mathcal{Y})}[l(f(x), y)]$. In the classical supervised learning, the goal is to find the model f such that the model risk is minimized.

In this chapter, we consider the selective model which is a pair of functions (f, g) where $f : \mathcal{X} \to \mathcal{Y}$ is the predictive model, and $g : \mathcal{X} \to \{0, 1\}$ is a selection function. The selection function is used to decide whether each example x is accepted or not:

$$(f, g)(x) = \begin{cases} f(x), & \text{if } g(x) = 1; \\ \text{ABSTAIN}, & \text{if } g(x) = 0. \end{cases} \tag{6.1}$$

Thus, the selective model rejects (i.e., abstains from) an example x iff $g(x) = 0$. Note that a soft selection function $g : \mathcal{X} \to [0, 1]$ can also be considered, from which decisions are taken probabilistically or deterministically using a threshold. The selective risk of (f, g) is

$$R(f, g) = \frac{\mathbb{E}_P[l(f(x), y)g(x)]}{\mathbb{E}_P[g(x)]}. \tag{6.2}$$

The coverage $\phi(g)$ is defined as $\mathbb{E}_P[g(x)]$, i.e., the ratio of the number of accepted examples among all examples. Clearly, there is a trade-off between risk and coverage. In other words, a lower risk could be achieved by sacrificing the coverage. The entire performance of a selective model can be measured via its risk-coverage curve [59]. As with [72], we adopt the goal of finding the model (f, g) such as its selective risk is minimized given a target coverage c.

Given a labeled set $S = \{(x_i, y_i)\}_{i=1}^m$, there are empirical counterparts for the selective risk and coverage. The empirical selective risk is

$$\hat{r}(f, g|S) = \frac{\frac{1}{m}\sum_{i=1}^m l(f(x_i), y_i)g(x_i)}{\hat{\phi}(g|S)}, \tag{6.3}$$

and the empirical coverage is

$$\hat{\phi}(g|S) = \frac{1}{m}\sum_{i=1}^m g(x_i). \tag{6.4}$$

6.3.2 Selective Neural Network Model

In this chapter, we focus on the neural network model SELECTIVENET from [72] because of its impressive empirical performance. SELECTIVENET optimizes both f and g in a single neural network model. The main body block can be any type of architecture that is typically used for solving the problem at hand. For example, convolutional neural networks or residual neural networks are typically used for image classification [84, 120], and transformer networks are typically used for natural language understanding [53]. SELECTIVENET has three output heads for prediction f, selection g and auxiliary prediction h, respectively. The prediction head f and selection head g are the two functions previously defined defined in Section 6.3.1. The auxiliary head h handles a prediction task with the goal of exposing the main body block to all training examples throughout the training process. As with Geifman and El-Yaniv [72], the auxiliary head h uses the same prediction task assigned to f in this chapter. We only need h during training by adding its loss function (defined later in Equation 6.6) to the main selective classification loss (Equation 6.5). Given a training set $S = \{x_i, y_i\}_{i=1}^m$ and a target coverage c, the selective training objective for the heads f and g is

$$\mathcal{L}_{(f,g)} = \hat{r}(f, g) + \lambda \max(0, c - \hat{\phi}(g|S))^2 \tag{6.5}$$

where λ is a hyper-parameter. The standard loss function is used for auxiliary head h:

$$\mathcal{L}_h = \frac{1}{m} \sum_{i=1}^{m} l(h(x_i), y_i). \tag{6.6}$$

Thus, the overall training objective is

$$\mathcal{L} = \alpha \mathcal{L}_{(f,g)} + (1 - \alpha)\mathcal{L}_h, \tag{6.7}$$

where α is a hyper-parameter controlling the relative importance of the above two losses. A stochastic gradient descent (SGD)-type optimization method is used to minimize the above objective [72], which is an iterative process. At each iteration, a mini-batch of examples are picked from the entire

training set, which are used to compute the gradient to update the model parameters. We treat this training method as our baseline, and propose a new method inspired by curriculum learning in the next section.

It is worth noting that, **regardless of the value of the target coverage c, *all* examples participate and are accounted for in minimizing our training objective (Equation 6.7)**. This is because the selection head g (used to decide whether an example should be accepted or not) is trained over *all* examples. Furthermore, the auxiliary head is optimized over *all* examples as it has shown to improve the main body block training [72]. Once the training is completed, the auxiliary head is no longer needed. We keep the prediction head f and selection head g together with the main body block, and use them for selective classification according to Equation 6.1.

6.4 Curriculum-inspired Training Method

Inspired by curriculum learning, we propose a new method for training the selective neural network model. Our method is presented in Algorithm 6.1. The key idea here is to (1) construct mini-batches by calibrating the ratio of easy and hard examples, and (2) exploit the difficulty ordering at the mini-batch level, i.e., control the order in which the mini-batches are exposed by gradually narrowing the difficulty gap between easy and hard examples within each mini-batch. These steps are discussed in Sections 6.4.1 and 6.4.2, respectively. In Section 6.4.3, we discuss our approach to estimate the example difficulty scores.

Algorithm 6.1 Curriculum-inspired method for training selective neural network.

Input: training set S, target coverage c, mini-batch size b, pacing function p and scoring function q

Output: sequence of mini-batches $[B_1, B_2, \cdots]$

Sort S according to q in ascending order //easier examples are before the harder examples

$result \leftarrow []$

for $i \leftarrow 1$ **to** T **do** //T is the total number of iterations

    $size \leftarrow p(i)$;

    $size_E \leftarrow \lceil p(i) * c \rceil$; //the size of easy active set S_E

    $size_H \leftarrow size - size_E$; //the size of hard active set S_H

    $S_E = S[: size_E]$; //select the first $size_E$ examples from the ordered training set S

    $S_H = S[-size_H :]$; //select the last $size_H$ examples from ordered training set S)

    $B_E \leftarrow$ uniformly sample $\lceil b * c \rceil$ examples from S_E;

    $B_H \leftarrow$ uniformly sample $b - \lceil b * c \rceil$ examples from S_H;

    $B_i = B_E \bigcup B_H$; //form the mini-batch for the i-th iteration

    append B_i to $result$;

end for

Return: $result$

6.4.1 Mini-batch Construction

Since we are using a stochastic gradient descent (SGD)-type optimization method to train the selective neural network model, we need to construct a mini-batch of examples to be exposed at each iteration. We can show[2] that, when the mini-batch has the proportion of easy examples different from the target coverage, as a result of minimizing the loss during the training, even an already perfect selection function tends to move to the direction where mistakes will be made (i.e., reject easy examples or accept hard examples). Therefore, a mini-batch which has the proportion of easy

[2]Detailed explanations are included in Section D.2 in the appendix.

examples different from the target coverage will misguide the selection function regarding which examples should be accepted or rejected, which further affects the prediction head's learning progress. Fortunately, we can leverage the estimated difficulty scores to alleviate this issue. Specifically, we partition all the training examples into an easy set and a hard set according to their estimated scores. Assume the mini-batch size is b, the total number of iterations is T and the total number of training examples is m. The easy set contains $m * c$ examples with the lowest difficulty scores, whereas the hard set contains the rest of the examples. We construct the mini-batch by selecting $\lceil b * c \rceil$ examples from the easy set, called B_E, and $b - \lceil b * c \rceil$ examples from the hard set, called B_H, as shown in Lines 11 and 12 of Algorithm 6.1. The union of B_E and B_H becomes the mini-batch B_i for the i-th iteration (Algorithm 6.1 Line 13). Here, we use the $\lceil \cdot \rceil$ operator because $b * c$ may not be an integer. By calibrating the ratio of easy and hard examples in each mini-batch in the above manner, we can ensure that the proportion of easy examples is roughly c.

6.4.2 Mini-batch Ordering

Curriculum learning techniques suggest exposing easy examples to the model early on in the training process, and gradually transitioning to hard examples. We still adopt the curriculum idea, but we argue that in a selective neural network context, curriculum should be designed at the batch level. We cannot expose examples exactly the same way that curriculum learning does because our mini-batch should include both easy and hard examples with their ratio appropriately calibrated to meet the coverage constraint, as explained earlier in subsection 6.4.1. We therefore need to generalize curriculum learning to impose the ordering at the mini-batch level, i.e., by exposing easy mini-batches to the model early in the training process. The key intuition is as follow. **We consider a mini-batch easy for the model (especially its selection head) if its constituent easy examples are clearly separate from its constituent hard examples in terms of their difficulty scores.** Similarly, we consider a mini-batch hard if its constituent easy examples are close to its constituent hard examples in terms of their difficulty scores. We propose to expose easy mini-batches to the model early in the training and gradually transition to hard mini-batches. This enables us

to gradually narrow down the difficulty gap between easy and hard examples. Similar to other curriculum learning techniques [83], we also need to provide two functions as the input to our curriculum-inspired method: a pacing function p and a scoring function q. A pacing function p is used to control the curriculum pace, i.e., the speed at which we transition from easy to hard mini-batches. A scoring function q is used to indicate how hard each example is (i.e., estimated difficulty score), which will be explained in the next subsection.

Specifically, the pacing function p will output an integer $p(i)$ for each iteration i, which we then use to decide how many examples to consider via uniform sampling when constructing the i-th mini-batch (Algorithm 6.1 Line 6). In this chapter, we use the fixed exponential pacing function from [83] because of its good performance in practice. At the i-th iteration, instead of picking the $p(i)$ easiest examples from the training set, we choose $\lceil p(i) * c \rceil$ easiest examples as the *easy active set*, called S_E, and $p(i) - \lceil p(i) * c \rceil$ hardest examples as the *hard active set*, called S_H (Algorithm 6.1 Line 9 and Line 10). Then, we uniformly sample $\lceil b * c \rceil$ examples from S_E, called B_E, and uniformly sample $b - \lceil b * c \rceil$ examples from S_H, called B_H (Algorithm 6.1 Line 11 and Line 12). The union of B_E and B_H becomes the mini-batch B_i for the i-th iteration (Algorithm 6.1 Line 13). The entire process is also illustrated in Figure 6.1.

6.4.3 Difficulty Score Estimation

We use the following approach for getting the scoring function q, which outputs a scalar value $q(x)$ for each example x as its estimated difficulty score. First, we train the same model using uniformly sampled mini-batches (i.e., vanilla method), and run each example through it to get the activation levels of its main body output as its condensed representation. We then run an outlier detection algorithm, Minimum Covariance Determinant (MCD) [190], to compute the Mahalanobis distance for each example with respect to all examples that belong to the same class with this example. The idea behind MCD is to find a subset of examples whose empirical covariance has the smallest determinant, hence resulting in the tightest representation of the overall dataset, which may otherwise be quite noisy and have outliers. This subset of examples is then used to compute the

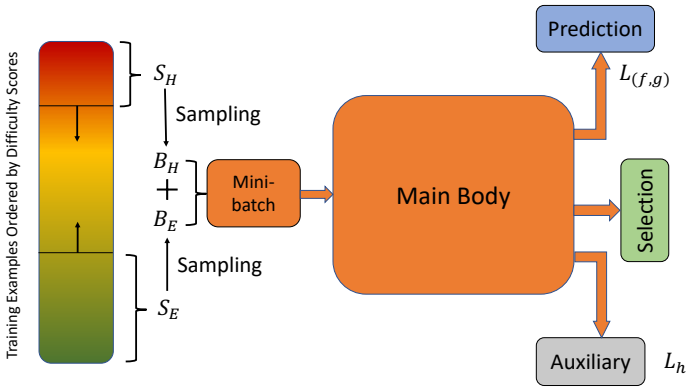


Figure 6.1: Diagram of our curriculum-inspired method for training the selective neural network model. The training examples are ordered by their difficulty scores as shown in the left-most box. Hard examples are at the top (red color) and easy examples are at the bottom (green color). At each iteration, the mini-batch is constructed by sampling from both the hard part and the easy part, with their ratio appropriately calibrated. As training process goes, we gradually narrow the difficulty gap between easy and hard examples, indicated by the boundaries of the easy and hard parts moving towards each other in the diagram.

center location μ and covariance Σ. For any example x, its Mahalanobis distance is computed as $d(x) = (x - \mu)^T \Sigma^{-1} (x - \mu)$. The larger the Mahalanobis distance $d(x)$, the noisier the example x. We thus treat $d(x)$ as the scoring function output $q(x)$ for example x, i.e., $q(x) = d(x), \forall x$.

In practice, the overhead of this new approach for difficulty score estimation should not become a concern. Firstly, the model using uniform sampling (i.e., without rejection) needs much less iterations to get an estimation of difficulty scores than the model with rejection. In all of our experiments, we use half of the iterations compared to training the model with rejection. Secondly, we run the MCD separately for the examples belonging to each class. The number of examples in each class is typically small enough for running MCD (e.g., 5k for CIFAR10, 7k for SVHN and 500 for CIFARI100). Both factors limit the overhead of our approach. What's more, the overhead of getting difficulty score estimation can be greatly reduced by simply using an exiting approach based on off-the-shelf pre-trained models (e.g., **Inception** in our ablation study in Section 6.5.3). Equipped with either approach for getting difficulty score estimation, our method can outperform the baseline. However, our new difficulty estimation approach leads to better empirical performance. We think this is because MCD offers the robustness needed for handling complex datasets (which

Table 6.1: Results of moderate-sized CNN model on CIFAR10. Testing classification error rates (in $\%$) are shown for different target coverage rates. The relative improvement of error rates of our method w.r.t. SelectiveNet is shown at the last column.

Coverage	**Error Rate**		**Impr.**
	Our Method	**SelectiveNet**	
0.95	$\mathbf{7.32 \pm 0.05}$	7.57 ± 0.1	3.30%
0.9	$\mathbf{5.63 \pm 0.04}$	6.07 ± 0.1	7.25%
0.85	$\mathbf{4.30 \pm 0.04}$	4.53 ± 0.08	5.08%
0.8	$\mathbf{3.02 \pm 0.02}$	3.47 ± 0.07	12.97%
0.75	$\mathbf{2.16 \pm 0.04}$	2.45 ± 0.07	11.84%
0.7	$\mathbf{1.47 \pm 0.02}$	1.67 ± 0.08	11.98%

might be noisy with outliers).

6.5 Evaluations

In this section, we evaluate our proposed method and compare it against a state-of-the-art method for training selective neural network models. Similar to [72], we also report both the testing classification error rates (which is the same as selective risk values when using 0-1 loss) as well as the relative improvements of the error rates. In summary, our experiments show the following [3]: (1) our method achieves a lower error rate (up to 10% lower) than the state-of-the-art for the same target coverage rate for a variety of different main body network architectures and datasets (section 6.5.2); (2) our method is better than other alternative designs using difficulty scores or example selection from existing curriculum learning literature (section 6.5.3).

6.5.1 Setup

Datasets. Similar to prior work, we use the following three datasets: **CIFAR10** [119], **CIFAR100** [119] and **SVHN** [165] in our experiments.

[3] In the appendix, we also show that (i) our method has a faster convergence speed (up to 50% faster) than the state-of-the-art (section D.5), and (ii) our method can reduce the coverage violation (i.e., the difference between target coverage and empirical coverage) by more than 10% (section D.6)

Table 6.2: Results of moderate-sized CNN model on CIFAR100. Testing classification error rates (in %) are shown for different target coverage rates. The relative improvement is shown at the last column.

Coverage	Error Rate		Impr.
	Our Method	SelectiveNet	
0.95	$\mathbf{30.63 \pm 0.06}$	31.13 ± 0.22	1.61%
0.9	$\mathbf{28.95 \pm 0.08}$	29.05 ± 0.14	0.35%
0.85	$\mathbf{27.07 \pm 0.08}$	27.32 ± 0.09	0.92%
0.8	$\mathbf{25.00 \pm 0.05}$	25.39 ± 0.07	1.54%
0.75	$\mathbf{23.16 \pm 0.03}$	23.67 ± 0.05	2.15%
0.7	$\mathbf{21.28 \pm 0.02}$	21.97 ± 0.03	3.14%

Table 6.3: Results of moderate-sized CNN model on SVHN. Testing classification error rates (in %) are shown for different target coverage rates. The relative improvement is shown at the last column.

Coverage	Error Rate		Impr.
	Our Method	SelectiveNet	
0.95	$\mathbf{1.91 \pm 0.03}$	2.08 ± 0.02	8.17%
0.9	$\mathbf{1.03 \pm 0.04}$	1.12 ± 0.06	8.04%
0.85	$\mathbf{0.63 \pm 0.02}$	0.66 ± 0.01	4.55%
0.8	$\mathbf{0.49 \pm 0.03}$	0.51 ± 0.02	3.92%
0.75	$\mathbf{0.41 \pm 0.01}$	0.42 ± 0.03	2.38%
0.7	$\mathbf{0.35 \pm 0.01}$	0.39 ± 0.01	10.26%

Baseline method and architectures. We compare our method against SELECTIVENET [72], which is considered the state-of-the-art. We vary the target coverage rate c with 6 different values: $0.95, 0.9, 0.85, 0.8, 0.75, 0.7$. We use two different neural network architectures for the main body of our selective neural network model: moderate-sized convolutional network architecture [83] and VGG-16 architecture [211]. Details about these architectures can be found in the appendix.

Training details. We follow the settings from [72] whenever possible. We train each model for 300 epochs, with an initial learning rate of 0.1 which is reduced by half every 25 epochs. The model is optimized using stochastic gradient descent (SGD) with momentum 0.9 and weight decay $5e-4$. All experiments are repeated 3 times similar to SELECTIVENET. We report both the average value and standard deviation (following $\pm$). The classification error rate that we report in our experiments is the percentage of incorrectly predicted examples among all the accepted examples, by evaluating the

Table 6.4: Results of VGG model on CIFAR100. Testing classification error rates (in $\%$) are shown for different target coverage rates. The relative improvement is shown at the last column.

Coverage	Error Rate		Impr.
	Our Method	**SelectiveNet**	
0.95	$\mathbf{28.41 \pm 0.07}$	28.92 ± 0.09	1.76%
0.9	$\mathbf{25.67 \pm 0.04}$	25.78 ± 0.02	0.43%
0.85	$\mathbf{22.94 \pm 0.02}$	23.56 ± 0.08	2.63%
0.8	$\mathbf{21.40 \pm 0.03}$	21.99 ± 0.02	2.68%
0.75	$\mathbf{18.25 \pm 0.01}$	18.91 ± 0.08	3.49%
0.7	$\mathbf{16.09 \pm 0.02}$	16.32 ± 0.01	1.41%

selective model on the hold-out test sets. Note that classification error rate is the same as empirical selective risk value when using the 0-1 loss. All experiments are run using NVIDIA V100 GPUs.

Post-training calibration. We report the calibrated selective error rate as the metric for comparing against the baseline for each target coverage value. To get the calibrated selective error, we estimate an appropriate threshold τ for selection head output values on a validation set, and use the following calibrated rule for our predictions:

$$(f, g)(x) = \begin{cases} f(x), & \text{if } \; g(x) \geq \tau; \\ \text{ABSTAIN}, & \text{otherwise .} \end{cases} \tag{6.8}$$

This calibration process is used to correct for the difference between the post-training empirical coverage $\hat{\phi}$ and the target coverage c. Specifically, given a validation set V, we set τ to be the $100(1-c)$ percentile of the distribution of $g(x_i), x_i \in V$.

6.5.2 Selective Classification Results

Moderate-sized CNN. As with Hacohen and Weinshall [83], we conduct experiments using moderate-sized convolutional neural networks (CNN) on all three datasets CIFAR10, CIFAR100 and SVHN. The corresponding results are shown in Table 6.1, Table 6.2, and Table 6.3. It can be seen that, as the target coverage rate becomes smaller, the error rate is lower for both our method and the baseline. This behavior is expected because, in general, any selective model should be able

Table 6.5: Ablation study with moderate-sized CNN on CIFAR10. Testing classification error rates (in %) are shown below.

Coverage	Our Method	SelectiveNet	Inception	Vanilla-Selection
0.9	$\mathbf{5.63 \pm 0.04}$	6.07 ± 0.1	5.87 ± 0.05	5.98 ± 0.04
0.8	$\mathbf{3.02 \pm 0.02}$	3.47 ± 0.07	3.14 ± 0.05	3.35 ± 0.04
0.7	$\mathbf{1.47 \pm 0.02}$	1.67 ± 0.08	1.59 ± 0.02	1.66 ± 0.03

to trade-off coverage for error rate, i.e., lower the error by reducing coverage. However, achieving lower error becomes more challenging when the target coverage is smaller because the model needs to figure out which additional examples should be further rejected.

Overall, our method outperforms the baseline by achieving lower selective error rates for each target coverage rate (corresponding to each row in the above mentioned tables). Our method consistently improves upon the baseline across all three datasets. These improvements are even higher for CIFAR10 and SVHN (up to 10%), especially when the target coverage is small. This demonstrates that our method is very effective at trading off the coverage for even lower error rate.

VGG. We also conduct experiments using the VGG-16 model [211] on the CIFAR100 dataset, because of VGG-16's strong prediction ability which is desired for the CIFAR100 dataset. VGG is a larger image recognition model with more layers. As shown in Table 6.5, our method still outperform the baseline SELECTIVENET consistently across all different target coverage rates. This verifies that our method can be applied to different backbone models. It is worth noting that, compared with the moderate-sized CNN on CIFAR100 shown in Table 6.2, the error rate for either method is smaller smaller in Table 6.5 for the same coverage rate. This is expected because VGG model is larger in size (more layers). In addition, the average improvement from Table 6.2 is 1.61% while the average improvement from Table 6.5 is 2.07%. This implies that our method can achieve an even greater improvement with larger models.

6.5.3 Ablation Study

We conduct ablation study to validate our design decisions by comparing our method against two variations of it. Note that our method is composed of two components: (1) estimating the

difficulty scores, and (2) selecting examples for each mini-batch during training. To show that both components play an important role in the overall success of our method, we implement two variants, each using one component from existing work. **Inception** is a variant of our method using the pre-trained inception model for difficulty estimation (i.e., *transfer scoring function* in [83]). **Vanilla-Selection** is another variant of our method using the vanilla method of selecting examples for each mini-batch that is typically used in existing curriculum literature. Different from our method which considers ordering at the mini-batch level, existing curriculum learning literature typically consider ordering at the example level while selecting examples for each mini-batch. Specifically, they construct the mini-batch by uniformly sampling from a working subset, which starts with a few easiest examples, and is being appended with harder and harder examples after every several epochs until all the training examples are added [215, 242]. We adopt this strategy in **Vanilla-Selection** which can be viewed as applying the vanilla curriculum learning (e.g., Algorithm 1 from [83]) to selective classification domain.

As reported in Table 6.5, both variants have higher error rates than our method, validating that both components are integral to and contribute to our method's impressive performance. Especially, applying the vanilla curriculum learning to the selective classification domain as in **Vanilla-Selection** is not as good as our method which has a specially tailored curriculum learning strategy. In addition, even though our method involves an overhead of model training in order to get the difficulty score estimation, this overhead is paid only once even if the coverage needs to change. **Inception** is the variant without this overhead since the off-the-self pre-trained inception model can be readily obtained online. If one does not wish to pay the upfront overhead of our method, these results imply that even **Inception** outperforms the baseline.

6.6 Summary

In this chapter, we proposed a curriculum-inspired method to train selective neural network models by leveraging difficulty scores. Our method calibrates the ratio of easy and hard examples in

each mini-batch to match what is best desired by the training objective. We generalized curriculum learning by utilizing difficulty scores to reorder mini-batches rather than simply reorder examples. Extensive experiments showed the superior performance of our method compared to state-of-the-art SELECTIVENET.

Part III

Deciding Token Routing

Chapter 7

Communication Efficient Regularization for Mixture-of-Experts Transformers

7.1 Motivation

In this chapter, we consider a new type of transformer models called Mixture-of-Experts (MoE) transformers which set new state-of-the-art achievements in many NLP tasks. Although large dense transformer models have been shown to be effective and powerful, sustaining the continued growth in size for these dense models requires tremendous compute resources, since all the parameters are used for all input examples. Seeking better computational efficiency, sparsely activated models have gained great interest; since they utilize a subset of the model weights for each input example [126, 206]. This leads to a desirable characteristic: sparsely activated models can scale up to an order of magnitude larger than dense models without significant increase in computational cost. However, libraries and hardware accelerators widely used by machine learning practitioners are much efficient in supporting dense matrix operations. Mixture of Experts (MoE) models are a specific type of sparsely activated models that have had notable success in machine translation [126, 207] and recently been explored in other tasks [188] while running efficiently on top of existing libraries and hardware accelerators. In this chapter, we focus on multilingual machine translation tasks.

A dense transformer model consists of a stack of layers composed of two sub-layers: a multi-head self-attention sub-layer followed by a feedforward network (FFN) sub-layer. An MoE model replaces the FFN sub-layer with a Mixture-of-Experts (MoE) sub-layer which takes as input a token representation and routes this token to the best top-k experts. The MoE sub-layer is essentially a group of experts with each expert still a FFN sub-layer. Typically, k is a very small value. As with [64, 111], we assume $k = 1$ by default. Note that our method can also be extended to the case when $k > 1$. During training and inference, an input token is routed to only one expert ($k = 1$), resulting in constant computational cost for one forward pass regardless of the total number of experts. The target expert for each token is determined by a gating network which is typically a small one-layer FFN.

To achieve highly efficient training, a combination of data, expert, and model parallelism is adopted: (1) the part of the model excluding MoE sub-layers is replicated across multiple machines in a data-parallel fashion, (2) experts from each MoE sub-layer are split across machines (i.e., different machines hold the parameters of different experts), and (3) techniques such as tensor slicing[1] are used to partition tensors among multiple machines if they are too large for a single machine (e.g., partition along the d_{ff} dimension inside each expert when d_{ff} is too large) [64]. Since tokens and their assigned experts are likely to reside at different machines, we need to shuffle the tokens around via the communication link among machines. This is typically achieved by using the all-to-all collective operation, which is very costly even if the data is cast to bfloat16 precision [64]. If we assume that token representation is of d dimension, the sequence length is L, total batch size is B, then the total amount of data that needs to be handled by the all-to-all operation is $2BLd$ bytes[2]. As a commonly used setting in practice, we assume $d = 4096 = 2^{12}, L = 1024 = 2^{10}, B = 128 = 2^7$, then $2BLd = 2^{1+12+10+7} = 2^{30} = 1G$. In other words, the all-to-all operation needs to shuffle roughly $1GB$ of data at each MoE sub-layer for each forward pass. Therefore, the communication cost from all-to-all operations is greatly affecting the training speed for MoE models.

[1]To understand the key idea of Gating Dropout, readers can focus on data and expert parallelism (i.e., ignore model parallelism such as tensor slicing) to simplify the complexity of parallelism.

[2]The factor 2 comes from the bfloat16 data representation, which uses 2 bytes per number.

In this chapter, we propose *Gating Dropout* as a communication-efficient regularization technique for training sparsely activated transformers, specifically for MoE models. Gating Dropout allows tokens to ignore the assignments from the gating network and stay at their local machines with certain probability, thus reducing the need for routing the tokens to experts at different machines. This significantly improves the throughput and convergence speed for MoE training. In addition, by forcing tokens to be routed to experts on the same machines (which might not be the optimal experts), we encourage the experts to learn a general ability. We think this leads to a regularization effect during training, resulting in better generalization performance when training is converged.

To validate the effectiveness of Gating Dropout, we have conducted extensive experiments on multilingual machine translation tasks using different model settings. With Gating Dropout, our results show that we improve a state-of-the-art MoE model [111] in terms of convergence speed (up to 58% less training time to target BLEU score), generalization performance (up to 0.6 BLEU score increase) and throughput (up to 16% more tokens per second).

7.2 Background

7.2.1 From Densely Activated Transformers to Sparsely Activated Transformers

In Vaswani et al. [233], the transformer architecture was introduced to the NLP community due to their superior performance in sequence-to-sequence tasks, such as neural machine translation. A sequence-to-sequence transformer contains an encoder module and a decoder module. The encoder module is a stack of encoder layers, each employing a self-attention sub-layer followed by an FFN sub-layer. The decoder module is a stack of decoder layers, each employing a self-attention sub-layer followed by a cross-attention sub-layer with an FFN sub-layer at the end. Ever since, transformer-based models have been the top performers in various NLP tasks, such as BERT [53], RoBERTa [150], GPT-3 [25], XLNet [256], ELECTRA [37] and TMR [142]. However, these

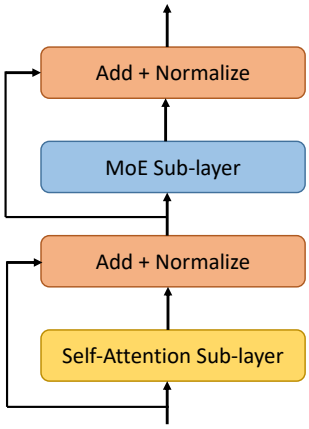


Figure 7.1: The MoE model architecture. The FFN sub-layer in the original transformer architecture is replaced by the MoE sub-layer, which consists of experts.

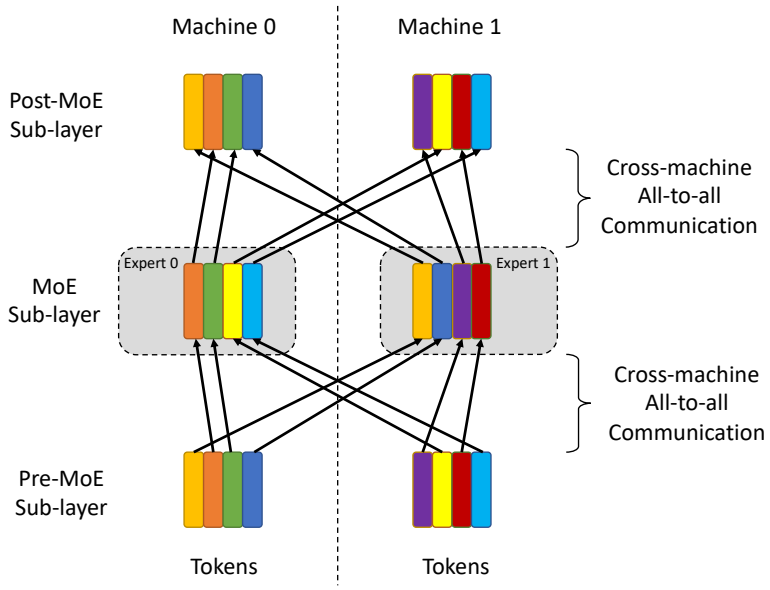


Figure 7.2: The all-to-all communication illustration of an MoE model with two experts in a two-machine setting. Expert 0 and 1 reside at machine 0 and 1, respectively. Each machine has a local batch of 4 tokens to process. All-to-all communication is needed right before (and right after) MoE sub-layer.

transformer models are densely activated, meaning that all model parameters are used to process all input examples. Scaling up the model size for them results in significant increase in computational cost during both training and inference. Therefore, the level of scaling depends on the availability of compute resources (e.g., GPU or TPU), which can be costly to obtain. For example, it is estimated that it takes 168 days to train a GPT-3 model with 178 billion parameters using 256 NVIDIA A100 GPUs [163]. Megatron-Turing NLG 530B model with 530 billion parameters had been trained on NVIDIA's Selene supercomputer with 560 DGX A100 nodes, each node having 8 NVIDIA 80GB A100 GPUs connected to each other with high bandwidth Infiniband configuration [213].

On the other hand, sparsely activated transformers use a subset of model parameters for each different input example, leading to outrageous scaling capability. An Mixture-of-Experts (MoE) [64] model has been scaled to 1.5 trillion parameters while GPT-3, one of the largest dense models, contains only 175 billion parameters. The core component of MoE models is the Mixture-of-Experts (MoE) sub-layer, which replaces the FFN sub-layer in the dense transformers, as illustrated in

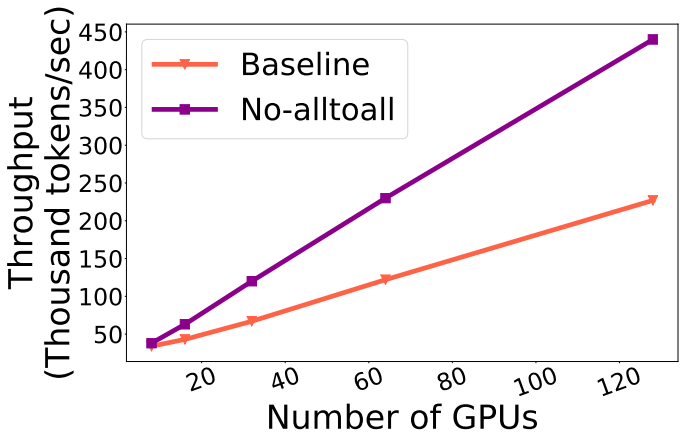


Figure 7.3: The all-to-all communication cost.

Figure 7.1. Let $\{E_i\}_{i=1}^{N}$ denote the set of experts at an MoE sub-layer where N is the total number of experts. Each expert E_i is still a feedforward network (FFN). The parameters of one expert is independent from those of another expert, meaning that they can be trained to converge to different values. Each token is routed to the top-k experts, which are determined by a gating network. Given a token $x \in \mathbb{R}^d$, where d is the hidden dimension, the output of a gating network is logits $h(x) = W_r \cdot x$, followed by a normalization via a softmax, i.e.,

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_{j=1}^{N} e^{h(x)_j}}, \tag{7.1}$$

where $W_r \in \mathbb{R}^{N \times d}$ is the trainable weight matrix of the gating network. Given the gating network output $\{p_i(x)\}_{i=1}^{N}$, we select the top-k experts to form the set of activated experts $\mathcal{T} \subset \{1, \cdots, N\}$, where $|\mathcal{T}| = k$. In other words, the token x is assigned to experts in $\mathcal{T}$. Typically k is much smaller than N. In Fedus et al. [64], it is shown that using $k = 1$ is sufficient with better computational efficiency. In this chapter, we assume $k = 1$ by default. Then the output of an MoE sub-layer for the input token x is

$$y = \sum_{i \in \mathcal{T}} p_i(x) E_i(x). \tag{7.2}$$

Table 7.1: Relative throughput improvement of no-alltoall compared to baseline.

Number of GPUs	Throughput Impr.
8	11.8%
16	46.5%
32	79.1%
64	88.5%
128	93.8%

7.2.2 All-to-all Communication Cost of MoE Models

Distributed training involving multiple machines is typically used to support highly scaled MoE models. Each machine contains a replication of the model parameters excluding experts (i.e., data parallelism), and a subset of experts (i.e., expert parallelism). Since experts are split across multiple machines, each token needs to be sent to the machine where its assigned expert resides. This is typically achieved by using the all-to-all collective communication [64], which is illustrated in Figure 7.2 using a two-machine setting. We conducted some experiments to demonstrate the all-to-all communication cost (see the settings on WMT-10 dataset in Section 7.4.1 for details). Specifically, we use a cluster of NVIDIA V100 GPUs connected via a 100Gb/s InfiniBand fabric. We use a state-of-the-art MoE model [111] as the baseline, and a no-alltoall variant which is the same as baseline but with the all-to-all operations entirely skipped. We compare the throughput in terms of the number of tokens processed per second between the baseline and the no-alltoall variant. To see how the throughput changes as we scale up the model, we increase the number of GPUs from 8 all the way to 128, and set the number of experts the same as the number of GPUs due to expert parallelism [64]. From Figure 7.3, we can see that as we increase the number of GPUs, the throughput for both the baseline model and no-alltoall variant increases. This is expected because using more GPUs provides greater computational power which increase the processing speed. We also observe that the no-alltoall variant has significantly higher throughput than the baseline when using the same number of GPUs. In Table 7.1, we list the relative throughput improvement of the no-alltoall compared to the baseline. As we increase the number of GPUs, the relative throughput improvement also increases. Especially, when the number of GPUs is 128, the relative improvement

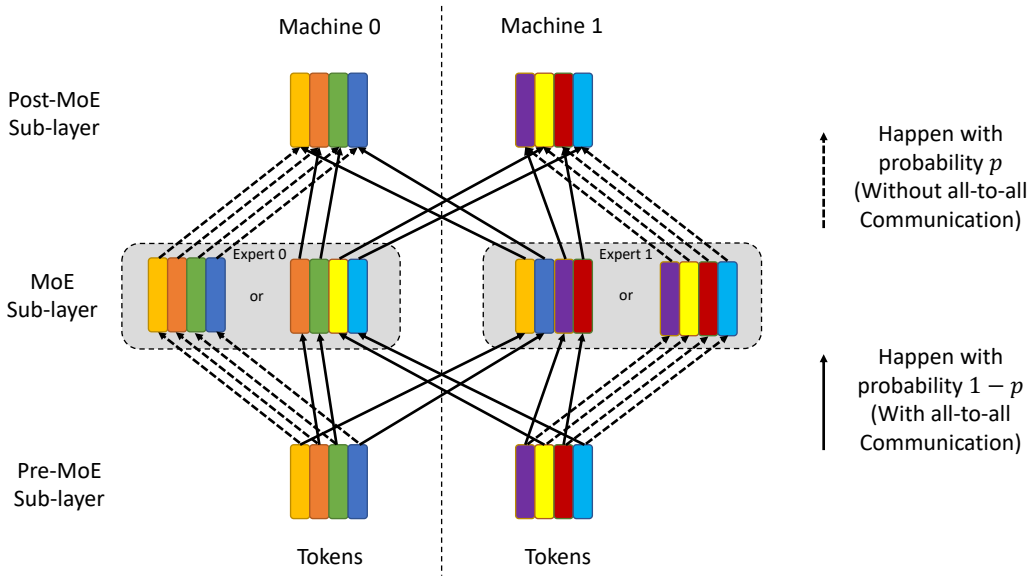


Figure 7.4: The Gating Dropout illustration of an MoE model with two experts in a two-machine setting. Expert 0 and 1 reside at machine 0 and 1, respectively. Each machine has a local batch of 4 tokens to process. Gating Dropout allows us to send tokens to the local experts at the same machines with probability p (denoted by the dashed arrows).

is more than 90%. Note that the difference between the baseline and no-alltoall comes from whether the all-to-all communication in the MoE sub-layers is invoked or not. From these results, we can see that all-to-all communication greatly affects the training speed. As the number of GPUs increases, the all-to-all communication becomes more costly because data exchange involves more machines and communication cost is proportional to the number of involved machines. In summary, to better support the scaling capability of MoE models, it will pay off to investigate methods that can reduce the all-to-all communication cost without sacrificing the performance.

7.3 Gating Dropout as Communication-efficient Regularization

In this section, we describe *Gating Dropout* as an effective way to reduce the all-to-all communication cost. We also show that Gating Dropout has a regularization effect during training, resulting in improved generalization performance. Gating Dropout essentially combines communication efficiency with regularization, which has been challenging to achieve yet desirable in distributed training, where communication cost is often the bottleneck. Gating Dropout works in the following

way, which is also illustrated in Figure 7.4, at each training iteration:

- with probability p, all the tokens are sent to the local experts at the same machine without all-to-all communication involved;
- with probability $1 - p$, all the tokens are sent to their assigned experts with all-to-all communication involved, as is done in normal MoE models.

The above probability p is the dropout rate. Note that, since all-to-all is a collective operation requiring all the machines to simultaneously participate, the decision of turning on/off Gating Dropout should be consensual among all the machines at each iteration. In other words, at each iteration, it is either (1) that all the machines keep their tokens locally (i.e., Gating Dropout is turned on), or (2) that all the machines use the all-to-all operations to send tokens to their assigned experts (i.e., Gating Dropout is turned off). To make sure all machines achieve consensual decisions, we appoint one machine as the coordinator, responsible for making the randomized decision, and broadcasting the decision to all the machines at each iteration. The overhead of broadcasting the decision is negligible, because the decision can be represented by a binary value.

During inference, we simply turn off the Gating Dropout by setting $p = 0$. We do not have the additional weight scaling step as in the traditional dropout (i.e., multiply the weight by p). In the traditional dropout, the weight scaling step is used to ensure the expected neuron output is the same between inference time and training time, because traditional dropout works by setting the neuron output as 0. However, Gating Dropout will not affect expected neuron output because it works by modifying the token routing strategy.

This technique can be easily implemented with several lines of code, by using the broadcast collective operation for broadcasting the decision and the conditional branch for skipping the all-to-all communication. We refer to this basic version of Gating Dropout as Gate-Drop in order to differentiate it from the extended version that will be introduced in Section 7.3.1.

The benefits of Gating Dropout are twofold:

- higher throughput due to reduced communication cost;

- better generalization performance due to its regularization effect.

Communication Efficiency. Gating Dropout reduces the all-to-all communication cost, thus improving the throughput during training. When the Gating Dropout is turned on at a certain iteration, all the tokens go to the local experts at the same machines. In this case, all-to-all communication is no longer needed at this iteration. However, when Gating Dropout is turned off, the all-to-all communication is still used. The dropout rate p essentially controls how often the gating dropout is turned-on on average, thus affecting the throughput during the training process. The no-alltoall variant in Figure 7.3 corresponds to the case when $p = 1$, which provides the upper-bound of the throughput improvement. It is worth noting that it is not possible to achieve this upper-bound since the model will not be able to learn any gating in such settings. Therefore, the model performance will deteriorate when p is too large. Next, we will show that Gating Dropout can lead to better performance when p is properly chosen.

Regularization. Gating Dropout can be viewed as a regularization technique to improve the model's generalization performance. The gating network is used to determine which expert a token should be sent to. The gating network is supposed to select the best expert for each token, which is unknown in general. Therefore, the common practice in MoE models is that the gating network is initialized randomly, and being trained together with all the experts. Intuitively, Gating Dropout can break up the co-training between gating network and experts, just as traditional dropout is used to break up the co-adaptation among neurons [216]. This forces experts to work in a robust way without relying too much on the guidance from the gating network (e.g., by obtaining a general ability). Another perspective that affects the dynamics of training with Gating Dropout comes from exploration-exploitation trade-off [13, 222]. The gating network provides a discrete decision over all the experts. This decision may not be optimal simply because other experts could be under-explored. The epsilon-greedy method [222] is a simple yet effective way to handle the exploration-exploitaion trade-off: with probability ϵ, make a random decision over all candidate options; with probability $1 - \epsilon$, make the greedily best decision based on the available information so far. Gating Dropout can be viewed as a simplified variant of the epsilon-greedy method that is tailored for MoE model

training while accounting for the communication cost. Specifically, instead of choosing a random expert, we choose the local expert at the same machine because the all-to-all communication is not needed to send tokens to their local experts. In our experiments, it is observed that this simplified variant of the epsilon-greedy method actually yields good performance improvement.

Gating Dropout vs Expert Dropout. It is worth noting the differences between Gating and Expert dropout. Expert Dropout is recommended in Fedus et al. [64] as a technique to regularize the MoE models. However, both techniques have quite distinctive characteristics. Expert Dropout is used during fine-tuning on downstream tasks that have very few examples to overcome possible overfitting due to lack of enough examples. However, Gating Dropout is used in pre-training. Furthermore, Expert Dropout is essentially traditional dropout [216] where the dropout rate inside MoE sub-layers is explicitly increased. They observed better performance on several smaller downstream tasks by setting a smaller dropout rate at non-MoE sub-layers and a higher rate at MoE sub-layers. On the other hand, Gating Dropout acts as a regularization technique that is completely different from traditional dropout.

Gating Dropout vs Other Exploration-Exploitation Techniques. The exploration-exploitation trade-off is also considered in Fedus et al. [64]. They investigated several techniques such as sample softmax, input dropout and input jitter. First, none of these techniques is communication efficient, which is one of the main benefits of our Gating Dropout. Second, Gating Dropout is compatible with these techniques, so that they can be complimentary used together during training. For example, input jitter, which has the best empirical performance in their experiments, injects noise to the incoming token representation. Gating Dropout can still be applied regardless of how the token representation is modified. In our experiments, we have adopted input jitter by default in the baseline.

7.3.1 Extension with Expert LayerDrop

It has been shown that skipping certain layers of neural networks during training can reduce the training time and also has a regularization effect [62, 90]. Inspired by LayerDrop proposed

Table 7.2: The results on WMT-10 dataset using models with Transformer-base architecture. The target BLEU score is the value achieved by the baseline method at convergence, i.e., 24.60. The training time to the target BLEU is in seconds. Hash-Layer uses a random hash function to replace the gating network. We use - in the table to indicate that Hash-Layer cannot reach the target BLEU.

Method	Throughput	BLEU@convergence	Time to target BLEU	Steps to target BLEU
Baseline	129k	24.60	250k	74.24k
Hash-Layer	135k	23.06	-	-
Gate-Drop	143k	25.06	147k	48.64k
Gate-Expert-Drop	**150k**	**25.12**	**104k**	**35.84k**

in Fan et al. [62] which randomly drops layers in (densely activated) transformers, we consider an extension of Gating Dropout by dropping experts. Specifically, when Gating Dropout is turned on, all the tokens skip the local experts and directly go to the sub-layer following the MoE sub-layer. When the Gating Dropout is off, we still send the tokens to their assigned experts with all-to-all communication. We call this extended version Gate-Expert-Drop, while the basic version from the previous section is referred to as Gate-Drop. This extension improves the two strengths of Gating Dropout, since (1) skipping experts reduces the computational cost, thus improving the throughput, and (2) skipping layers randomly has been shown with a regularization effect.

7.4 Evaluations

We evaluate our proposed Gating Dropout (both Gate-Drop and Gate-Expert-Drop) and compare it against a state-of-the-art MoE model [111] on multilingual translation tasks. Multilingual large-scale translation tasks have been a good use case for large dense and sparse transformer models due to their high level of complexity that necessitates such powerful models. We use the Z-code M3 model from Kim et al. [111] as our **baseline**, which has been shown better than prior MoE models. All of our implementations are based on DeepSpeed library[3], due to its strong support for distributed training. We show that Gating Dropout can improve the throughput, BLEU score and convergence speed.

[3] `https://github.com/microsoft/DeepSpeed`

7.4.1 Setup

Datasets. Similar to prior work [111], we use the following datasets in our experiments.

- **WMT-10:** We use WMT-10 benchmark dataset which includes bitext data between English and other **10** languages [243]. There are 32.5 million parallel sentences in total.
- **Web-50:** We use a dataset composed of an in-house **web**-crawled parallel dataset augmented with data from CCNet [247]. It contains 700 million parallel sentences in **50** different languages. This dataset is much bigger than the WMT-10 dataset.

Model Settings. We use the settings recommended in [64] to set up the MoE models. Specifically, MoE sub-layers are added to replace every other FFN sub-layers in both the encoder and decoder of the transformer. The capacity factor is set to 1.0 during training and 2.0 during testing [64]. Jittering noise is applied to the token representation right before the gating network. An additional balancing loss with multiplicative coefficient 0.01 is used to better balance the utilization among different experts. We use two different model sizes for experiments on the above two datasets.

- **WMT-10:** We adopt the architectural settings according to Transformer-base architecture with 12 encoder layers and 6 decoder layers [233]. For each MoE sub-layer, we use 128 experts by default. The model contains around 5.6 billion parameters.
- **Web-50:** We adopt the architectural settings according to Transformer-big architecture with 24 encoder layers and 12 decoder layers [233]. For each MoE sub-layer, we use 64 experts by default. The model contains around 10 billion parameters.

Hardware. We use a cluster of NVIDIA V100 GPUs connected via a 100Gb/s InifiBand fabric in most experiments. For some experiments on the Web-50 dataset, we run jobs on a cluster of NVIDIA A100 GPUs connected via a 1.6Tb/s InifiBand fabric. We use 16 GPUs for WMT-10 and 64 GPUs for Web-50 if not stated otherwise.

Training Details. For training, we use Adam as the optimizer with $\beta_1 = 0.9$ and $\beta = 0.99$. The learning rate is set 0.03 with 5000 warm-up steps and inverse square root scheduler as proposed

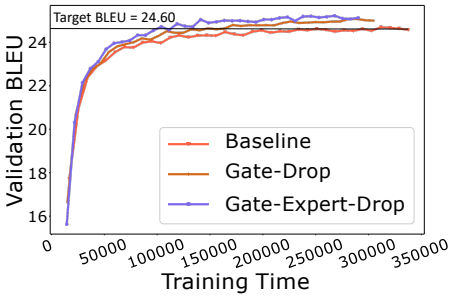


Figure 7.5: BLEU score vs. training time (in seconds) on the WMT-10 dataset. Both Gate-Drop and Gate-Expert-Drop reach the target BLEU (denoted as the black line) much faster than the baseline.

Table 7.3: The throughput results on Web-50 dataset measured in two different clusters.

Method	V100 Cluster	A100 Cluster
Baseline	126k	362k
Gate-Drop	140k	372k
Gate-Expert-Drop	146k	384k

in Raffel et al. [178]. We set the batch size to be equivalent to 435k tokens. For the gating dropout rate p, we set it to 0.3 for Gate-Drop and 0.2 for Gate-Expert-Drop by default, which are chosen because of their good performance (see Section 7.4.4). For training on the Web-50 dataset, we use Denoising Auto-Encoding (DAE) loss [235] together with Machine Translation (MT) loss because multi-tasking learning has been shown to further improve the cross-language knowledge transfer [111, 243].

Metrics. We compare between different methods using the following three metrics:

- **Throughput:** the number of tokens processed per second by the model during training.
- **Generalization Performance:** BLEU score on a holdout validation set. Especially, we report the case-sensitive detokenized sacreBLEU [175].
- **Convergence Speed:** the amount of training time to reach a target BLEU score on a holdout validation set.

Table 7.4: The results on Web-50 dataset using models with Transformer-big architecture. **BLEU (avg)** is the average BLEU score for translating between English and other languages (both directions). **E→X** only considers translating from English to other languages. **X→E** only considers translating from other languages to English. And **(low)** is used to indicate that only low-resource languages are considered.

Method	BLEU (avg)	E→X	E→X (low)	X→E	X→E (low)
Baseline	28.63	23.01	22.15	34.26	33.89
Gate-Drop	**29.22**	**23.86**	**22.87**	**34.59**	**34.34**
Gate-Expert-Drop	28.85	23.22	22.61	34.49	34.22

7.4.2 Results on WMT-10 Dataset

We train MoE models based on the Transformer-base architecture [233] on the WMT-10 dataset for more than 100k steps using different methods. The results are shown in Table 7.2. We see that both Gate-Drop and Gate-Expert-Drop have higher throughput than the baseline method with relative improvement 10.85% and 16.28% respectively. Gate-Expert-Drop has higher throughput than Gate-Drop because computational cost is further reduced by skipping experts.

As for the generalization performance, both Gate-Drop and Gate-Expert-Drop reach BLEU scores that are greater than that of the baseline when the training is converged. The BLEU score improvement is 0.46 for Gate-Drop and 0.52 for Gate-Expert-Drop. The slightly better performance of Gate-Expert-Drop is likely to come from the additional regularization effect of randomly skipping expert layers. From Table 7.2, we also observe that Gate-Drop and Gate-Expert-Drop have faster convergence speed than the baseline. We choose the target BLEU score as the score achieved by the baseline method at convergence (i.e., 24.60). The amount of training time to reach the target BLEU score is also significantly shorter for both Gate-Drop (by 40.89%) and Gate-Expert-Drop (by 58.48%) compared to the baseline. This can also be observed in Figure 7.5, where we show the curves of BLEU score vs. training time. Similarly, both Gate-Drop and Gate-Expert-Drop require smaller number of training steps to reach the target BLEU score compared to the baseline. These results confirm that our methods' fast convergence speed in terms of both the training time and the number of steps.

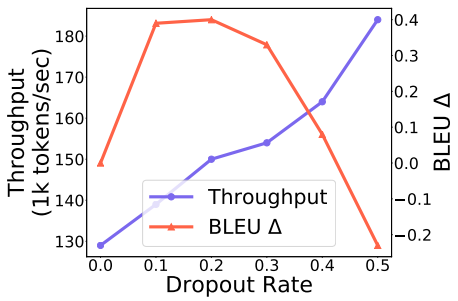


Figure 7.6: The effect of the dropout rate on throughput and BLEU Δ (i.e., difference w.r.t. the baseline's BLEU) for Gate-Expert-Drop on the WMT-10 dataset (measured after training for 35k steps).

In addition, Roller et al. [189] suggest replacing the parameterized gating network with some hash function, which can alleviate the co-training issue between the gating network and experts. As shown in Table 7.2, we also compare against their method Hash-Layer based on a random hash function which is simple to implement but has been shown to have very strong performance [189]. Hash-Layer has slightly better throughput than the baseline, which is expected because invoking a hash function is more computational efficient than passing tokens through a gating network. However, Hash-Layer still needs the all-to-all communication to send tokens to their assigned experts (which are determined by the hash function), which explains why both Gate-Drop and Gate-Expert-Drop still achieve higher throughput than Hash-Layer. We also observe that Hash-Layer does not perform as well as our methods in terms of final BLEU at convergence.

7.4.3 Results on Web-50 Dataset

We train MoE models based on the Transformer-big architecture [233] with 10 billion parameters on the Web-50 dataset for up to 20k steps, it is very time consuming to train such large models for too many steps. In Table 7.3, we report the throughput of different methods measure in two different clusters: the cluster of V100 GPUs with a 100Gb/s InfiniBand fabric and the cluster of A100 GPUs with a 1.6Tb/s InfiniBand fabric. We observe that both Gate-Drop and Gate-Expert-Drop get higher throughput than the baseline in either cluster. It is noted that the relative throughput improvement of our methods in the V100 cluster is more significant because the hardware (GPU for computation

and InfiniBand for communication) is much slower than the A100 cluster. Higher improvement is expected with slower hardware where communication is the bottleneck.

We report the best BLEU scores within 20k steps in Table 7.4. We can see that both Gate-Drop and Gate-Expert-Drop achieve larger BLEU scores than the baseline with the improvement 0.59 for Gate-Drop and 0.22 for Gate-Expert-Drop. Different from the results on the WMT-10 dataset (which is a much smaller dataset), Gate-Drop is better than Gate-Expert-Drop in terms of BLEU scores. We hypothesize that this is probably because Gate-Expert-Drop imposes too much regularization during training, limiting the model's fitting ability on such large scale dataset. We further investigate the results more by looking into the BLEU score for each translation direction, i.e., from English to other languages (E→X) and from other languages to English (X→E). Especially, we consider the performance on only low-resource languages, denoted with **(low)** in Table 7.4. Low-resource languages are those with few training data, such as Galician (Gl), Burmese (My), and Tagalog (Tl). It is difficult to achieve high quality translation performance on these low-resource languages. We see that both Gate-Drop and Gate-Expert-Drop improve upon the baseline on the low-resource language translation tasks by at least 0.3 increase in BLEU score (from Gate-Expert-Drop on **X→E (low)**). And Gate-Drop can yield more than 0.7 BLEU score increase on **E→X (low)**). These results suggest that our methods improve the model's translation quality, especially on low-resource languages which would really benefit from the regularization effect of Gating Dropout.

7.4.4 Effect of Dropout Rate

We investigate the effect of the dropout rate p of Gate-Expert-Drop on throughput and BLEU score using the WMT-10 dataset. Similar observations are expected for Gate-Drop, but we only report results for Gate-Expert-Drop because it outperforms Gate-Drop on the this dataset. We change the dropout rate from 0 to 0.5 for Gate-Expert-Drop. When dropout rate is 0, it is the same as the baseline method, because Gating Dropout will never be turned on in this case. From Figure 7.6, we see that, as we increase the dropout rate, the throughput also increases. This trend is expected because the dropout rate essentially controls how often the all-to-all communication (and the expert

layer) is skipped during training. Larger dropout rate implies more skipping, thus resulting in higher throughput. However, when the dropout rate is too large, the BLEU score might become worse, as shown in Figure 7.6. Specifically, when the dropout rate is 0.5, BLEU score becomes even lower than the baseline (BLEU Δ is -0.23). Actually, the BLEU score is highest when the dropout rate is 0.2, and becomes smaller and smaller as we further increase the rate beyond 0.2. Similar experiments indicate that Gate-Drop achieves highest BLEU score when dropout rate is 0.3. This indicates that the dropout rate needs to be properly chosen in order for our methods to work as expected.

7.5 Related Work

Several techniques have been proposed to improve the efficacy of MoE models. A new routing network which takes as input both the shared token representation and the hierarchical representation from different MoE layers is proposed in You et al. [260] for speech recognition. Expert prototyping proposed in Yang et al. [255] is shown to improve model quality by splitting experts into different prototypes, while Zuo et al. [270] suggest to randomly select experts to encourage learning from other experts as teachers. Linear assignment is used to alleviate the load imbalance issue among experts [127]. As an effective way to regularize model training, the dropout idea has been widely studied for transformer models in general [11, 22, 70, 114]. Several dropout variants have been proposed in different settings, such as BPE-Dropout [176], R-Drop [250], MC-Dropout [68], recurrent dropout [204], curriculum dropout [159] and so on. Unlike these dropout variants, our method considers the communication cost which is crucial in the distributed training setting.

7.6 Summary

We proposed Gating Dropout as a communication-efficient regularization technique to train sparsely activated transformers. We observed that sparsely activated transformers, such as MoE models, typically have very high cross-machine communication cost, since they need to send tokens

to their assigned experts via the all-to-all communication operations. Gating Dropout reduces the communication cost by randomly skipping the all-to-all operations. This random skipping also has a regularization effect during training, leading to improved generalization performance. Experiments on multilingual translation tasks demonstrated the effectiveness of the proposed method in terms of throughput, generalization performance and convergence speed.

Chapter 8

Conclusions

In this thesis, we considered several settings in which sequential decision making can make large scale machine learning less expensive. These settings were categorized into three types, depending on what the decisions meant in each setting.

The first type of decision was coordinate selection. We considered both the Maximum Inner Product Search (MIPS) problem and the distributed learning problem. For MIPS, we cast this problem into the framework of best arm identification which allowed us to select as few coordinates for multiplication operations as possible. We proved that our method is guaranteed to return an ϵ-approximate solution and have fewer multiplication operations than the naïve exhaustive search method. For distributed learning, a common way to reduce the communication cost among machines is gradient sparsification, which drops some less important coordinates of vectors to be sent. We used bandit-based methods to select coordinates to drop at each training iteration. We proved that our methods have the same asymptotic convergence rate with the baselines. Our experiments also demonstrated that our methods converge faster than the baselines in terms of the wall-clock time.

The second type of decision was example selection. We first considered sample-efficient transformer training. Specifically, we used memory replay to alleviate the *distribution drift* issue of ELECTRA, which is a sample efficient variant of BERT. Memory replay allowed us to save past examples and reuse the high-quality ones. Our experiments showed that our method achieves higher accuracy than the baselines within the same amount of time. We then presented an efficient method

for accelerating the training of Adam by endowing it with bandit sampling, which allowed Adam to select more important examples at each iteration. We provided a theoretical analysis to show that our method can improve the convergence rate of Adam in some settings. Our experiments further demonstrated that our method is effective in reducing the training time for several tasks with different deep learning models. Lastly, we proposed a method to train selective neural network models by generalizing curriculum learning–a technique that selects and reorders examples according to the easy-to-hard curriculum. Our experimental results demonstrated that our method outperforms both the state-of-the-art and alternative methods that use vanilla curriculum techniques for training selective neural network models.

The third type of decision was token routing. This type is about how to route tokens (i.e., vector representations of words in NLP) across different machines. We considered a recent transformer variant, called MoE transformer, due to its outrageous scaling capability, leading to a state-of-the-art performance in several NLP tasks such as machine translation. Our method reduced the communication cost by randomly skipping the all-to-all communication operations. This random skipping also has a regularization effect during training because it can be viewed as a variant of the ϵ-greedy method in multi-armed bandit. Experiments on multilingual translation tasks demonstrated the effectiveness of the proposed method in terms of throughput, generalization performance and convergence speed.

Future work. There are two different directions to expand this dissertation.

The first direction is to further improve our methods, e.g., in terms of better sample efficiency or lower overhead. For example, our token routing strategy for MoE transformers uses the same probability value to skip communication throughout the training process. It might be better to use varying probability values so that more exploration is encouraged at the early stage of training.

The second direction is to consider new training settings. The field of machine learning is still rapidly developing. New model types and new training methods are being invented from time to time. We are interested in the potential of sequential decision making being applied to other training settings. For example, modern deep learning-based recommendation systems have huge embedding

tables, consuming a large amount of memory [161, 253]. Model pruning is a promising way to reduce the model size. Therefore, selecting which embedding rows to prune during training is worth investigating.

Bibliography

[1] ChatGPT. `https://openai.com/blog/chatgpt/`.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[4] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.

[5] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[6] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[7] Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

[8] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[9] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*, pages 5973–5983, 2018.

[10] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

[11] Raman Arora, Peter Bartlett, Poorya Mianjy, and Nathan Srebro. Dropout: Explicit forms and capacity control. In *International Conference on Machine Learning*, pages 351–361. PMLR, 2021.

[12] Jean-Yves Audibert and Sébastien Bubeck. Best arm identification in multi-armed bandits. In *COLT-23th Conference on Learning Theory-2010*, pages 13–p, 2010.

[13] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[14] Alex Auvolat, Sarath Chandar, Pascal Vincent, Hugo Larochelle, and Yoshua Bengio. Clustering is efficient for approximate maximum inner product search. *arXiv preprint arXiv:1507.05910*, 2015.

[15] Yoram Bachrach, Yehuda Finkelstein, Ran Gilad-Bachrach, Liran Katzir, Noam Koenigstein, Nir Nice, and Ulrich Paquet. Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 257–264. ACM, 2014.

[16] Grey Ballard, Tamara G Kolda, Ali Pinar, and C Seshadhri. Diamond sampling for approximate maximum all-pairs dot-product (mad) search. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 11–20. IEEE, 2015.

[17] Rémi Bardenet, Odalric-Ambrym Maillard, et al. Concentration inequalities for sampling without roeplacement. *Bernoulli*, 21(3):1361–1385, 2015.

[18] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4: 430–449, 2022.

[19] Amir Beck and Marc Teboulle. Mirror descent and nonlinear projected subgradient methods for convex optimization. *Operations Research Letters*, 31(3):167–175, 2003.

[20] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[21] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

[22] Shahin Boluki, Randy Ardywibowo, Siamak Zamani Dadaneh, Mingyuan Zhou, and Xiaoning Qian. Learnable bernoulli dropout for bayesian deep learning. In *International Conference on Artificial Intelligence and Statistics*, pages 3905–3916. PMLR, 2020.

[23] Antoine Bordes, Seyda Ertekin, Jason Weston, and Léon Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6(Sep):1579–1619, 2005.

[24] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration inequalities: A nonasymptotic theory of independence*. Oxford university press, 2013.

[25] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[26] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In *International conference on Algorithmic learning theory*, pages 23–37. Springer, 2009.

[27] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[28] Olivier Canévet, Cijo Jose, and Francois Fleuret. Importance sampling tree for large-scale empirical expectation. In *International Conference on Machine Learning*, pages 1454–1462, 2016.

[29] Beidi Chen, Yingchen Xu, and Anshumali Shrivastava. Fast and accurate stochastic gradient estimation. In *Advances in Neural Information Processing Systems*, pages 12339–12349, 2019.

[30] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[31] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[32] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[33] Tianyi Chen, Georgios Giannakis, Tao Sun, and Wotao Yin. Lag: Lazily aggregated gradient for communication-efficient distributed learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[34] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[35] Francois Chollet. *Deep learning with python*. Manning Publications Co., 2017.

[36] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 208–214. JMLR Workshop and Conference Proceedings, 2011.

[37] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[38] Edith Cohen and David D Lewis. Approximating matrix multiplication for pattern recognition tasks. *Journal of Algorithms*, 30(2):211–252, 1999.

[39] Harold Connamacher, Nikil Pancha, Rui Liu, and Soumya Ray. Rankboost+: an improvement to rankboost. *Machine Learning*, 109(1):51–78, 2020.

[40] Corinna Cortes, Giulia DeSalvo, and Mehryar Mohri. Boosting with abstention. *Advances in Neural Information Processing Systems*, 29:1660–1668, 2016.

[41] Corinna Cortes, Giulia DeSalvo, Claudio Gentile, Mehryar Mohri, and Scott Yang. Online learning with abstention. In *international conference on machine learning*, pages 1059–1067. PMLR, 2018.

[42] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 39–46. ACM, 2010.

[43] Dominik Csiba, Zheng Qu, and Peter Richtárik. Stochastic dual coordinate ascent with adaptive probabilities. In *International Conference on Machine Learning*, pages 674–683, 2015.

[44] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[45] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[46] Varsha Dani, Sham M Kakade, and Thomas P Hayes. The price of bandit information for online optimization. In *Advances in Neural Information Processing Systems*, pages 345–352, 2008.

[47] John Darzentas. Problem complexity and method efficiency in optimization. *Journal of the Operational Research Society*, 35(5):455–455, 1984.

[48] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 561–574. ACM, 2017.

[49] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[50] Christopher M De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, pages 2674–2682, 2015.

[51] Thomas Dean, Mark A Ruzon, Mark Segal, Jonathon Shlens, Sudheendra Vijayanarasimhan, and Jay Yagnik. Fast, accurate detection of 100,000 object classes on a single machine. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 1814–1821. IEEE, 2013.

[52] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[53] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[54] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. *arXiv preprint arXiv:1905.03197*, 2019.

[55] Timothy Dozat. Incorporating nesterov momentum into adam. *International Conference on Learning Representations, Workshop Track*, 2016.

[56] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[57] Aritra Dutta, El Houcine Bergou, Ahmed M Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3817–3824, 2020.

[58] Rasheed El-Bouri, David Eyre, Peter Watkinson, Tingting Zhu, and David Clifton. Student-teacher curriculum learning via reinforcement learning: predicting hospital inpatient admission location. In *International Conference on Machine Learning*, pages 2848–2857. PMLR, 2020.

[59] Ran El-Yaniv et al. On the foundations of noise-free selective classification. *Journal of Machine Learning Research*, 11(5), 2010.

[60] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. Pac bounds for multi-armed bandit and markov decision processes. In *International Conference on Computational Learning Theory*, pages 255–270. Springer, 2002.

[61] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. *Journal of machine learning research*, 7(Jun):1079–1105, 2006.

[62] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[63] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning*, pages 3061–3071. PMLR, 2020.

[64] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[65] Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. Technical report, 2020.

[66] Justin Fu, Aviral Kumar, Matthew Soh, and Sergey Levine. Diagnosing bottlenecks in deep q-learning algorithms. In *International Conference on Machine Learning*, pages 2021–2030. PMLR, 2019.

[67] Giorgio Fumera and Fabio Roli. Support vector machines with embedded reject option. In *International Workshop on Support Vector Machines*, pages 68–82. Springer, 2002.

[68] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[69] Aditya Gangrade, Anil Kag, and Venkatesh Saligrama. Selective classification via one-sided prediction. In *International Conference on Artificial Intelligence and Statistics*, pages 2179–2187. PMLR, 2021.

[70] Hongchang Gao, Jian Pei, and Heng Huang. Demystifying dropout. In *International Conference on Machine Learning*, pages 2112–2121. PMLR, 2019.

[71] Yonatan Geifman and Ran El-Yaniv. Selective classification for deep neural networks. *arXiv preprint arXiv:1705.08500*, 2017.

[72] Yonatan Geifman and Ran El-Yaniv. Selectivenet: A deep neural network with an integrated reject option. In *International Conference on Machine Learning*, pages 2151–2159. PMLR, 2019.

[73] Zahra Ghodsi, Siva Kumar Sastry Hari, Iuri Frosio, Timothy Tsai, Alejandro Troccoli, Stephen W Keckler, Siddharth Garg, and Anima Anandkumar. Generating and characterizing scenarios for safety testing of autonomous vehicles. In *2021 IEEE Intelligent Vehicles Symposium (IV)*, pages 157–164. IEEE, 2021.

[74] Rohit Girdhar, Joao Carreira, Carl Doersch, and Andrew Zisserman. Video action transformer network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 244–253, 2019.

[75] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

[76] Chen Gong, Jian Yang, and Dacheng Tao. Multi-modal curriculum learning over graphs. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(4):1–25, 2019.

[77] Ian Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

[78] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[79] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. Power-bert: Accelerating bert inference via progressive word-vector elimination. In *International Conference on Machine Learning*, pages 3690–3699. PMLR, 2020.

[80] Sheng Guo, Weilin Huang, Haozhi Zhang, Chenfan Zhuang, Dengke Dong, Matthew R Scott, and Dinglong Huang. Curriculumnet: Weakly supervised learning from large-scale web images. In *Proceedings of the European conference on computer vision (ECCV)*, pages 135–150, 2018.

[81] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W Mahoney. Training recommender systems at scale: Communication-efficient model and data parallelism. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2928–2936, 2021.

[82] Mert Gurbuzbalaban, Asuman Ozdaglar, Pablo A Parrilo, and Nuri Vanli. When cyclic coordinate descent outperforms randomized coordinate descent. *Advances in Neural Information Processing Systems*, 30, 2017.

[83] Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. *arXiv preprint arXiv:1904.03626*, 2019.

[84] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[85] Wenjia He, Michael R Anderson, Maxwell Strome, and Michael Cafarella. A method for optimizing opaque filter queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1257–1272, 2020.

[86] Martin E Hellman. The nearest neighbor classification rule with a reject option. *IEEE Transactions on Systems Science and Cybernetics*, 6(3):179–185, 1970.

[87] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[88] Le Hou, Richard Yuanzhe Pang, Tianyi Zhou, Yuexin Wu, Xinying Song, Xiaodan Song, and Denny Zhou. Token dropping for efficient bert pretraining. *arXiv preprint arXiv:2203.13240*, 2022.

[89] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer. *arXiv preprint arXiv:1809.04281*, 2018.

[90] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

[91] Lang Huang, Chao Zhang, and Hongyang Zhang. Self-adaptive training: beyond empirical risk minimization. *Advances in neural information processing systems*, 33:19365–19376, 2020.

[92] Yu-Siang Huang and Yi-Hsuan Yang. Pop music transformer: Beat-based modeling and generation of expressive pop piano compositions. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1180–1188, 2020.

[93] Martin Jaggi. Revisiting frank-wolfe: Projection-free sparse convex optimization. In *ICML (1)*, pages 427–435, 2013.

[94] Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I Jordan. Communication-efficient distributed dual coordinate ascent. *Advances in neural information processing systems*, 27, 2014.

[95] Prateek Jain and Ashish Kapoor. Active learning for large multi-class problems. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 762–769. IEEE, 2009.

[96] Kevin Jamieson and Robert Nowak. Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In *Information Sciences and Systems (CISS), 2014 48th Annual Conference on*, pages 1–6. IEEE, 2014.

[97] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[98] Kevin Jamieson, Matthew Malloy, Robert Nowak, and Sébastien Bubeck. lil'ucb: An optimal exploration algorithm for multi-armed bandits. In *Conference on Learning Theory*, pages 423–439, 2014.

[99] Ziheng Jiang, Chiyuan Zhang, Kunal Talwar, and Michael C Mozer. Exploring the memorization-generalization continuum in deep learning. 2020.

[100] Thorsten Joachims. Training linear svms in linear time. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–226. ACM, 2006.

[101] Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural svms. *Machine Learning*, 77(1):27–59, 2009.

[102] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[103] Shivaram Kalyanakrishnan, Ambuj Tewari, Peter Auer, and Peter Stone. Pac subset selection in stochastic multi-armed bandits. In *ICML*, volume 12, pages 655–662, 2012.

[104] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246, 2013.

[105] Pavel Karpov, Guillaume Godin, and Igor V Tetko. A transformer model for retrosynthesis. In *International Conference on Artificial Neural Networks*, pages 817–830. Springer, 2019.

[106] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. *arXiv preprint arXiv:1803.00942*, 2018.

[107] Omid Keivani, Kaushik Sinha, and Parikshit Ram. Improved maximum inner product search with better theoretical guarantees. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 2927–2934. IEEE, 2017.

[108] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[109] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. Learned token pruning for transformers. *arXiv preprint arXiv:2107.00910*, 2021.

[110] Young Jin Kim and Hany Hassan Awadalla. Fastformers: Highly efficient transformer models for natural language understanding. *arXiv preprint arXiv:2010.13382*, 2020.

[111] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and efficient moe training for multitask multilingual models. *arXiv preprint arXiv:2109.10465*, 2021.

[112] D Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, volume 5, 2015.

[113] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[114] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. *Advances in neural information processing systems*, 28:2575–2583, 2015.

[115] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[116] Anastasia Koloskova, Tao Lin, Sebastian U Stich, and Martin Jaggi. Decentralized deep learning with arbitrary communication compression. *arXiv preprint arXiv:1907.09356*, 2019.

[117] Benjamin Kompa, Jasper Snoek, and Andrew L Beam. Second opinion needed: communicating uncertainty in medical machine learning. *NPJ Digital Medicine*, 4(1):1–6, 2021.

[118] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.

[119] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[120] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[121] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, volume 333, pages 2267–2273, 2015.

[122] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[123] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Asaga: asynchronous parallel saga. *arXiv preprint arXiv:1606.04809*, 2016.

[124] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Improved asynchronous parallel optimization analysis for stochastic incremental methods. *The Journal of Machine Learning Research*, 19(1):3140–3207, 2018.

[125] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[126] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[127] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. Base layers: Simplifying training of large, sparse models. *arXiv preprint arXiv:2103.16716*, 2021.

[128] Conglong Li, Minjia Zhang, and Yuxiong He. Curriculum learning: A regularization method for efficient and stable billion-scale gpt model pre-training. *arXiv preprint arXiv:2108.06084*, 2021.

[129] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[130] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[131] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[132] Xingguo Li, Tuo Zhao, Raman Arora, Han Liu, and Mingyi Hong. On faster convergence of cyclic block coordinate descent-type methods for strongly convex minimization. *The Journal of Machine Learning Research*, 18(1):6741–6764, 2017.

[133] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[134] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[135] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[136] Tao Lin, Lingjing Kong, Sebastian Stich, and Martin Jaggi. Extrapolation for large-batch training in deep learning. In *International Conference on Machine Learning*, pages 6094–6104. PMLR, 2020.

[137] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[138] Han Liu, Mark Palatucci, and Jian Zhang. Blockwise coordinate descent procedures for the multi-task lasso, with applications to neural semantic basis discovery. In *Proceedings of the 26th annual international conference on machine learning*, pages 649–656, 2009.

[139] Hao Liu, Alexander Trott, Richard Socher, and Caiming Xiong. Competitive experience replay. *arXiv preprint arXiv:1902.00528*, 2019.

[140] Ji Liu, Stephen J Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *The Journal of Machine Learning Research*, 16(1):285–322, 2015.

[141] Rui Liu and Barzan Mozafari. Communication-efficient distributed learning for large batch optimization. In *International Conference on Machine Learning*, pages 13925–13946. PMLR, 2022.

[142] Rui Liu and Barzan Mozafari. Transformer with memory replay. *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7567–7575, 2022.

[143] Rui Liu and Soumya Ray. An analysis of boosted linear classifiers on noisy data with applications to multiple-instance learning. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 287–296. IEEE, 2017.

[144] Rui Liu, Wei Cheng, Hanghang Tong, Wei Wang, and Xiang Zhang. Robust multi-network clustering via joint cross-domain cluster alignment. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 291–300. IEEE, 2015.

[145] Rui Liu, Tianyi Wu, and Barzan Mozafari. A bandit approach to maximum inner product search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4376–4383, 2019.

[146] Rui Liu, Tianyi Wu, and Barzan Mozafari. Adam with bandit sampling for deep learning. *Advances in Neural Information Processing Systems*, 33:5393–5404, 2020.

[147] Rui Liu, Young Jin Kim, Alexandre Muzio, and Hany Hassan. Gating dropout: Communication-efficient regularization for sparsely activated transformers. In *International Conference on Machine Learning*, pages 13782–13792. PMLR, 2022.

[148] Ruishan Liu and James Zou. The effects of memory replay in reinforcement learning. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 478–485. IEEE, 2018.

[149] Yaohua Liu, Wei Xu, Gang Wu, Zhi Tian, and Qing Ling. Communication-censored admm for decentralized consensus optimization. *IEEE Transactions on Signal Processing*, 67(10): 2565–2579, 2019.

[150] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[151] Ziyin Liu, Zhikang Wang, Paul Pu Liang, Russ R Salakhutdinov, Louis-Philippe Morency, and Masahito Ueda. Deep gamblers: Learning to abstain with portfolio theory. *Advances in Neural Information Processing Systems*, 32:10623–10633, 2019.

[152] Francesco Locatello, Rajiv Khanna, Michael Tschannen, and Martin Jaggi. A unified optimization view on generalized matching pursuit and frank-wolfe. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, number EPFL-CONF-229229, 2017.

[153] Jieliang Luo and Hui Li. Dynamic experience replay. In *Conference on Robot Learning*, pages 1191–1200. PMLR, 2020.

[154] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*, 2015.

[155] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

[156] H Brendan McMahan and Avrim Blum. Online geometric optimization in the bandit setting against an adaptive adversary. In *International Conference on Computational Learning Theory*, pages 109–123. Springer, 2004.

[157] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[158] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[159] Pietro Morerio, Jacopo Cavazza, Riccardo Volpi, René Vidal, and Vittorio Murino. Curriculum dropout. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3544–3552, 2017.

[160] Barzan Mozafari and Ning Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 2015.

[161] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[162] Hongseok Namkoong, Aman Sinha, Steve Yadlowsky, and John C Duchi. Adaptive sampling probabilities for non-smooth optimization. In *International Conference on Machine Learning*, pages 2574–2583, 2017.

[163] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*, 2021.

[164] Deanna Needell, Rachel Ward, and Nati Srebro. Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm. In *Advances in Neural Information Processing Systems*, pages 1017–1025, 2014.

[165] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[166] Behnam Neyshabur and Nathan Srebro. On symmetric and asymmetric LSHs for inner product search. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pages 1926–1934. JMLR. org, 2015.

[167] Lam M Nguyen, Phuong Ha Nguyen, Marten van Dijk, Peter Richtárik, Katya Scheinberg, and Martin Takáč. Sgd and hogwild! convergence without the bounded gradients assumption. *arXiv preprint arXiv:1802.03801*, 2018.

[168] Julie Nutini, Mark Schmidt, Issam Laradji, Michael Friedlander, and Hoyt Koepke. Coordinate descent converges faster with the gauss-southwell rule than random selection. In *International Conference on Machine Learning*, pages 1632–1641. PMLR, 2015.

[169] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. Neighbor-sensitive hashing. *PVLDB*, 2015.

[170] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. BlinkML: Approximate machine learning with probabilistic guarantees y. *Technical Report*, `http://web.eecs.umich.edu/~mozafari/php/data/uploads/blinkml_report.pdf`, 2018.

[171] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

[172] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[173] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[174] Emmanouil Antonios Platanios, Otilia Stretcu, Graham Neubig, Barnabas Poczos, and Tom M Mitchell. Competence-based curriculum learning for neural machine translation. *arXiv preprint arXiv:1903.09848*, 2019.

[175] Matt Post. A call for clarity in reporting bleu scores. *arXiv preprint arXiv:1804.08771*, 2018.

[176] Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. Bpe-dropout: Simple and effective subword regularization. *arXiv preprint arXiv:1910.13267*, 2019.

[177] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

[178] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[179] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[180] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021.

[181] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[182] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[183] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations (ICLR)*, 2018.

[184] Cèdric Renggli, Dan Alistarh, Torsten Hoefler, and Mehdi Aghagolzadeh. Sparcml: High-performance sparse communication for machine learning. *arXiv preprint arXiv:1802.08021*, 2018.

[185] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[186] Peter Richtárik and Martin Takáč. Distributed coordinate descent method for learning with big data. *The Journal of Machine Learning Research*, 17(1):2657–2681, 2016.

[187] Peter Richtárik and Martin Takáč. On optimal probabilities in stochastic coordinate descent methods. *Optimization Letters*, 10(6):1233–1243, 2016.

[188] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *arXiv preprint arXiv:2106.05974*, 2021.

[189] Stephen Roller, Sainbayar Sukhbaatar, Arthur Szlam, and Jason Weston. Hash layers for large sparse models. *arXiv preprint arXiv:2106.04426*, 2021.

[190] Peter J Rousseeuw. Least median of squares regression. *Journal of the American statistical association*, 79(388):871–880, 1984.

[191] Cynthia Rudin, Ingrid Daubechies, and Robert E Schapire. The dynamics of adaboost: Cyclic behavior and convergence of margins. *Journal of Machine Learning Research*, 5(Dec): 1557–1595, 2004.

[192] Ankan Saha and Ambuj Tewari. On the finite time convergence of cyclic coordinate descent methods. *arXiv preprint arXiv:1005.2146*, 2010.

[193] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. *arXiv preprint arXiv:1901.06588*, 2019.

[194] Farnood Salehi, Patrick Thiran, and L Elisa Celis. Coordinate descent with bandit sampling. *arXiv preprint arXiv:1712.03010*, 2017.

[195] Farnood Salehi, Patrick Thiran, and L Elisa Celis. Stochastic dual coordinate descent with bandit sampling. *arXiv preprint arXiv:1712.03010*, 2017.

[196] Farnood Salehi, Patrick Thiran, and Elisa Celis. Coordinate descent with bandit sampling. In *Advances in Neural Information Processing Systems*, pages 9267–9277, 2018.

[197] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[198] Robert E Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[199] Robert E Schapire. Explaining adaboost. In *Empirical inference*, pages 37–52. Springer, 2013.

[200] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[201] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[202] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *arXiv preprint arXiv:1907.10597*, 2019.

[203] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong 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.

[204] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118*, 2016.

[205] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research*, 14(Feb):567–599, 2013.

[206] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[207] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[208] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. Training region-based object detectors with online hard example mining. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 761–769, 2016.

[209] Anshumali Shrivastava and Ping Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in Neural Information Processing Systems*, pages 2321–2329, 2014.

[210] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529 (7587):484–489, 2016.

[211] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[212] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[213] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing NLG 530b, A large-scale generative language model. *CoRR*, abs/2201.11990, 2022. URL `https://arxiv.org/abs/2201.11990`.

[214] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. *arXiv preprint arXiv:1905.02450*, 2019.

[215] Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. Curriculum learning: A survey. *International Journal of Computer Vision*, pages 1–40, 2022.

[216] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[217] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. In *Advances in Neural Information Processing Systems*, pages 4447–4458, 2018.

[218] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the Inaernational Speech Communication Association*, 2015.

[219] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[220] Peiquan Sun, Wengang Zhou, and Houqiang Li. Attentive experience replay. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5900–5907, 2020.

[221] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[222] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[223] Hanlin Tang, Shaoduo Gan, Ce Zhang, Tong Zhang, and Ji Liu. Communication compression for decentralized training. In *Advances in Neural Information Processing Systems*, pages 7652–7662, 2018.

[224] Hanlin Tang, Chen Yu, Xiangru Lian, Tong Zhang, and Ji Liu. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pages 6155–6165. PMLR, 2019.

[225] Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In *International Conference on Machine Learning*, pages 10118–10129. PMLR, 2021.

[226] Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pages 9438–9447. PMLR, 2020.

[227] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[228] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[229] Paul Tseng. Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications*, 109(3):475–494, 2001.

[230] Paul Tseng and Sangwoon Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Mathematical Programming*, 117(1):387–423, 2009.

[231] Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. Variance-based gradient compression for efficient distributed deep learning. *arXiv preprint arXiv:1802.06058*, 2018.

[232] Hado van Hasselt, Matteo Hessel, and John Aslanides. When to use parametric models in reinforcement learning? *arXiv preprint arXiv:1906.05243*, 2019.

[233] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[234] Lucas Nunes Vieira, Minako O'Hagan, and Carol O'Sullivan. Understanding the societal impacts of machine translation: a critical review of the literature on medical and legal use cases. *Information, Communication & Society*, 24(11):1515–1532, 2021.

[235] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[236] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *arXiv preprint arXiv:1905.13727*, 2019.

[237] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Practical low-rank communication compression in decentralized deep learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[238] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[239] Hanrui Wang, Zhekai Zhang, and Song 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.

[240] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*, pages 9850–9861, 2018.

[241] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7675–7684, 2018.

[242] Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[243] Yiren Wang, ChengXiang Zhai, and Hany Hassan Awadalla. Multi-task learning for multilingual neural machine translation. *arXiv preprint arXiv:2010.02523*, 2020.

[244] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[245] Jianqiao Wangni, @articletsuzuku2018variance, title=Variance-based gradient compression for efficient distributed deep learning, author=Tsuzuku, Yusuke and Imachi, Hiroto and Akiba, Takuya, journal=arXiv preprint arXiv:1802.06058, year=2018 Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, pages 1299–1309, 2018.

[246] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[247] Guillaume Wenzek, Marie-Anne Lachaux, Alexis Conneau, Vishrav Chaudhary, Francisco Guzmán, Armand Joulin, and Edouard Grave. Ccnet: Extracting high quality monolingual datasets from web crawl data. *arXiv preprint arXiv:1911.00359*, 2019.

[248] Chenshen Wu, Luis Herranz, Xialei Liu, Joost van de Weijer, Bogdan Raducanu, et al. Memory replay gans: Learning to generate new categories without forgetting. *Advances in Neural Information Processing Systems*, 31:5962–5972, 2018.

[249] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*, pages 5325–5333. PMLR, 2018.

[250] Lijun Wu, Juntao Li, Yue Wang, Qi Meng, Tao Qin, Wei Chen, Min Zhang, Tie-Yan Liu, et al. R-drop: regularized dropout for neural networks. *Advances in Neural Information Processing Systems*, 34, 2021.

[251] Xiaoxia Wu, Ethan Dyer, and Behnam Neyshabur. When do curricula work? *arXiv preprint arXiv:2012.03107*, 2020.

[252] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[253] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2020.

[254] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020.

[255] An Yang, Junyang Lin, Rui Men, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Jiamang Wang, Yong Li, et al. Exploring sparse expert models and beyond. *arXiv preprint arXiv:2105.15082*, 2021.

[256] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[257] Zhewei Yao, Xiaoxia Wu, Conglong Li, Connor Holmes, Minjia Zhang, Cheng Li, and Yuxiong He. Random-ltd: Random and layerwise token dropping brings efficient training for large-scale transformers. *arXiv preprint arXiv:2211.11586*, 2022.

[258] Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 6:12, 2017.

[259] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[260] Zhao You, Shulin Feng, Dan Su, and Dong Yu. Speechmoe: Scaling to large acoustic models with dynamic routing mixture of experts. *arXiv preprint arXiv:2105.03036*, 2021.

[261] Hsiang-Fu Yu, Cho-Jui Hsieh, Qi Lei, and Inderjit S Dhillon. A greedy approach for budgeted maximum inner product search. In *Advances in Neural Information Processing Systems*, pages 5459–5468, 2017.

[262] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander Schwing, Murali Annavaram, and Salman Avestimehr. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed cnn training. *arXiv preprint arXiv:1811.03617*, 2018.

[263] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[264] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4035–4043. JMLR. org, 2017.

[265] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv preprint arXiv:1905.11881*, 2019.

[266] Peilin Zhao and Tong Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *international conference on machine learning*, pages 1–9, 2015.

[267] Shuai Zheng, Ziyue Huang, and James T Kwok. Communication-efficient distributed blockwise momentum sgd with error-feedback. *arXiv preprint arXiv:1905.10936*, 2019.

[268] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[269] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 928–936, 2003.

[270] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. Taming sparsely activated transformer with stochastic experts. *arXiv preprint arXiv:2110.04260*, 2021.

Appendix A
Proofs for Chapter 2

We use the following corollary from [17].

Corollary A.1 (Corollary 2.5 in [17]). *Let $\mathcal{X} = (x_1, x_2, \cdots, x_N)$ be a finite set of size $N > 1$ in $[a, b]$ with mean $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$, and $(X_1, \cdots, X_m)$ be a list of size $m < N$ sampled without replacement from $\mathcal{X}$. Then for any $m \leq N$, and any $\delta \in [0, 1]$, it holds*

$$\mathbb{P}\left[\frac{1}{m}\sum_{t=1}^{m} X_t - \mu \leq (b-a)\sqrt{\frac{\rho_m \log(1/\delta)}{2m}}\right] \geq 1 - \delta \tag{A.1}$$

where ρ_m is defined as

$$\rho_m = min\left\{(1 - \frac{m-1}{N}), (1 - \frac{m}{N})(1 + 1/m)\right\} \tag{A.2}$$

Based on this corollary, we have the following lemma.

Lemma A.1. *Let $\mathcal{X} = (x_1, x_2, \cdots, x_N)$ be a finite set of size $N > 1$ in $[a, b]$ with mean $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$, and $(X_1, \cdots, X_m)$ be a list of size $m < N$ sampled without replacement from $\mathcal{X}$. Then, for any given $0 < \epsilon < 1, 0 < \delta < 1$, if*

$$m \geq min\{\frac{u+1}{1+\frac{u}{N}}, \frac{u+\frac{u}{N}}{1+\frac{u}{N}}\} \tag{A.3}$$

where $u = \frac{\log(1/\delta)}{2}\frac{(b-a)^2}{\epsilon^2}$, then we have

$$\mathbb{P}\left[\frac{1}{m}\sum_{t=1}^{m} X_t - \mu \leq \epsilon\right] \geq 1 - \delta \tag{A.4}$$

Proof. From Corollary A.1, it is obvious that for any given $\epsilon > 0, \delta > 0$, as long we have

$$\frac{m}{\rho_m} \geq \frac{\log(1/\delta)}{2}\frac{(b-a)^2}{\epsilon^2} \tag{A.5}$$

then we also have

$$\mathbb{P}\left[\frac{1}{m}\sum_{t=1}^{m} X_t - \mu \leq \epsilon\right] \geq 1 - \delta \tag{A.6}$$

Denote the right hand side of Eq. A.5 as u. Since ρ_m is the minimum of two functions, we can equivalently solve for m by using either of the two function, and then take the minimum of their solutions.

If $\rho = 1 - \frac{m-1}{N}$, we have

$$\begin{aligned}
&\frac{m}{1-\frac{m-1}{N}} \geq u \\
&m \geq u - \frac{m-1}{N}u \\
&(1+\frac{u}{N})m \geq u + \frac{u}{N} \\
&m \geq \frac{u+\frac{u}{N}}{1+\frac{u}{N}}
\end{aligned} \tag{A.7}$$

If $\rho = \frac{m}{(1-\frac{m}{N})(1+\frac{1}{m})}$, we have

$$\begin{aligned}
&\frac{m}{(1-\frac{m}{N})(1+\frac{1}{m})} \geq u \\
&m \geq [(1+\frac{1}{m}) - \frac{m}{N} - \frac{1}{N}]u \\
&m \geq u + \frac{u}{m} - \frac{u}{N}m - \frac{u}{N} \\
&(1+\frac{u}{N})m - \frac{u}{m} \geq u - \frac{u}{N} \\
&(1+\frac{u}{N})m^2 - (u - \frac{u}{N})m - u \geq 0
\end{aligned} \tag{A.8}$$

We need to find an m that satisfies $(1+\frac{u}{N})m^2 - (u-\frac{u}{N})m - u \geq 0$. However, the closed-form solution would be quite complicated. We thus relax it slightly, as follows. We know that for any m that satisfies $(1+\frac{u}{N})m^2-(u-\frac{u}{N})m-u-1 \geq 0$, it must also satisfy $(1+\frac{u}{N})m^2-(u-\frac{u}{N})m-u \geq 0$. Therefore, we relax this problem to $(1+\frac{u}{N})m^2 - (u-\frac{u}{N})m - u - 1 \geq 0$. The solution of the relaxed problem is also a valid solution to the original problem. To solve the relaxed problem,

$$\begin{aligned}
(1+\frac{u}{N})m^2 - (u-\frac{u}{N})m - u - 1 &\geq 0 \\
[(1+\frac{u}{N})m - u - 1][m+1] &\geq 0 \\
m &\geq \frac{u+1}{1+\frac{u}{N}}
\end{aligned} \tag{A.9}$$

Combining (A.7) and (A.9), we have

$$m \geq min\{\frac{u+1}{1+\frac{u}{N}}, \frac{u+\frac{u}{N}}{1+\frac{u}{N}}\} \tag{A.10}$$

We would like the sample size to be as small as possible. Thus, we simply use the equal sign from above inequality. It is easy to see that, for any $u \geq 0$, we have $0 \leq m \leq N$. □

Lemma A.2. *For Algorithm 2.1, at any iteration l, we have*

$$\mathbb{P}\left[\tilde{p}_{S_l} \leq \tilde{p}_{S_{l+1}} + \epsilon_l\right] \geq 1 - \delta_l \tag{A.11}$$

Proof. Let us denote z_l be the arm with K-th highest true mean of reward in S_l. Let $\tilde{S}_l^\epsilon = \{a \in S_l : p_{z_l} - p_a \leq \epsilon\}$. At the end of each round in Algorithm 1, we remove $\left\lceil \frac{|S_l|-K}{2} \right\rceil$ arms with the least empirical means. In other words, $K + \left\lfloor \frac{|S_l|-K}{2} \right\rfloor$ arms with the highest empirical means will remain in S_{l+1}. When there are at least K arms from $\tilde{S}_l^\epsilon$ that end up in S_{l+1}, the K-th highest true mean of the arms in S_{l+1} will be within ϵ_l of the K-th highest true mean of the arms in S_l. Hence, the event $\tilde{p}_{S_l} \leq \tilde{p}_{S_{l+1}} + \epsilon_l$ fails when there are strictly more than $K + \left\lfloor \frac{|S_l|-K}{2} \right\rfloor - K = \left\lfloor \frac{|S_l|-K}{2} \right\rfloor$ arms in $S_l \setminus \tilde{S}_l^\epsilon$ whose empirical means are greater than the empirical mean of every arm in $\tilde{S}_l^\epsilon$.

Let us compute the probability that an arm from $S_l \setminus \tilde{S}_l^\epsilon$ is better than every arm from $\tilde{S}_l^\epsilon$ in terms of the empirical mean. For any $j \in S_l \setminus \tilde{S}_l^\epsilon$, we have

$$\begin{aligned}
&\mathbb{P}\left[\hat{p}_j \geq \hat{p}_i \ \ \forall i \in \tilde{S}_l^{\epsilon_l}\right] \\
\leq &\mathbb{P}\left[\hat{p}_j \geq \hat{p}_{z_l}\right] \\
\leq &\mathbb{P}\left[\hat{p}_j \geq p_j + \epsilon_l/2 \text{ or } \hat{p}_{z_l} \leq p_{z_l} - \epsilon_l/2\right] \\
\leq &\mathbb{P}\left[\hat{p}_j \geq p_j + \epsilon_l/2\right] + \mathbb{P}\left[\hat{p}_{z_l} \leq p_{z_l} - \epsilon_l/2\right]
\end{aligned} \tag{A.12}$$

According to Lemma A.1, we know that after pulling each arm for $m\left(\frac{2}{\epsilon_l^2}\log\left(\frac{2(|S_l|-K)}{\delta_l\left(\left\lfloor\frac{|S_l|-K}{2}\right\rfloor+1\right)}\right)\right)$ times, the above probability will be bounded by $\frac{\left\lfloor\frac{|S_l|-K}{2}\right\rfloor+1}{|S_l|-K}\delta_l$. Let n_{bad} be the number of arms in $S_l \setminus \tilde{S}_l^\epsilon$ that are better than every arm in $\tilde{S}_l^\epsilon$ in terms of their empirical mean of rewards. Clearly,

$$\begin{aligned}
\mathbb{E}[n_{bad}] &\leq (|S_l| - K)\frac{\left\lfloor\frac{|S_l|-K}{2}\right\rfloor + 1}{|S_l| - K}\delta_l \\
&\leq \left(\left\lfloor\frac{|S_l| - K}{2}\right\rfloor + 1\right)\delta_l
\end{aligned} \tag{A.13}$$

Using Markov inequality, we can bound the probability of failure as

$$\begin{aligned}
&\mathbb{P}\left[n_{bad} \geq \left\lfloor\frac{|S_l| - K}{2}\right\rfloor + 1\right] \\
\leq &\frac{\mathbb{E}[n_{bad}]}{\left\lfloor\frac{|S_l|-K}{2}\right\rfloor + 1} \\
\leq &\delta_l
\end{aligned} \tag{A.14}$$

□

We can now prove the main property of Algorithm 2.1.

Theorem A.1. *The* BOUNDEDME *algorithm (Algorithm* 2.1*) is guaranteed to return ϵ-optimal solution with probability at least $1-\delta$.*

Proof. It follows immediately from Lemma A.2 since

$$\sum_{i=1}^{\log n}\delta_i=\sum_{i=1}^{\log n}\frac{\delta}{2^i}\leq\sum_{i=1}^{\infty}\frac{\delta}{2^i}=\delta$$

and

$$\sum_{i=1}^{\log n}\epsilon_i=\sum_{i=1}^{\log n}\frac{\epsilon}{4}\left(\frac{3}{4}\right)^{i-1}\leq\sum_{i=1}^{\infty}\frac{\epsilon}{4}\left(\frac{3}{4}\right)^{i-1}=\epsilon$$

□

Corollary A.2. *For each arm, the number of times it is pulled by Algorithm* 2.1 *is upper-bounded by N.*

Proof. Based on the definition of function $m(u)$, it is not hard to see that $m(u)\leq N$ if $u\geq 0$. Moreover, at round l of Algorithm 2.1, the accumulated number of times that an arm in S_l has been pulled is

$$t_l=m\left(\frac{2}{\epsilon_l^2}\log\left(\frac{2(|S_l|-K)}{\delta_l\left(\left\lfloor\frac{|S_l|-K}{2}\right\rfloor+1\right)}\right)\right) \tag{A.15}$$

Clearly, $t_l\leq N$. Thus, the total number of times any arm is pulled is upper-bounded by N. □

Corollary A.3. *The time complexity of Algorithm* 2.1 *is $O\left(\frac{n\sqrt{N}}{\epsilon}\sqrt{\log\left(\frac{1}{\delta}\right)}\right)$.*

Proof. At round l, we know that $\epsilon_l=\frac{\epsilon}{4}\left(\frac{3}{4}\right)^{l-1}$ and $\delta_l=\frac{\delta}{2^l}$. The number of arms that will be removed at the end of the l-th iteration is no greater than $\frac{n}{2^l}$. It is easy to see that for the arms that are removed at the end of the l-th iteration, the total number of pulls for any of them will be t_l. Further, we know that $m(u)\leq O(\sqrt{Nu})$ for $u>0$. Also, note that K is typically a small constant, e.g., 1, 5, or 10. Therefore, the sample complexity is

$$\begin{aligned}
&\sum_{l=1}^{\log n}n_l t_l\\
\leq&\sum_{l=1}^{\log n}\frac{n}{2^l}m\left(\frac{2}{\epsilon_l^2}\log\left(\frac{2(|S_l|-K)}{\delta_l\left(\left\lfloor\frac{|S_l|-K}{2}\right\rfloor+1\right)}\right)\right)\\
\leq&\sum_{l=1}^{\log n}\frac{n\sqrt{N}}{2^l}\frac{\sqrt{2}}{\epsilon_l}\sqrt{\log\left(\frac{2(|S_l|-K)}{\delta_l\left(\left\lfloor\frac{|S_l|-K}{2}\right\rfloor+1\right)}\right)}\\
\leq&O\left(\frac{n\sqrt{N}}{\epsilon}\sqrt{\log\left(\frac{1}{\delta}\right)}\right)
\end{aligned} \tag{A.16}$$

□

Appendix B
Proofs for Chapter 3

B.1 Proof of Theorem 3.1

For ease of presentation, we define $G_{t,d} = \sum_{m=1}^{M} g_{t,d}^m$, and $G_{t,[d]} = \sum_{m=1}^{M} g_{t,[d]}^m$. We also define $G_t = \sum_{d=1}^{D} G_{t,[d]}$. According to the definition of S_d-smoothness of f with respect to x_d and the update rule $x_{t+1,d} = x_{t,d} - \alpha_x G_{t,d}$, we have

$$\begin{aligned} f(x_{t+1}) - f(x_t) &\leq \sum_{d=1}^{D} \langle f_d'(x_t), x_{t+1,d} - x_{t,d} \rangle + \sum_{d=1}^{D} \frac{S_d}{2} \|x_{t+1,d} - x_{t,d}\|^2 \\ &\leq \sum_{d=1}^{D} \langle f_d'(x_t), x_{t+1,d} - x_{t,d} \rangle + \frac{S_\infty}{2} \|x_{t+1} - x_t\|^2 \\ &= -\alpha_x \sum_{d=1}^{D} \langle f_d'(x_t), G_{t,d} \rangle + \frac{S_\infty \alpha_x^2}{2} \|G_t\|^2. \end{aligned} \tag{B.1}$$

Thus, taking expectation on both sides and using the unbiasedness of the gradient, i.e., $\mathbb{E}G_{t,d} = f_d'(x_t)$, we have

$$\mathbb{E}\left[f(x_{t+1})\right] - \mathbb{E}\left[f(x_t)\right] \leq -\alpha_x \sum_{d=1}^{D} \|f_d'(x_t)\|^2 + \frac{S_\infty \alpha_x^2}{2} \mathbb{E}\|G_t\|^2. \tag{B.2}$$

Rearranging the order, we have

$$\sum_{d=1}^{D} \|f_d'(x_t)\|^2 \leq \frac{\mathbb{E}\left[f(x_t)\right] - \mathbb{E}\left[f(x_{t+1})\right]}{\alpha_x} + \frac{S_\infty \alpha_x}{2} \mathbb{E}\|G_t\|^2. \tag{B.3}$$

Then,

$$\begin{aligned}
&\sum_{t=1}^{T}\sum_{d=1}^{D}\|f'(x_t)\|^2 \\
\leq&\frac{\mathbb{E}\left[f(x_1)\right]-\mathbb{E}\left[f(x_{T+1})\right]}{\alpha_x}+\frac{S_\infty\alpha_x}{2}\sum_{t=1}^{T}\mathbb{E}\|G_t\|^2 \\
\leq&\frac{\mathbb{E}\left[f(x_1)\right]-\mathbb{E}\left[f(x_{T+1})\right]}{\alpha_x}+\frac{S_\infty\alpha_x}{2M^2}\sum_{m=1}^{M}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|g^m_{t,[d]}\|^2}{p^m_{t,d}} \\
\leq&\frac{R^2}{\alpha_x}+\frac{S_\infty\alpha_x}{2M^2}\sum_{m=1}^{M}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|g^m_{t,[d]}\|^2}{p^m_{t,d}}.
\end{aligned} \tag{B.4}$$

B.2 Proof of Theorem 3.2

As for Algorithm 3.1, it corresponds to mirror descent update with $\psi(p)=\sum_{d=1}^{D}p_d\log p_d$ and $\psi^*(u)=\sum_{d=1}^{D}\exp(u_d-1)$. Denote $\hat{h}^m_{t,d}=\frac{\tilde{l}^m_{t,d}Z^m_{t,d}}{p^m_{t,d}}$. Based on Theorem 5.3 from [27], we have

$$\begin{aligned}
&\bar{R}^m_T \\
=&\mathbb{E}\sum_{t=1}^{T}L^m_t(p^m_t)-\min_{p}\mathbb{E}\sum_{t=1}^{T}L^m_t(p) \\
\leq&\frac{B_\psi(p^m_*,p^m_1)}{\alpha_p}+\frac{1}{\alpha_p}\sum_{t=1}^{T}\mathbb{E}\left[B_{\psi^*}(\nabla\psi(p^m_t)-\alpha_p\hat{h}^m_{t,d},\nabla\psi(p^m_t))\right].
\end{aligned} \tag{B.5}$$

Furthermore, we have

$$
\begin{aligned}
&\bar{R}_T^m \\
=&\mathbb{E}\sum_{t=1}^{T} L_t^m(p_t^m) - \min_p \mathbb{E}\sum_{t=1}^{T} L_t^m(p) \\
\leq&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{1}{\alpha_p}\sum_{t=1}^{T}\mathbb{E}\left[B_{\psi^*}(\nabla\psi(p_t^m) - \alpha_p\hat{h}_{t,d}^m, \nabla\psi(p_t^m))\right] \\
=&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{1}{\alpha_p}\sum_{t=1}^{T}\mathbb{E}\left[\sum_{d=1}^{D} p_{t,d}^m(\exp(-\alpha_p\hat{h}_{t,d}^m) + \alpha_p\hat{h}_{t,d}^m - 1)\right] \\
&\text{(due to inequality } e^z - z - 1 \leq z^2 \text{ for } z \leq 0) \\
\leq&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\mathbb{E}\left[\sum_{d=1}^{D} p_{t,d}^m(\hat{h}_{t,d}^m)^2\right] \\
=&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\mathbb{E}\left[\sum_{d=1}^{D} p_{t,d}^m\frac{(\tilde{l}_{t,d}^m)^2(Z_{t,d}^m)^2}{(p_{t,d}^m)^2}\right] \\
=&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{d=1}^{D} p_{t,d}^m\frac{(\tilde{l}_{t,d}^m)^2\mathbb{E}\left[(Z_{t,d}^m)^2\right]}{(p_{t,d}^m)^2} \\
=&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{d=1}^{D} p_{t,d}^m\frac{(\tilde{l}_{t,d}^m)^2 p_{t,d}^m}{(p_{t,d}^m)^2} \\
=&\frac{B_\psi(p_*^m, p_1^m)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{d=1}^{D}(\tilde{l}_{t,d}^m)^2.
\end{aligned}
\tag{B.6}
$$

Since $\tilde{l}_{t,d}^m = -\frac{\|G_{t,[d]}^m\|^2}{(p_{t,d}^m)^2} + \frac{L^2}{p_{min}^2}$, we have $\tilde{l}_{t,d}^m \leq \frac{L^2}{p_{min}^2}$. Similar to Assumption 3.1, we also assume that $B_\psi(p_*^m, p_1^m) \leq R^2$. Thus, with choosing $\alpha_p = \sqrt{\frac{2R^2p_{min}^4}{DTL^4}}$, we could get

$$
\begin{aligned}
\bar{R}_T^m \leq&\frac{R^2}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{L^4}{p_{min}^4} \\
=&\frac{R^2}{\alpha_p} + \frac{\alpha_p}{2}\frac{TDL^4}{p_{min}^4} \\
=&\frac{RL^2}{p_{min}^2}\sqrt{2DT}.
\end{aligned}
\tag{B.7}
$$

B.3 Proof of Theorem 3.3

Let $\bar{R}^* = max_m\, min_p\, \mathbb{E}\left[\sum_{t=1}^{T} L_t^m(p)\right]$. Assume $\sum_{d=1}^{D} p_{t,d}^m = s, \forall m, t$, where s is the sparsity rate. Plugging Theorem 3.2 into Theorem 3.1, and choosing $\alpha_x = \frac{R}{DL}\sqrt{\frac{2Ms}{S_\infty T}}$, we have

$$\begin{aligned}
&\sum_{t=1}^{T}\sum_{d=1}^{D} \|f'(x_t)\|^2 \\
\leq &\frac{R^2}{\alpha_x} + \frac{S_\infty \alpha_x}{2M}\left(\frac{RL^2}{p_{min}^2}\sqrt{2DT} + \bar{R}^*\right) \\
= &RDL\sqrt{\frac{S_\infty T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{S_\infty s}{2MT}}\bar{R}^* + \frac{R^2 L}{p_{min}^2}\sqrt{\frac{S_\infty s}{MD}}.
\end{aligned} \tag{B.8}$$

B.4 Proof of Lemma 3.1

Let us now consider each iteration at each machine. First, let us prove the following inequality

$$\min_{p\in\mathcal{P}} \sum_{j=1}^{D} \frac{L^2}{p_j} \leq O(\frac{L^2 D^2}{s}). \tag{B.9}$$

To prove the above inequality, the Lagrangian of the left hand side is

$$\mathcal{L}(p, \eta, \theta) = \sum_{d=1}^{D} \frac{L^2}{p_d} - \eta(p^T \mathbb{1} - s) - \theta^T(p - p_{min}\mathbb{1}). \tag{B.10}$$

Setting its first order derivative with respect to p_j to 0, then we have

$$p_j = \frac{L}{\sqrt{|\eta + \theta_j|}}. \tag{B.11}$$

According to complementary slackness, we have $\theta_j(p_j - p_{min}) = 0$. Bcause p_{min} is small, it implies that

$$p_j = \frac{L}{\sqrt{|\eta|}}. \tag{B.12}$$

Plugging p_j's into the equality constraint, we get

$$\sum_{j=1}^{D} p_j = \frac{DL}{\sqrt{|\eta|}} = s. \tag{B.13}$$

It implies that

$$\sqrt{|\eta|} = \frac{DL}{s}. \tag{B.14}$$

Therefore, we have

$$\min_{p\in\mathcal{P}} \sum_{j=1}^{D} \frac{L^2}{p_j} = O(\frac{L^2D^2}{s}). \tag{B.15}$$

Plugging Equation B.15 into $\bar{R}^*$, we have

$$\bar{R}^* = \max_{m} \min_{p\in\mathcal{P}} \mathbb{E}\left[\sum_{t=1}^{T}\sum_{d=1}^{D} \frac{\|g^m_{t,[d]}\|^2}{p_d}\right] \leq O(\frac{TL^2D^2}{s}). \tag{B.16}$$

B.5 Proof of Theorem 3.4

Plugging Lemma 3.1 into the Theorem 3.3, we have

$$\begin{aligned}
&\sum_{t=1}^{T}\sum_{d=1}^{D} \|f'_d(x_t)\|^2 \\
\leq& RDL\sqrt{\frac{S_\infty T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{S_\infty s}{2MT}}\bar{R}^* + \frac{R^2L}{p^2_{min}}\sqrt{\frac{S_\infty s}{MD}} \\
\leq& RDL\sqrt{\frac{S_\infty T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{S_\infty s}{2MT}}\frac{TL^2D^2}{s} + \frac{R^2L}{p^2_{min}}\sqrt{\frac{S_\infty s}{MD}} \\
=& RDL\sqrt{\frac{S_\infty T}{2Ms}} + RDL\sqrt{\frac{S_\infty T}{2Ms}} + \frac{R^2L}{p^2_{min}}\sqrt{\frac{S_\infty s}{MD}} \\
\leq& O\left(RLD\sqrt{\frac{2S_\infty T}{Ms}}\right).
\end{aligned} \tag{B.17}$$

Therefore, we have $\frac{1}{T}\sum_{d=1}^{D}\sum_{t=1}^{T}\|f'_d(x_t)\|^2 \leq O\left(RLD\sqrt{\frac{2S_\infty}{MsT}}\right)$. It implies that the average convergence rate is still $O(\frac{1}{\sqrt{T}})$.

B.6 Analysis for Convex Problems for JOINTSPAR

We discuss convergence guarantee for convex problems in this section, which gives us a tighter convergence bound for JOINTSPAR. We make the following typical assumptions for convex problems [19, 47]:

Assumption B.2. (1) For all $x, y \in \mathcal{X}$, we have $B(x, y) \leq R^2$, where $B(x, y)$ is the Bregman divergence between x and y; (2) $L \geq \|g^m_{t,[d]}\|, \forall t, d, m$.

Theorem B.2. *Under Assumption B.2,* ORACLESPAR *achieves*

$$\begin{aligned}&\mathbb{E}\left[\sum_{t=1}^{T} f(x_t) - f(x^*)\right]\\ \leq&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{m=1}^{M}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|G_{t,[d]}^m\|^2}{p_{t,d}^m}.\end{aligned} \tag{B.18}$$

Proof. This can be proven from the perspective of stochastic mirror descent [19, 47], which is a generalization of stochastic gradient descent. The specific form of stochastic mirror descent depends on the choice of Bregman Divergence. Based on Proposition 1 from [162] or Theorem 4.1 from [19], we have

$$\begin{aligned}&\mathbb{E}\left[\sum_{t=1}^{T} f(x_t) - f(x^*)\right]\\ \leq&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2}\sum_{t=1}^{T}\mathbb{E}\left[\|G_t\|^2\right].\end{aligned} \tag{B.19}$$

According to the update rule, we know $G_t = \frac{1}{M}\sum_{m=1}^{M} Q(G_t^m)$. Therefore, we have

$$\begin{aligned}&\mathbb{E}\left[\sum_{t=1}^{T} f(x_t) - f(x^*)\right]\\ =&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2}\sum_{t=1}^{T}\mathbb{E}\left[\|\frac{1}{M}\sum_{m=1}^{M} Q(G_t^m)\|^2\right]\\ \leq&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{t=1}^{T}\sum_{m=1}^{M}\mathbb{E}\left[\|Q(G_t^m)\|^2\right]\\ =&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{t=1}^{T}\sum_{m=1}^{M}\mathbb{E}\left[\left\|\sum_{d=1}^{D}\frac{G_{t,[d]}^m}{p_{t,d}^m}Z_{t,d}^m\right\|^2\right]\\ \leq&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{t=1}^{T}\sum_{m=1}^{M}\sum_{d=1}^{D}\frac{\|G_{t,[d]}^m\|^2}{(p_{t,d}^m)^2}\mathbb{E}\left[(Z_{t,d}^m)^2\right]\\ =&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{t=1}^{T}\sum_{m=1}^{M}\sum_{d=1}^{D}\frac{\|G_{t,[d]}^m\|^2}{p_{t,d}^m}\\ =&\frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{m=1}^{M}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|G_{t,[d]}^m\|^2}{p_{t,d}^m}.\end{aligned} \tag{B.20}$$

□

Theorem B.3. *Under Assumption B.2,* JOINTSPAR *achieves*

$$\begin{aligned}&\mathbb{E}\left[\sum_{t=1}^{T} f(x_t) - f(x^*)\right]\\ \leq& RDL\sqrt{\frac{T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{s}{2MT}}\bar{R}^* + \frac{R^2L}{p_{min}^2}\sqrt{\frac{s}{MD}}\end{aligned} \tag{B.21}$$

where

$$\bar{R}^* = \max_{m} \min_{p\in\mathcal{P}} \mathbb{E}\left[\sum_{t=1}^{T}\sum_{d=1}^{D} \frac{\|g_{t,[d]}^m\|^2}{p_d}\right]. \tag{B.22}$$

Proof. Plugging the original Theorem 3.2 into the above Theorem B.2, and choosing $\alpha_x = \frac{R}{DL}\sqrt{\frac{2Ms}{T}}$, we have

$$\begin{aligned}&\mathbb{E}\left[\sum_{t=1}^{T} f(x^t) - f(x^*)\right]\\ \leq& \frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{m=1}^{M}\mathbb{E}\left[\sum_{t=1}^{T} L_t^m(p_t^m)\right]\\ \leq& \frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M^2}\sum_{m=1}^{M}\left(\frac{RL^2}{p_{min}^2}\sqrt{2DT} + \bar{R}^*\right)\\ =& \frac{R^2}{\alpha_x} + \frac{\alpha_x}{2M}\left(\frac{RL^2}{p_{min}^2}\sqrt{2DT} + \bar{R}^*\right)\\ =& RLD\sqrt{\frac{T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{s}{2MT}}\bar{R}^* + \frac{R^2L}{p_{min^2}}\sqrt{\frac{s}{MD}}.\end{aligned} \tag{B.23}$$

□

Theorem B.4. *Under Assumption B.2, our distributed learning method* JOINTSPAR *has the following convergence rate*

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} f(x_t) - f(x^*)\right] \leq O\left(RLD\sqrt{\frac{2S}{MsT}}\right) \tag{B.24}$$

where f is the non-convex and S-smooth objective function, and s is the sparsity budget.

Proof. Plugging the original Lemma 3.1 into the Theorem B.3, we have

$$\begin{aligned}
&\mathbb{E}\left[\sum_{t=1}^{T} f(x_t) - f(x^*)\right] \\
=&RLD\sqrt{\frac{T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{s}{2MT}}\bar{R}^* + \frac{R^2L}{p_{min^2}}\sqrt{\frac{s}{MD}} \\
\leq& O\left(RLD\sqrt{\frac{T}{2Ms}} + \frac{R}{LD}\sqrt{\frac{s}{2MT}}T\frac{L^2D^2}{s}\right) \\
=&O\left(RLD\sqrt{\frac{2T}{Ms}}\right).
\end{aligned} \tag{B.25}$$

Therefore, we have $\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} f(x_t) - f(x^*)\right] \leq O\left(RLD\sqrt{\frac{2}{MsT}}\right)$. It implies that the average convergence rate is still $O(\frac{1}{\sqrt{T}})$. □

B.7 Proof of Theorem 3.5

Similar to the proof for Theorem 2 from [259], setting $\alpha_x = \sqrt{\frac{2(f(x_1)-f(x^*))}{\alpha_u^2\|S\|_1 T}}$, we could obtain

$$\begin{aligned}
&\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[\frac{1}{\sqrt{D}}\sum_{d=1}^{D}\|f_d'(x_t)\|\right] \\
\leq&\frac{f(x_1)-f(x^*)}{T\alpha_l\alpha_x} + \frac{\alpha_x\alpha_u^2}{2\alpha_l}\|S\|_1 + \frac{2\alpha_u}{\alpha_l}\frac{1}{\sqrt{D}T}\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\left[\|\Delta_{t,d}\|\right] \\
&\text{(due to } \alpha_x = \sqrt{\frac{2(f(x_1)-f(x^*))}{\alpha_u^2\|S\|_1 T}}\text{)} \\
=&\frac{\alpha_u}{\alpha_l}\sqrt{\frac{2(f(x_1)-f(x^*))\|S\|_1}{TD}} + \frac{2\alpha_u}{\alpha_l}\frac{1}{\sqrt{D}T}\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\left[\|\Delta_{t,d}\|\right]
\end{aligned} \tag{B.26}$$

where $\Delta_{t,d} = G_{t,d} - f_d'(x_t)$ and $G_{t,d} = \frac{1}{MB}\sum_{m=1}^{M}\sum_{b=1}^{B}\frac{g_{t,[d]}^{m,b}}{p_{t,d}^m}Z_{t,d}^m$. Note that b is the index for individual example in a batch, and B is the batch size.

Applying square to both sides, we have

$$\begin{aligned}
&\left(\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[\frac{1}{\sqrt{D}}\sum_{d=1}^{D}\|f_d'(x_t)\|\right]\right)^2 \\
\leq&\frac{\alpha_u}{\alpha_l}\frac{2(f(x_1)-f(x^*))\|S\|_1}{TD}+\frac{4\alpha_u^2}{\alpha_l^2}\frac{1}{DT^2}\left(\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\|\Delta_{t,d}\|\right)^2 \\
\leq&\frac{\alpha_u}{\alpha_l}\frac{2(f(x_1)-f(x^*))\|S\|_1}{TD}+\frac{4\alpha_u^2}{\alpha_l^2}\frac{1}{T}\sum_{t=1}^{T}\sum_{d=1}^{D}\left(\mathbb{E}\|\Delta_{t,d}\|\right)^2 \\
&\text{(due to Jensen's inequality)} \\
\leq&\frac{\alpha_u}{\alpha_l}\frac{2(f(x_1)-f(x^*))\|S\|_1}{TD}+\frac{4\alpha_u^2}{\alpha_l^2}\frac{1}{T}\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\|\Delta_{t,d}\|^2
\end{aligned} \tag{B.27}$$

Next, we will obtain an upper bound on $\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\|\Delta_{t,d}\|^2$. Since $\mathbb{E}G_{t,d}=f_d'(x_t)$, we have

$$\begin{aligned}
\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\|\Delta_{t,d}\|^2 &= \sum_{t=1}^{T}\sum_{d=1}^{D}\frac{1}{M^2B^2}\sum_{m=1}^{M}\sum_{b=1}^{B}\mathbb{E}\|\frac{g_{t,[d]}^{m,b}}{p_{t,d}^m}Z_{t,d}^m-f_d'(x_t)\|^2 \\
&= \sum_{t=1}^{T}\sum_{d=1}^{D}\frac{1}{M^2B^2}\sum_{m=1}^{M}\sum_{b=1}^{B}\left(\mathbb{E}\|\frac{g_{t,[d]}^{m,b}}{p_{t,d}^m}Z_{t,d}^m\|^2-\|f_d'(x_t)\|^2\right) \\
&\leq \sum_{t=1}^{T}\sum_{d=1}^{D}\frac{1}{M^2B^2}\sum_{m=1}^{M}\sum_{b=1}^{B}\mathbb{E}\|\frac{g_{t,[d]}^{m,b}}{p_{t,d}^m}Z_{t,d}^m\|^2 \\
&= \frac{1}{M^2B^2}\sum_{m=1}^{M}\sum_{b=1}^{B}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|g_{t,[d]}^{m,b}\|^2}{p_{t,d}^m}
\end{aligned} \tag{B.28}$$

From Theorem 3.2 and Lemma 3.1, we could get

$$\begin{aligned}
&\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|g_{t,[d]}^{m,b}\|^2}{p_{t,d}^m} \\
\leq&\frac{RL^2}{p_{min}^2}\sqrt{2DT}+\frac{TL^2D^2}{s}
\end{aligned} \tag{B.29}$$

Therefore, we have

$$
\begin{aligned}
&\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\|\Delta_{t,d}\|^2 \\
\leq&\frac{1}{M^2B^2}\sum_{m=1}^{M}\sum_{b=1}^{B}\sum_{t=1}^{T}\sum_{d=1}^{D}\frac{\|g_{t,[d]}^{m,b}\|^2}{p_{t,d}^m} \\
\leq&\frac{1}{M^2B^2}\sum_{m=1}^{M}\sum_{b=1}^{B}\left(\frac{RL^2}{p_{min}^2}\sqrt{2DT}+\frac{TL^2D^2}{s}\right) \\
\leq&\frac{1}{MB}\left(\frac{RL^2}{p_{min}^2}\sqrt{2DT}+\frac{TL^2D^2}{s}\right) \\
&\text{(due to the large batch assumption that } B=T) \\
=&\frac{1}{MT}\left(\frac{RL^2}{p_{min}^2}\sqrt{2DT}+\frac{TL^2D^2}{s}\right) \\
=&\frac{1}{M}\left(\frac{RL^2}{p_{min}^2}\sqrt{\frac{2D}{T}}+\frac{L^2D^2}{s}\right)
\end{aligned}
\tag{B.30}
$$

Plugging the above bound to Inequality B.27, we finally have

$$
\begin{aligned}
&\left(\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[\frac{1}{\sqrt{D}}\sum_{d=1}^{D}\|f_d'(x_t)\|\right]\right)^2 \\
\leq&\frac{\alpha_u}{\alpha_l}\frac{2(f(x_1)-f(x^*))\|S\|_1}{TD}+\frac{4\alpha_u^2}{\alpha_l^2}\frac{1}{T}\sum_{t=1}^{T}\sum_{d=1}^{D}\mathbb{E}\|\Delta_{t,d}\|^2 \\
\leq&\frac{\alpha_u}{\alpha_l}\frac{2(f(x_1)-f(x^*))\|S\|_1}{TD}+\frac{4\alpha_u^2}{\alpha_l^2}\frac{1}{T}\frac{1}{M}\left(\frac{RL^2}{p_{min}^2}\sqrt{\frac{2D}{T}}+\frac{L^2D^2}{s}\right) \\
=&O\left(\frac{(f(x_1)-f(x^*))\|S\|_1}{TD}+\frac{L^2D^2}{MsT}\right)
\end{aligned}
\tag{B.31}
$$

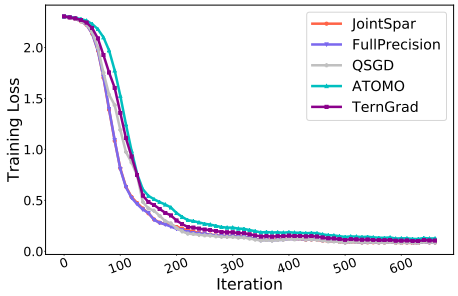
B.8 More Details on Experimental Setup

We train the model on each dataset for 90 epochs with the first 5 epochs as the warmup stage as suggested in [78]. For the learning rate schedule, we set the initial learning rate as 0.1, and shrink the learning rate by 0.1 at epoch $30, 50, 70$.

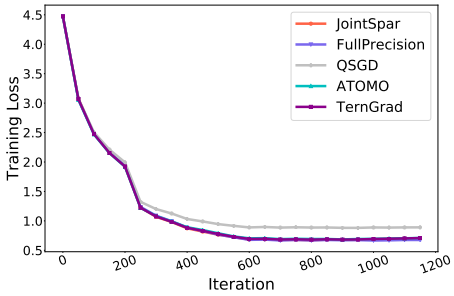
B.9 Experiments on SGD-based JOINTSPAR

We include experimental results for JOINTSPAR which uses SGD as the base optimizer (its pseudocode in Algorithm 3.2). The convergence rates in terms of the number of iterations are shown in Figure B.1, where x-axis is the number of iterations, and y-axis is the training loss. We can see

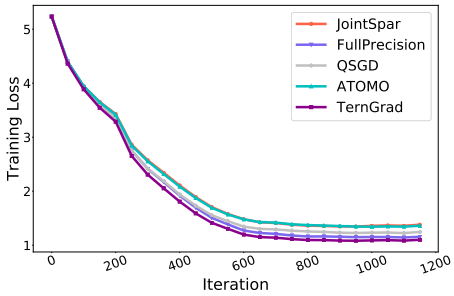
that JOINTSPAR has the same convergece rate as with baselines, which verifies Theorem 3.4 in the main paper. The convergence rates in terms of the wall-clock time are shown in Figure B.2, where x-axis is the wall-clock, and y-axis is the testing accuracy. We can see that JOINTSPAR in general converges faster than baselines.



(a) MNIST



(b) CIFAR10

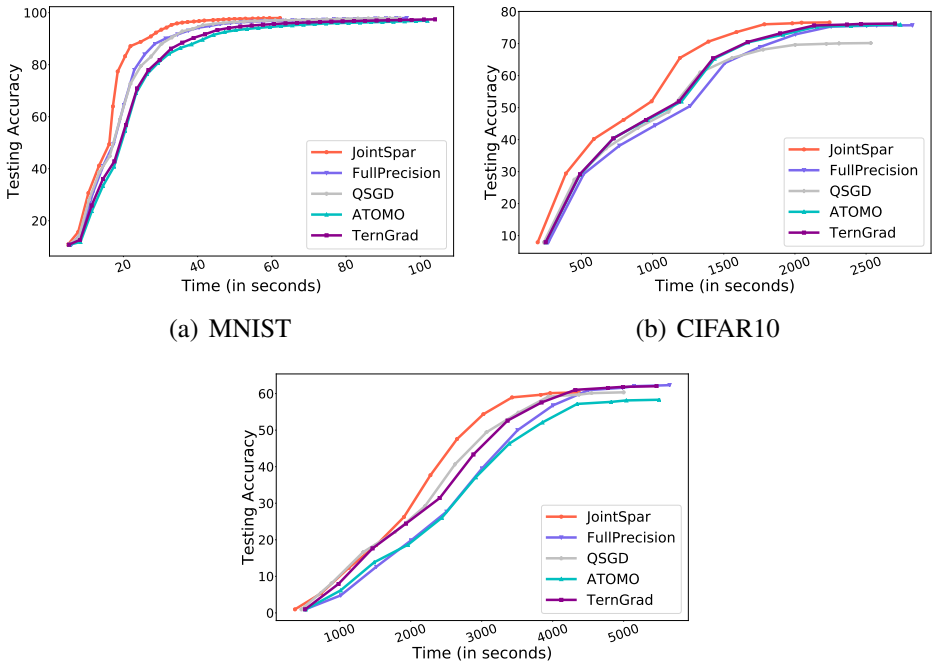


(c) CIFAR100

Figure B.1: Convergence in terms of the number of iterations, confirming that our method JOINTSPAR enjoys a similar convergence rate as most baselines.

B.10 Gradient Computation Time per Epoch

By joint sparsification of the gradient computation and communication, JOINTSPAR aims to reduce both the time spent on computing the gradient and the time spent on communicating it, whereas state-of-the-art communication-efficient methods focus only on the latter. We thus expect JOINTSPAR to spend less time on gradient computation per iteration. However, the total time per iteration can vary based on the method and the number of processes used. Therefore, to ensure a fair comparison, we compare the ratio of the computation time to the total time per iteration, as we increase the number of processes. By examining this ratio, we are accounting for the different total time per iteration. Specifically, we measure the total time per epoch, the communication time per epoch, and their ratio. Here, we still use the MNIST dataset, and train a small convolutional neural network (CNN) model, i.e., LeNet [125, 246], on it. We use $M = 2, 4, 8, 16$ processes to collectively train the CNN model. We set all methods to keep half of the size of a full gradient vector, as in the previous subsection. The results are shown in Figure B.3. We can see that all methods spend approximately the same amount of time on communication, due to their identical sparsity levels. However, JOINTSPAR enjoys a much smaller total time per epoch than the baselines. This is because JOINTSPAR reduces both the gradient computation and communication time thanks to its



(a) MNIST (b) CIFAR10

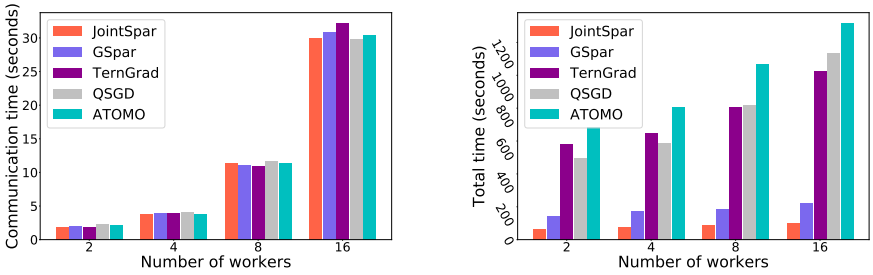
(c) CIFAR100

Figure B.2: Convergence in terms of wall-clock time, confirming our method JOINTSPAR's faster convergence than baselines.

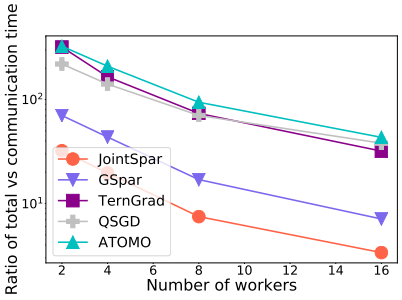
joint sparsification. As the number of processes increases, the ratio of total time vs. communication time becomes smaller. This increase in communication time is due to heavier overhead when there are more processes involved in communication. We can see that, for baseline methods, the ratio of total time vs. communication time is very large, i.e., at least 10 (up to 10^3). While being successful at reducing communication time, baseline methods causes the gradient computation time to become dominant among the total time. This verifies our very motivation to propose a new method that jointly performs sparsification on gradient computation and communication. From this experiment, we empirically observe that JOINTSPAR has the smallest ratio, meaning our method can effectively reduce the gradient computation time for each iteration.

B.11 Parameter Sensitivity

We investigate the convergence behavior of JOINTSPAR when changing the sparsity budget s. The smaller s is, the sparser the compressed gradient is. We vary sparsity budget among $10\%, 50\%, 90\%$ of the full gradient size. We use the same LeNet on MNIST dataset. The results are shown in Figure B.4, where x-axis and y-axis are number of epochs and loss value measured at the end of each epoch, respectively. Here, we use the convergence rate in terms of the number of epochs because we want to see effect of information change (caused by gradient sparsification) on the loss value. We observe that when there is less sparsification (i.e., larger sparsity budget), the time for each epoch is longer. Less sparsification can lead to faster convergence rate in training loss. This is because less compressed gradient loses less information of the full gradient. However, the convergence rate will not gain much if we increase s from 50% to 90%, due to some redundancy



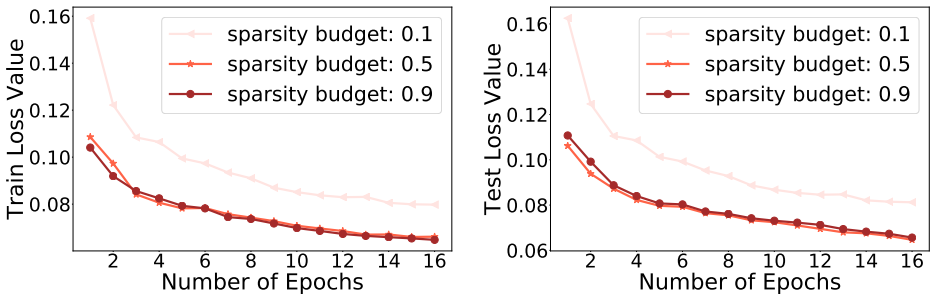
(a) Communication Time (b) Total Time



(c) Total to Communication Time Ratio

Figure B.3: (a) Communication time per iteration of each method, (b) total time per iteration, and (c) total time to communication time ratio.

in the gradient. On the other hand, the best convergence rate in testing loss is obtained when the sparsity budget is not too small or too large, likely because very large sparsity budget can lead to overfitting. Cross validation can be used to choose the optimal value of s.



(a) Train (b) Test

Figure B.4: Convergence rate for three different levels of sparsity budget s: low (0.1), medium (0.5), and high (0.9). Train loss is measured in (a), and test loss is measure in (b).

Appendix C
Proofs for Chapter 5

C.1 Proof of Theorem 5.1

According to Theorem 4.1 in [112], the convergence rate of Adam is

$$\begin{aligned}
&\sum_{t=1}^{T}[f_t(\theta_t) - f_t(\theta^*)] \\
\leq &\frac{D^2}{2\alpha(1-\beta_1)}\sum_{i=1}^{d}\sqrt{T\hat{v}_{T,i}} + \\
&\frac{\alpha(1+\beta_1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2}\sum_{i=1}^{d}\|g_{1:T,i}\|_2 + \\
&\sum_{i=1}^{d}\frac{D_\infty^2 G_\infty\sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\lambda)^2}
\end{aligned} \tag{C.1}$$

First, we show $\sum_{i=1}^{d}\sqrt{\hat{v}_{T,i}} \leq dG_\infty$

Proof.

$$
\begin{aligned}
&\sum_{i=1}^{d} \sqrt{\hat{v}_{T,i}} \\
=&\sum_{i=1}^{d} \sqrt{\sum_{j=1}^{T} \frac{1-\beta_2}{1-\beta_2^T} \beta_2^{T-j} g_{j,i}^2} \\
\leq&\sum_{i=1}^{d} \sqrt{\sum_{j=1}^{T} \frac{1-\beta_2}{1-\beta_2^T} \beta_2^{T-j} G_\infty^2} \\
=&G_\infty \sqrt{\frac{1-\beta_2}{1-\beta_2^T}} \sum_{i=1}^{d} \sqrt{\sum_{j=1}^{T} \beta_2^{T-j}} \\
=&G_\infty \sqrt{\frac{1-\beta_2}{1-\beta_2^T}} \sum_{i=1}^{d} \sqrt{\frac{1-\beta_2^T}{1-\beta_2}} \\
=&dG_\infty
\end{aligned}
\tag{C.2}
$$

□

Therefore, the above bound can be rewritten as

$$
\begin{aligned}
&\sum_{t=1}^{T} [f_t(\theta_t) - f_t(\theta^*)] \\
\leq&\rho_1 d\sqrt{T} + \sqrt{d}\rho_2 \sqrt{\sum_{t=1}^{T} \|\hat{G}_t\|^2} + \rho_3
\end{aligned}
\tag{C.3}
$$

where

$$
\begin{aligned}
\rho_1 &= \frac{D^2 G_\infty}{2\alpha(1-\beta_1)} \\
\rho_2 &= \frac{\alpha(1+\beta_1) G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \\
\rho_3 &= \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\gamma)^2}
\end{aligned}
\tag{C.4}
$$

Proof.

$$
\begin{aligned}
&\sum_{t=1}^{T}[f_t(\theta_t) - f_t(\theta^*)] \\
\leq&\rho_1 d\sqrt{T} + \rho_2 \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \rho_3 \\
=&\rho_1 d\sqrt{T} + d\rho_2 \sum_{i=1}^{d} \frac{1}{d} \sqrt{\sum_{t=1}^{T} (\hat{G}_{t,i})^2} + \rho_3 \\
&\text{(due to } \sqrt{\cdot} \text{ is concave)} \\
\leq&\rho_1 d\sqrt{T} + d\rho_2 \sqrt{\sum_{i=1}^{d} \frac{1}{d} \sum_{t=1}^{T} (\hat{G}_{t,i})^2} + \rho_3 \\
=&\rho_1 d\sqrt{T} + \sqrt{d}\rho_2 \sqrt{\sum_{t=1}^{T} \|\hat{G}_t\|^2} + \rho_3
\end{aligned}
\tag{C.5}
$$

□

we could get the following theorem for the convergence of Adam with sampling with replacement with batch size K.

$$
\begin{aligned}
&\sum_{t=1}^{T}[f_t(\theta_t) - f_t(\theta^*)] \\
=&\rho_1 d\sqrt{T} + \sqrt{d}\rho_2 \sqrt{\frac{1}{n^2 K} \sum_{t=1}^{T} \mathbb{E}\left[\sum_{j=1}^{n} \frac{\|g_t\|^2}{p_j^t}\right]} + \rho_3
\end{aligned}
\tag{C.6}
$$

Proof. Since $\hat{G}_t = \frac{1}{K}\sum_{k=1}^{K} \hat{g}_{I_k^t} = \frac{1}{K}\sum_{k=1}^{K} \frac{g_{I_k^t}}{n p_{I_k^t}}$, we have

$$
\begin{aligned}
\mathbb{E}[\|\hat{G}_t\|^2] \leq&\frac{1}{n^2 K^2} \sum_{k=1}^{K} \mathbb{E}\left[\frac{\|g_{I_k^t}\|^2}{p_{I_k^t}^2}\right] \\
=&\frac{1}{n^2 K^2} \sum_{k=1}^{K} \mathbb{E}\left[\sum_{j=1}^{n} \frac{\|g_j^t\|^2}{(p_j^t)^2} p_j^t\right] \\
=&\frac{1}{n^2 K^2} \sum_{k=1}^{K} \mathbb{E}\left[\sum_{j=1}^{n} \frac{\|g_j^t\|^2}{p_j^t}\right] \\
=&\frac{1}{n^2 K} \mathbb{E}\left[\sum_{j=1}^{n} \frac{\|g_j^t\|^2}{p_j^t}\right]
\end{aligned}
\tag{C.7}
$$

From previous step, we know

$$\begin{aligned}
&\sum_{t=1}^{T}[f_t(\theta_t) - f_t(\theta^*)] \\
\leq &\rho_1 d\sqrt{T} + \sqrt{d}\rho_2\sqrt{\sum_{t=1}^{T}\|\hat{G}_t\|^2} + \rho_3 \\
\leq &\rho_1 d\sqrt{T} + \sqrt{d}\rho_2\sqrt{\sum_{t=1}^{T}\frac{1}{n^2K}\mathbb{E}\left[\sum_{j=1}^{n}\frac{\|g_j^t\|^2}{p_j^t}\right]} + \rho_3 \\
= &\rho_1 d\sqrt{T} + \sqrt{d}\rho_2\sqrt{\frac{1}{n^2K}\sum_{t=1}^{T}\mathbb{E}\left[\sum_{j=1}^{n}\frac{\|g_j^t\|^2}{p_j^t}\right]} + \rho_3
\end{aligned} \tag{C.8}$$

□

C.2 Proof of Theorem 5.2

Theorem 5.2 follows from Theorem 5.1 and Lemma 5.1. Therefore, we focus on the proof of Lemma 5.1 here. We prove Lemma 5.1 using the framework of online learning with bandit feedback.

Online optimization is interested in choosing p^t to solve the following problem

$$\underset{p^t\in\mathcal{P}, 1\leq t\leq T}{min}\sum_{t=1}^{T}L_t(p^t) \tag{C.9}$$

where $L_t(p^t)$ is the loss that incurs at each iteration. Equivalently, the goal is the same as minimizing the pseudo-regret:

$$\bar{R}_T = \mathbb{E}\sum_{t=1}^{T}L_t(p^t) - \underset{p\in\mathcal{P}}{min}\,\mathbb{E}\sum_{t=1}^{T}L_t(p) \tag{C.10}$$

The following Algorithm C.1 similar to EXP3 could be used to solve the above problem.

Algorithm C.1 Online optimization

1: Input: stepsize α_p
2: Initialize x^1 and p^1.
3: **for** $t = 1, \cdots, T$ **do**
4: Play a perturbation $\tilde{p}^t$ of p^t and observe $\nabla L_t(\tilde{p}^t)$
5: Compute an unbiased gradient estimate $\hat{h}_t$ of $\nabla L_t(p^t)$, i.e. as long as $\mathbb{E}[\hat{h}_t] = \nabla L_t(p^t)$
6: Update p^t: $p^{t+1} \leftarrow argmin_{p\in\mathcal{P}}\left\{\langle\hat{h}_t, p\rangle + \frac{1}{\alpha_p}B_\phi(p, p^t)\right\}$.
7: **end for**

To be clear, the Bregman divergence $B_\phi(x, y) = \phi(x) - \phi(y) - \langle\nabla\phi(y), x - y\rangle$. Note that, the

updating step of Algorithm C.1 is equivalent to

$$\begin{aligned} w^{t+1} &= \nabla\phi^*(\nabla\phi(p^t) - \alpha_p\hat{h}_t) \\ p^{t+1} &= \underset{y\in\mathcal{P}}{argmin}\, B_\phi(y, w^{t+1}) \end{aligned} \tag{C.11}$$

We have the following convergence result for Algorithm C.1.

Proposition C.1. *The Algorithm C.1 has the following convergence result*

$$\begin{aligned} \bar{R}_T &= \mathbb{E}\sum_{t=1}^{T} L_t(\tilde{p}^t) - \min_{p\in\mathcal{P}} \mathbb{E}\sum_{t=1}^{T} L_t(p) \\ &\leq \frac{B_\phi(p, p^1)}{\alpha_p} + \frac{1}{\alpha_p}\sum_{t=1}^{T} \mathbb{E}B_{\phi^*}(\nabla\phi(p^t) - \alpha_p\hat{h}_t, \nabla\phi(p^t)) \\ &\quad + \sum_{t=1}^{T} \mathbb{E}\left[\|p_t - \tilde{p}_t\|\|\hat{h}_t\|_*\right] \end{aligned} \tag{C.12}$$

If $L_t(p^t)$ is linear (i.e. $L_t(p^t) = \langle l_t, p^t\rangle$), then we have

$$\begin{aligned} \bar{R}_T &= \mathbb{E}\sum_{t=1}^{T} L_t(\tilde{p}^t) - \min_{p\in\mathcal{P}} \mathbb{E}\sum_{t=1}^{T} L_t(p) \\ &\leq \frac{B_\phi(p, p^1)}{\alpha_p} + \frac{1}{\alpha_p}\sum_{t=1}^{T} \mathbb{E}B_{\phi^*}(\nabla\phi(p^t) - \alpha_p\hat{h}_t, \nabla\phi(p^t)) \\ &\quad + \sum_{t=1}^{T} \mathbb{E}\left[\|p_t - \mathbb{E}[\tilde{p}_t|p_t]\|\|\hat{h}_t\|_*\right] \end{aligned} \tag{C.13}$$

For example, let $\phi(p) = \sum_{j=1}^{n} p_j \log p_j$. Then its convex conjugate is $\phi^*(u) = \sum_{j=1}^{n} \exp(u_j - 1)$. In this case, due to Equation (C.11), the updating step of Algorithm C.1 becomes

$$w_j^{t+1} = p_j^t \exp(-\alpha_p\hat{h}_{t,j}), \forall 1 \leq j \leq n \tag{C.14}$$

The convergence result can also be simplified because

$$\begin{aligned} & B_{\phi^*}(\nabla\phi(p^t) - \alpha_p\hat{h}_t, \nabla\phi(p^t)) \\ =& \sum_{j=1}^{n} p_j^t \left(\exp(-\alpha_p\hat{h}_{t,j}) + \alpha_p\hat{h}_{t,j} - 1\right) \\ & \text{(assume } \hat{h}_{t,j} \geq 0 \text{ and due to } e^z - z - 1 \leq z^2/2 \text{ for } z \leq 0) \\ \leq& \frac{\alpha_p^2}{2}\sum_{j=1}^{n} p_j^t \hat{h}_{t,j}^2 \end{aligned} \tag{C.15}$$

Linear Case: Let's consider a special case where $L_t(p^t)$ is linear, i.e. $L_t(p^t) = \langle l_t, p^t\rangle$. Assume

p^t is a probability distribution, i.e. $\mathcal{P} = \{p : \sum_{i=1}^n p_i = 1, 0 \leq p_i \leq 1, \forall i\}$. In this case, $\nabla L_t(p^t) = l_t$. At iteration t, assume that we can't get the whole vector of l_t. Instead, we can get only one coordinate l_{t,J_t}, where J_t is sampled according to the distribution p^t. This is equivalent to $\tilde{p}_j^t = \mathbb{1}(j = J_t)$. Obviously,

$$\mathbb{E}[\tilde{p}_j^t] = \mathbb{E}[\mathbb{1}(j = J_t)] = \sum_{i=1}^n p_i^t \mathbb{1}(j = i) = p_j^t \tag{C.16}$$

Furthermore, to get the unbiased estimate of gradient, we set $\hat{h}_{t,j} = \frac{l_{t,j}\mathbb{1}(j=J_t)}{p_j^t}$. To verify $\hat{h}_{t,j}$ is indeed unbiased w.r.t. $l_{t,j}$,

$$\mathbb{E}[\hat{h}_{t,j}] = \mathbb{E}[\frac{l_{t,j}\mathbb{1}(j = J_t)}{p_j^t}] = \frac{l_{t,j}\mathbb{E}[\mathbb{1}(j = J_t)]}{p_j^t} = \frac{l_{t,j}}{p_j^t} p_j^t = l_{t,j} \tag{C.17}$$

For sampling multiple actions I^t, we have the following

$$\hat{h}_{t,j} = \frac{l_{t,j} \sum_{k=1}^K \mathbb{1}(j = I_k^t)}{K p_j^t} \tag{C.18}$$

And its convergence result is

$$
\begin{aligned}
\bar{R}_T &= \mathbb{E}\sum_{t=1}^{T} L_t(\tilde{p}^t) - \min_{p\in\mathcal{P}} \mathbb{E}\sum_{t=1}^{T} L_t(p) \\
&\leq \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{1}{\alpha_p}\sum_{t=1}^{T}\mathbb{E}B_{\phi^*}(\nabla\phi(p^t) - \alpha_p\hat{h}_t, \nabla\phi(p^t)) \\
&\quad + \sum_{t=1}^{T}\mathbb{E}\left[\|p_t - \mathbb{E}[\tilde{p}_t|p_t]\|\|\hat{h}_t\|_*\right] \\
&= \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{1}{\alpha_p}\sum_{t=1}^{T}\mathbb{E}B_{\phi^*}(\nabla\phi(p^t) - \alpha_p\hat{h}_t, \nabla\phi(p^t)) \\
&\leq \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\mathbb{E}\sum_{j=1}^{n} p_j^t\hat{h}_{t,j}^2 \\
&= \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\mathbb{E}\sum_{j=1}^{n} p_j^t\frac{l_{t,j}^2(\sum_{k=1}^{K}\mathbb{1}(j=I_k^t))^2}{(Kp_j^t)^2} \\
&\leq \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\mathbb{E}\sum_{j=1}^{n} p_j^t\frac{l_{t,j}^2\sum_{k=1}^{K}(\mathbb{1}(j=I_k^t))^2}{K(p_j^t)^2} \\
&= \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\mathbb{E}\frac{l_{t,J_t}^2}{p_{J_t}^t} \\
&= \frac{B_\phi(p,p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{j=1}^{n} l_{t,j}^2
\end{aligned}
\tag{C.19}
$$

We want to apply online optimization/bandit to learn p_j^t, in which case the loss at t-th iteration is $l_t(p^t) = \sum_{j=1}^{n}\frac{\|g_j(x^t)\|_*^2}{p_j^t}$. Because the loss $l_t(p^t)$ is a nonlinear function, we could use nonlinear bandit as a general approach to solve this. However, another simpler approach is to convert it to linear problem, where linear bandit could be used. Specifically, we have

$$
\begin{aligned}
&\sum_{t=1}^{T}\mathbb{E}\left[\sum_{j=1}^{n}\|g_j(x^t)\|_*^2\left(\frac{1}{p_j^t} - \frac{1}{p_j}\right)\right] \\
\leq &\sum_{t=1}^{T}\mathbb{E}\left[\left\langle -\left\{\|g_j(x^t)\|_{j,*}^2/(p_j^t)^2\right\}_{j=1}^{n}, p^t - p\right\rangle\right] \\
= &\sum_{t=1}^{T}\mathbb{E}\left[\left\langle -\left\{\|g_j(x^t)\|_{j,*}^2/(p_j^t)^2 + \frac{L^2}{p_{min}^2}\right\}_{j=1}^{n}, p^t - p\right\rangle\right]
\end{aligned}
\tag{C.20}
$$

This is equivalent to online linear optimization setting where $l_{t,j}(x) = -\frac{\|g_j(x)\|_{j,*}^2}{(p_j^t)^2} + \frac{L^2}{p_{min}^2}$. And

it's easy to see that $0 \leq l_{t,j} \leq \frac{L^2}{p_{min}^2}$. Then the convergence result is

$$\begin{aligned}
&\mathbb{E}\sum_{t=1}^{T} L_t(\tilde{p}^t) - \min_{p\in\mathcal{P}} \mathbb{E}\sum_{t=1}^{T} L_t(p) \\
&= \frac{B_\phi(p, p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{j=1}^{n} l_{t,j}^2 \\
&\leq \frac{B_\phi(p, p^1)}{\alpha_p} + \frac{\alpha_p}{2}\sum_{t=1}^{T}\sum_{j=1}^{n} \frac{L^4}{p_{min}^4} \\
&= \frac{B_\phi(p, p^1)}{\alpha_p} + \frac{\alpha_p}{2}\frac{nTL^4}{p_{min}^4} \\
&\leq \frac{R^2}{\alpha_p} + \frac{\alpha_p}{2}\frac{nTL^4}{p_{min}^4} \\
&\quad (\text{set } \alpha_p = \sqrt{\frac{2R^2 p_{min}^4}{nTL^4}}) \\
&= \frac{RL^2}{p_{min}^2}\sqrt{2nT}
\end{aligned} \tag{C.21}$$

C.3 Proofs about Comparison with Uniform Sampling

C.3.1 Proof of Lemma 5.2

Proof.

$$\begin{aligned}
&\mathbb{E}\sum_{j=1}^{n} \frac{\|g_j\|^2}{p_j} \\
=&\mathbb{E}n\sum_{j=1}^{n} \|g_j\|^2 \\
=&n\sum_{j=1}^{n} \mathbb{E}\|g_j\|^2 \\
\leq&n\sum_{j=1}^{n} \mathbb{E}\sum_{i=1}^{d} z_{j,i}^2 \\
=&n\sum_{j=1}^{n}\sum_{i=1}^{d} \beta_3 i^{-\gamma} j^{-\gamma} \\
=&\beta_3 n\sum_{j=1}^{n} j^{-\gamma}\sum_{i=1}^{d} i^{-\gamma} \\
\leq&\beta_3 n\log n\log d
\end{aligned} \tag{C.22}$$

□

C.3.2 Proof of Theorem 5.3

Proof. By plugging Lemma 5.2 into Theorem 5.2, we have

$$\begin{aligned}&\sum_{t=1}^{T}[f_t(\theta_t) - f_t(\theta^*)] \\ \leq&\rho_1 d\sqrt{T} + \frac{\rho_2\sqrt{\beta_3}}{\sqrt{K}}\sqrt{d\log d}\frac{\sqrt{n\log n}}{n}\sqrt{T} + \rho_3\end{aligned} \tag{C.23}$$

□

C.3.3 Proof of Lemma 5.3

Proof.

$$\begin{aligned}&\min_{p\geq p_{min}}\sum_{j=1}^{n}\mathbb{E}\frac{\|g_j\|^2}{p_j} \\ =&\min_{p\geq p_{min}}\sum_{j=1}^{n}\frac{1}{p_j}\mathbb{E}\sum_{i=1}^{d}z_{j,i}^2 \\ =&\min_{p\geq p_{min}}\sum_{j=1}^{n}\frac{1}{p_j}\sum_{i=1}^{d}\beta_3 i^{-\gamma}j^{-\gamma} \\ =&\min_{p\geq p_{min}}\sum_{j=1}^{n}\beta_3\frac{j^{-\gamma}}{p_j}\sum_{i=1}^{d}i^{-\gamma} \\ \leq&\min_{p\geq p_{min}}\sum_{j=1}^{n}\beta_3\frac{j^{-\gamma}}{p_j}\log d \\ =&\beta_3\log d\min_{p\geq p_{min}}\sum_{j=1}^{n}\frac{j^{-\gamma}}{p_j}\end{aligned} \tag{C.24}$$

From Proposition 5 in [162], we know

$$\min_{p\geq p_{min}}\sum_{j=1}^{n}\frac{j^{-\gamma}}{p_j} = O(\log^2 n) \tag{C.25}$$

Thus, we have

$$\begin{aligned}
&\min_{p \geq p_{min}} \sum_{j=1}^{n} \mathbb{E} \frac{\|g_j\|^2}{p_j} \\
=&\beta_3 \log d \min_{p \geq p_{min}} \sum_{j=1}^{n} \frac{j^{-\gamma}}{p_j} \\
=&O(\log d \log^2 n)
\end{aligned} \tag{C.26}$$

□

C.3.4 Proof of Theorem 5.4

Proof. It follows simply by plugging Lemma 5.3 into Theorem 5.2. □

C.4 AMSGrad with Bandit Sampling

We can also use bandit sampling to endow AMSGrad [183] with the ability to adapt to different training examples. Analogous to Theorem 5.1 in the main text, we have the following theorem regarding AMSGrad with Bandit Sampling.

Theorem C.1. *Assume the gradient $\hat{G}_t$ is bounded, $\|\hat{G}_t\|_\infty \leq G_\infty$, and $\alpha_t = \alpha/\sqrt{t}, \beta_1 = \beta_{11}$, and $\gamma = \beta_1/\sqrt{\beta_2} < 1$, $\beta_{1t} = \beta_1 \lambda^{t-1}$. Then, AMSGrad with Bandit Sampling achieves the following convergence rate*

$$R(T) \leq \rho_1' d\sqrt{T} + \rho_2' \sqrt{d(1+\log T)} \sqrt{\sum_{t=1}^{T} \|\hat{G}_t\|^2} + \rho_3'. \tag{C.27}$$

where

$$\begin{aligned}
\rho_1' =& \frac{D_\infty^2 G_\infty}{\alpha(1-\beta_1)} \\
\rho_2' =& \frac{\alpha}{(1-\beta_1)^2(1-\gamma)\sqrt{1-\beta_2}} \\
\rho_3' =& \frac{\beta_1 D_\infty^2 G_\infty}{2(1-\beta_1)(1-\lambda)^2}
\end{aligned} \tag{C.28}$$

Proof. According to Corrollary 1 from [183], we have

$$R(T) \leq \frac{\rho_1' \sqrt{T}}{G_\infty} \sum_{i=1}^{d} \hat{v}_{T,i}^{1/2} + \rho_2' \sqrt{(1+\log T)} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \rho_3'. \tag{C.29}$$

From section "Proof of Theorem 5.1", we know that

$$\sum_{i=1}^{d} \sqrt{\hat{v}_{T,i}} \leq dG_\infty \tag{C.30}$$

Therefore,

$$\begin{aligned}
R(T) \leq & \rho_1' d\sqrt{T} + \rho_2' \sqrt{1+\log T} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \rho_3' \\
= & \rho_1' d\sqrt{T} + d\rho_2' \sqrt{1+\log T} \sum_{i=1}^{d} \frac{1}{d} \sqrt{\sum_{t=1}^{T} (\hat{G}_{t,i})^2} + \rho_3' \\
& \text{(due to } \sqrt{\cdot} \text{ is concave)} \\
\leq & \rho_1' d\sqrt{T} + d\rho_2' \sqrt{1+\log T} \sqrt{\sum_{i=1}^{d} \frac{1}{d} \sum_{t=1}^{T} (\hat{G}_{t,i})^2} + \rho_3' \\
= & \rho_1' d\sqrt{T} + \sqrt{d(1+\log T)} \rho_2' \sqrt{\sum_{t=1}^{T} \|\hat{G}_t\|^2} + \rho_3'
\end{aligned} \tag{C.31}$$

□

Simiarly, we could derive theorems about AMSGrad with Bandit Sampling that are analogous to Theorems 5.2, 5.3, 5.4 for ADAMBS. We omit the details here.

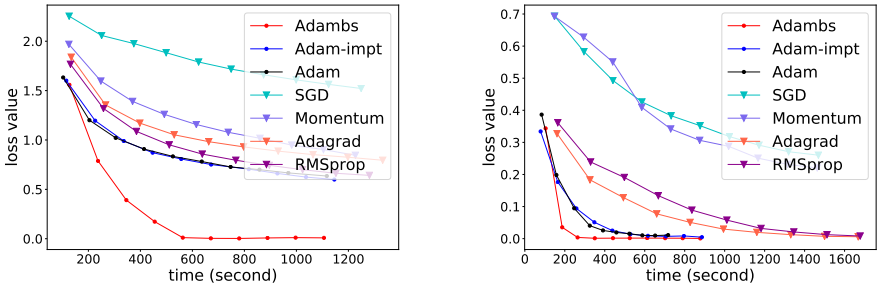
C.5 Additional Experiments

C.5.1 Comparison with Basic SGD Variants

In the main paper, we compared our method with Adam and Adam with importance sampling. Here, we compare with some basic SGD variants (e.g., SGD, Momentum, Adagrad and RMSprop). As shown in Figure C.1, these basic SGD variants have worse performance than Adam-based methods.

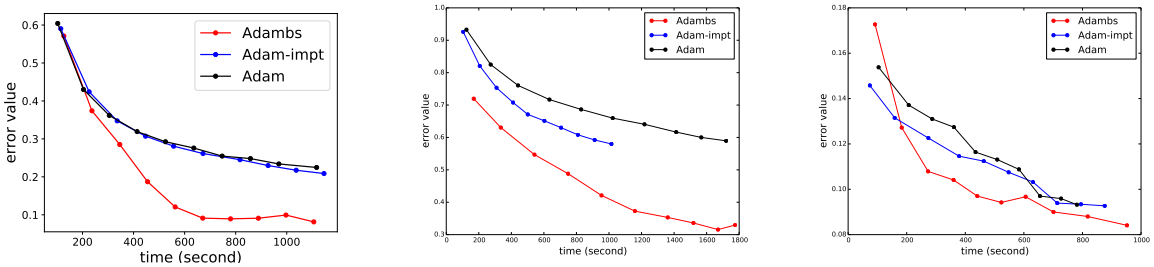
C.5.2 Comparison on Error Rate

In the main paper, we have shown the plots of loss value vs. wall clock time. Some might also be interested in the convergence curves of loss values. Here, we include some plots of error rate vs. wall clock time, as shown in Figure C.2 and Figure C.3. They demonstrated the faster convergence of our method in terms of the error rate.



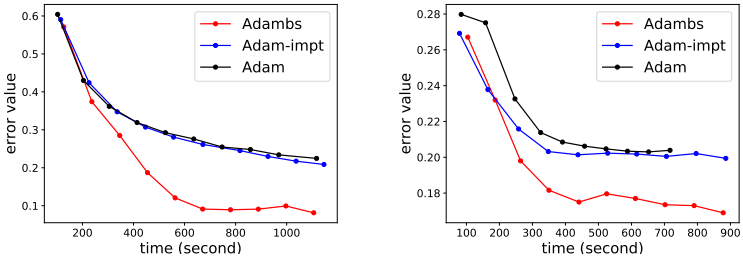
(a) CNN on CIFAR10 (b) RCNN on IMDB

Figure C.1: Experiments comparing with SGD variants.



(a) CNN on CIFAR10 (b) CNN on CIFAR100 (c) CNN on Fashion MNIST

Figure C.2: Error Plot of CNN models



(a) RNN on Fashion MNIST (b) RCNN on IMDB

Figure C.3: Error Plot of RNN and RCNN models

Appendix D
More Results for Chapter 6

D.1 Scenarios Where Selective Classification Is Needed

In a nutshell, selective classification is needed in settings where prediction error of the model can cause critical consequences (e.g., in medical diagnosis or self-driving cars). In these settings, we would like the model to proactively indicate when it's not confident in its prediction, so that human intervention is triggered.

Medical diagnosis. In medical diagnosis [117], we can train a machine learning model to predict if a patient has a tumor based on the MRI image. When the model is not confident enough, we can ask an oncologist, and even call upon an urgent meeting of multiple oncologists across different hospitals for a group consultation.

Self-driving cars. In self-driving cars [73], we can train a machine learning model to detect the traffic light, which will help the self-driving car to decide what to do at cross-roads. However, when the model is not confident enough (e.g., traffic light is not clear due to rain or obstruction), the human driver should be alerted and take control of the car.

D.2 Detailed Explanations about Target Coverage Match

We elaborate on why it's ideal to match the proportion of easy examples in each mini-batch with the target coverage as briefly mentioned in Section 6.4.1. We start by showing that the empirical coverage $\hat{\phi}$ will converge to the target coverage c. Then we show that a mini-batch in which the proportion of easy examples is the same as the target coverage is ideal. Therefore, a mini-batch which has the proportion of easy examples different from the target coverage will misguide the selection function regarding which examples should be accepted or rejected, which further affects the prediction head's learning progress.

Empirical coverage converges to target coverage. As a result of minimizing Equation 5, the empirical coverage $\hat{\phi}$ will converge to the target coverage c. Without being too technically rigorous, we demonstrate this by showing that, whether or not the empirical coverage $\hat{\phi}$ is larger than target coverage c, the loss $\mathcal{L}_{(f,g)}$ (i.e., left hand side of Equation 6.5) will become smaller if empirical coverage is moved closer to target coverage.

- When the empirical coverage is larger than the target coverage, the second term of Equation 6.5 becomes 0. Note that the first term $\hat{r}$ is the average loss of accepted examples. By simply

rejecting the example with the highest loss (i.e., decreasing the empirical coverage), the first term $\hat{r}$ will become smaller while keeping the second term still 0. Therefore, when the empirical coverage is larger than the target coverage, decreasing the empirical coverage will make the loss $\mathcal{L}_{(f,g)}$ smaller.

- When the empirical coverage is smaller than the target coverage, the second term is a positive value. As long as the hyperparameter λ is large enough to make the second term dominant, the loss $\mathcal{L}_{(f,g)}$ will become smaller by increasing the empirical coverage. As suggested in [72], the hyperparameter λ is chosen to be a large value 32 in our experiments. Therefore, when the empirical coverage is smaller than the target coverage, increasing the empirical coverage will make the loss $\mathcal{L}_{(f,g)}$ smaller.

Mini-batch should ideally match target coverage. We now show that a mini-batch in which the proportion of easy examples is the same as the target coverage is ideal. We demonstrate it by showing that, when the mini-batch has the proportion of easy examples different from the target coverage, as a result of minimizing the loss during the training, even an already perfect selection function tends to move to the direction where mistakes will be made (i.e., reject easy examples or accept hard examples). Note that, by an already perfect selection function, we mean that the selection function accepts or rejects examples based on the ground truth (i.e., reject hard examples or accept easy examples).

- When the proportion of easy examples in the mini-batch is larger than the target coverage, a perfect selection function gives the empirical coverage on this mini-batch that is larger than target coverage. As we have just discussed above, when the empirical coverage is larger than the target coverage, decreasing the empirical coverage will make the loss $\mathcal{L}_{(f,g)}$ smaller. It implies that the selection function tends to reject more examples, which are actually accepted by a perfect selection function.

- When the proportion of easy examples in the mini-batch is smaller than the target coverage, a perfect selection function gives the empirical coverage on this mini-batch that is smaller than target coverage. As we have just discussed above, when the empirical coverage is smaller than the target coverage, increasing the empirical coverage will make the loss $\mathcal{L}_{(f,g)}$ smaller. It implies that the selection function tends to accept more examples, which are actually rejected by a perfect selection function.

D.3 More Details about Experimental Setup

The moderate-sized convolutional network architecture contains 8 convolutional layers with $32, 32, 64, 64, 128, 128, 256, 256$ filters respectively. The filter size is 3×3 for the first 6 layers and 2×2 for the last 2 layers. There is a 2×2 max-pooling layer and a dropout layer with rate 0.25 after every two layers. Following the convolutional layers, there is a fully-connected layer with 512 units and a dropout layer with rate 0.25. At last, another fully connected layer is used to match the number of classes in the dataset. We omit the detailed layer-by-layer description of VGG-16 as it is described in [211]. In addition, the prediction head f and auxiliary head h are fully-connected softmax layers, and the selection head g is a fully-connected hidden layer with 513

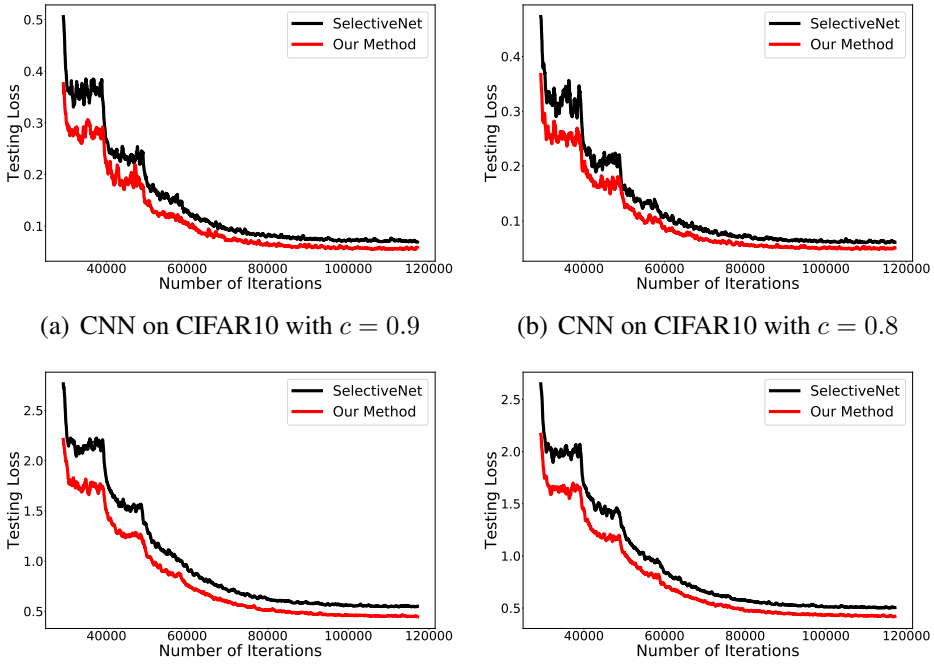
Table D.1: Ablation study with moderate-sized CNN on CIFAR10. Testing error rates (in %) are shown below.

Coverage	Our Method	SelectiveNet	Inception	Vanilla-Selection
0.95	$\mathbf{7.32 \pm 0.05}$	7.57 ± 0.1	7.43 ± 0.07	7.51 ± 0.04
0.9	$\mathbf{5.63 \pm 0.04}$	6.07 ± 0.1	5.87 ± 0.05	5.98 ± 0.04
0.85	$\mathbf{4.30 \pm 0.04}$	4.53 ± 0.08	4.37 ± 0.03	4.42 ± 0.02
0.8	$\mathbf{3.02 \pm 0.02}$	3.47 ± 0.07	3.14 ± 0.05	3.35 ± 0.04
0.75	$\mathbf{2.16 \pm 0.04}$	2.45 ± 0.07	2.23 ± 0.04	2.38 ± 0.09
0.7	$\mathbf{1.47 \pm 0.02}$	1.67 ± 0.08	1.59 ± 0.02	1.66 ± 0.03

neurons, followed by batch normalization, ReLU activation and another fully connected layer to one output neuron with a sigmoid activation. These architectures have been used in the recent selective classification and curriculum learning papers [72, 83, 251].

D.4 More Experimental Results

The complete results of ablation study are shown in Table D.1.



(a) CNN on CIFAR10 with $c = 0.9$

(b) CNN on CIFAR10 with $c = 0.8$

(c) VGG on CIFAR100 with $c = 0.9$

(d) VGG on CIFAR100 with $c = 0.8$

Figure D.1: Testing loss vs. the number of iterations for different models on different datasets. Our method convergences faster by achieving a lower testing loss for the same number of iterations.

Table D.2: The number of iterations to reach the target loss for training moderate-sized CNN on the CIFAR10 dataset. The target loss value is the lowest testing loss achieved by SELECTIVENET for each coverage rate.

Coverage	# of Iterations		Impr.
	Our Method	SelectiveNet	
0.95	332k	694k	52%
0.9	73k	116k	37%
0.85	97k	459k	79%
0.8	74k	115k	36%
0.75	75k	109k	31%

D.5 Convergence Speed

It has been shown that curriculum learning can improve the convergence speed when training on various classical neural network models [83, 128]. In this section, we investigate whether this applies to our method as well. We plot the curves of testing loss versus the number of iterations under different settings in Figure D.1. Specifically, Figures D.1 (a) and (b) correspond to training moderate-sized CNN on the CIFAR10 dataset with a target coverage of $c = 0.9$ and $c = 0.8$, respectively, while Figures D.1 (c) and (d) correspond to training VGG-16 on the CIFAR100 dataset with a target coverage of $c = 0.9$ and $c = 0.8$, respectively. We can see that our method has a better convergence rate than the baseline because our method achieves a lower testing loss for the same number of training iterations than SELECTIVENET. To better understand how faster our method's converges, we also measure the number of iterations to reach the same testing loss as summarized in Table D.2. Specifically, the testing loss to reach is set as the lowest testing loss achieved by SELECTIVENET. It can be seen that our method can reduce the the number of iterations by up to 80%. This demonstrates that our method is significantly faster than the baseline during training.

D.6 Coverage Violation

Our method leverages the difficulty scores to calibrate the ratio of the difficult and easy examples in each mini-batch. This could effectively reduce the discrepancy between the target coverage and empirical coverage (i.e., the coverage on the testing set before applying the calibration process as described in Equation 8). We measure the empirical coverage and how much it differs from the target coverage (i.e., violation) as summarized in Table D.3. As expected, the empirical coverage achieved by our method is closer to the target coverage, resulting in smaller violation compared to the baseline (i.e., SELECTIVENET). The relative reduction of the violation achieved by our method is larger than 10% for most values of target coverage (i.e., 90%, 70%). Especially, when the target coverage is small, i.e., 70%, the violation reduction is as high as 41.7%.

Table D.3: Coverage violation of moderate-sized CNN on CIFAR10. Empirical Coverage (in $\%$) are shown for different target coverage rates. Our method achieves smaller coverage violation (difference between target coverage and empirical coverage) than SELECTIVENET.

Coverage (in $\%$)	**Our Method**		**SelectiveNet**		**Violation Reduction**
	Empirical Coverage	**Violation**	**Empirical Coverage**	**Violation**	
95	90.2 ± 0.1	4.8	90.0 ± 0.1	5.0	4%
90	84.7 ± 0.1	5.3	84.1 ± 0.2	5.9	10.2%
85	80.7 ± 0.2	4.3	80.2 ± 0.1	4.8	10.4%
80	76.6 ± 0.1	3.4	76.3 ± 0.2	3.7	8.1%
75	73.1 ± 0.1	1.9	72.7 ± 0.3	2.3	17.4%
70	69.3 ± 0.1	0.7	68.8 ± 0.1	1.2	41.7%