Efficient Neural Network Based Systems on Mobile and Cloud Platforms

by

Jiachen Mao

Department of Electrical and Computer Engineering
Duke University

Date: ________________________

Approved:

__________________________
Yiran Chen, Advisor

__________________________
Hai Li

__________________________
Benjamin C Lee

__________________________
Maria Gorlatova

__________________________
Qiang Qiu

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University

2020
Abstract
Efficient Neural Network Based Systems on Mobile and Cloud Platforms
by
Jiachen Mao
Department of Electrical and Computer Engineering
Duke University

Date: __________________________
Approved:

___________________________
Yiran Chen, Advisor

___________________________
Hai Li

___________________________
Benjamin C Lee

___________________________
Maria Gorlatova

___________________________
Qiang Qiu

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University 2020
Abstract

In recent years, machine learning, especially neural networks arouses unprecedented influence in both academia and industry. The reason lies in the state-of-the-art performance of neural networks on many critical applications such as object detection, translation, and games. However, the deployment of neural network models on resource-constrained devices (e.g. edge devices) is challenged by their heavy memory and computing cost during execution. Many efforts have been done in previous literature for efficient execution of neural networks, including the perspectives of hardware, software, and algorithm. My research focus during my Ph.D. study is mainly on software, and algorithm targeting at mobile platforms. More specifically, we emphasize the system design, system optimization, and model compression of neural networks for better mobile user experience.

From the system design perspective, we first propose MoDNN – a local distributed mobile computing system for DNN testing. MoDNN can partition already trained DNN models onto several mobile devices to accelerate DNN computations by alleviating device-level computing cost and memory usage. Two model partition schemes are also designed to minimize non-parallel data delivery time, including both wakeup time and transmission time. Then, we propose AdaLearner – an adaptive local distributed mobile computing system for DNN training. To exploit the potential of our system, we adapt the neural networks training phase to mobile device-wise resources and fiercely decrease the transmission overhead for better system scalability.
From the system optimization perspective, we propose MobiEye, a cloud-based video detection system optimized for deployment in real-time mobile applications. MobiEye is based on a state-of-the-art video detection framework called Deep Feature Flow (DFF). MobiEye optimizes DFF by three system-level optimization methods.

From the model compression perspective, we propose Tprune, a model analyzing and pruning framework for Transformer. In TPrune, we first proposed Block-wise Structured Sparsity Learning (BSSL) to analyze Transformer model property. Then, based on the characters derived from BSSL, we apply Structured Hoyer Square (SHS) to derive the final compressed models.

The realization of the projects during my PhD study could contribute to the current research on efficient neural network execution and thus result in more user-friendly and smart applications on edge devices for more users.
Contents

Abstract iv
List of Tables xi
List of Figures xii
Acknowledgements xv
1 Introduction 1
   1.1 Testing on Mobile Devices ........................................... 1
   1.2 Training on Mobile Devices .......................................... 4
   1.3 DNN Models for Video Detection System .......................... 6
   1.4 DNN Models for Neural Machine Translation ...................... 7
2 Distributed Testing on Mobile Devices 11
   2.1 Preliminary .............................................................. 11
      2.1.1 Testing on Mobile Devices ...................................... 11
      2.1.2 Traditional Convolutional Neural Networks .................. 12
      2.1.3 Model Pruning ...................................................... 12
   2.2 System Framework of MoDNN ......................................... 14
      2.2.1 Definition of Terminologies and Variables .................. 15
      2.2.2 Network Establishment and Setup .............................. 16
      2.2.3 Input Partition for CL .......................................... 16
      2.2.4 Weight Partition for Sparse FLs .............................. 18
2.2.5 Processing Flow of DNN ........................................ 22
2.3 Experiments ....................................................... 23
  2.3.1 Environment Setup and Test bench Selection ............... 23
  2.3.2 Data Delivery Time Evaluation of BODP .................... 25
  2.3.3 Transmission Size Evaluation of MSCC & FGCP ........... 26
  2.3.4 Overall Evaluation of MoDNN .............................. 27
2.4 Conclusion ....................................................... 27

3 Distributed Training on Mobile Devices .......................... 28
  3.1 Preliminary ..................................................... 29
    3.1.1 Backpropagation with SGD ............................... 29
    3.1.2 Parallel Schemes for Distributed NN Training ........... 30
    3.1.3 Quantization Methods ..................................... 32
  3.2 Methods ....................................................... 34
    3.2.1 Software Components .................................... 36
    3.2.2 System Architecture of AdaLearner ....................... 37
  3.3 Computation Cost in Adaptive Scheduler ....................... 39
  3.4 Communication Cost in Adaptive Scheduler ................. 41
    3.4.1 1-bit Quantization with Error Feedback .................. 42
    3.4.2 Gradients Aggregating Scheme .......................... 43
    3.4.3 Computation / Communication Tradeoff .................... 43
  3.5 Incorporate Adaptive Scheduler to AdaLearner ............... 45
  3.6 Experiments .................................................. 45
    3.6.1 System Implementation and Experiments Setup ............. 45
    3.6.2 Evaluation of Efficient Computation Architecture ....... 47
    3.6.3 Evaluation of Efficient Communication Architecture ...... 48
4 Efficient Video Detection 51

4.1 Preliminary 51
  4.1.1 DNN Model Profiling on Mobile Devices 51
  4.1.2 DNNs in Object Detection 52
  4.1.3 Motion Vector in H.264 Video Codec 53
  4.1.4 Feature Map Sparsity in DNNs 54

4.2 Methods 55

4.3 System Framework 55
  4.3.1 Video Detection Framework (DFF) 56
  4.3.2 Asynchronous Computation (ADFF) 58

4.4 Video-based Dynamic Scheduling 60
  4.4.1 Motion Vector and Optical Flow 61
  4.4.2 H.264 Motion Vector Extraction 62
  4.4.3 Video-based Dynamic Scheduling (VDS) 63

4.5 Inference with Spatial Sparsity 63
  4.5.1 Spatial Sparsity Inference (SSI) 64
  4.5.2 Realization of SSI Index Layer 65
  4.5.3 Realization of SSI Convolutional Layer 67
  4.5.4 Applying SSI on FlowNet 68
  4.5.5 Comparison of SSI with Related Works 69

4.6 Experiments 70
  4.6.1 Experimental Setup 70
  4.6.2 Evaluation of Communication Cost 72
5 Efficient Neural Machine Translation 82

5.1 Preliminary ................................. 82
  5.1.1 Model Profiling on Mobile Devices ................. 82
  5.1.2 Transformer Model Pruning .......................... 83
  5.1.3 Model Pruning with Regularization ................. 84

5.2 Methods ...................................... 85

5.3 Transformer Redundancy Analysis ....................... 85
  5.3.1 Analyzing Model Components ....................... 85
  5.3.2 Analyzing Target ................................ 86
  5.3.3 Block-wise Structured Sparsity Learning (BSSL) .... 87
  5.3.4 Observation using BSSL Analysis ................... 90
  5.3.5 Conclusion of BSSL Analysis ....................... 92

5.4 Transformer Pruning ................................ 94
  5.4.1 Structured Hoyer Square (SHS) ..................... 94
  5.4.2 Pruning Strategy .................................. 96

5.5 Experiments .................................... 97
List of Tables

2.1 Overall evaluation of MoDNN with 2-4 worker nodes. . . . . . . . . 26
4.1 Profiling of 2 state-of-the-art DNN models. . . . . . . . . . . . . . 51
4.2 Configuration of convolutional layers in FlowNet. . . . . . . . . . 78
5.1 Summary of our line-wise pruning results, previous head-wise pruning results [1] and line-wise pruning results [2]. . . . . . . . . . . . . . 107
List of Figures

1.1 Average computing time and memory usage of the layers in DNNs. ........ 3
1.2 Transformer Architecture in [3] ........................................ 8
2.1 GEMM steps: (a) Patch to column step. (b) Matrix multiplication step. 11
2.2 Two partition schemes for CLs. ........................................ 17
2.3 Hybrid matrix representation for SpMV and GEMV. ..................... 18
2.4 Two-stage processing: MSCC&FGCP. ................................ 19
2.5 Processing flow of convolutional-related layers. ......................... 22
2.6 Processing flow of sparse FLs. ........................................ 22
2.7 Experimental setup. .................................................... 23
2.8 Performance character. of Nexus 5. .................................. 23
2.9 Computing time and transmission time of CLs in VGG-16. ............ 24
2.10 Transmission size decrease ratio of MoDNN to baseline. .............. 24
3.1 (a) Model parallelism and (b) Data parallelism in distributed train- ing. [4]. ................................................................. 30
3.2 Two network topology for distributed training. [5]. ...................... 31
3.3 System overview of AdaLearner. ..................................... 36
3.4 Distributed architecture for efficient computation. ....................... 37
3.5 Distributed architecture for efficient communication. .................. 38
3.6 Message format in AdaLearner. ...................................... 39
3.7 Time consumption with different minibatch sizes. ...................... 41
3.8 Lookup table for minibatch size and computing time. .......... 41
3.9 Tradeoff between computation and communication costs. ....... 42
3.10 Experimental setup. ............................................. 45
3.11 Time consumption on three neural network models with efficient computation architecture. ............................................. 47
3.12 Time consumption on three neural network models with efficient communication architecture. ............................................. 48
3.13 Accuracy results using 1-bit quantization under different number of WNs. .......................................................... 49
4.1 System overview of our video detection system. ................. 55
4.2 Execution flow in DFFdetection framework [6]. ................. 56
4.3 Original sequential execution of DFF (top) vs proposed asynchronous execution of DFF (bottom). ................................. 58
4.4 Optical flow using Lucas Kanade (left) and Gunnar Farneback (right). ................................................................. 61
4.5 Standard H.264 GoP pattern (top) and adopted H.264 GoP pattern in our system (bottom). ................................................. 62
4.6 Feedback computation mask. ........................................ 64
4.7 Brightness error computation mask. ................................ 64
4.8 Original layers (left) and layers with SSI functionality (right). 65
4.9 Realization of SSI index operation on GPU. ...................... 66
4.10 Procedures of SSI convolution operation. ......................... 67
4.11 Network Architecture of FlowNet in DFF. ....................... 68
4.12 (a) Motion value histogram on ImageNet VID with H.264, and (b) encode/decode time on tested hardware. ........................ 71
4.13 Communication time per frame under different bandwidths and resolutions. ............................................................... 72
4.14 Computation time comparison between sequential and asynchronous execution of DFF. ............................................ 73

xiii
4.15 mAP under different average key frame interval with static scheme and VDS scheme. 74
4.16 Sparsity-accuracy tradeoff with SSI. 75
4.17 Speedup of SSI index operation. 76
4.18 Speedup of SSI convolution operation. 77
4.19 Layer-wise speedup on FlowNet with SSI convolution. 78
4.20 Memory consumption of original, SSI, and SBNNet Convolution. 79
5.1 Weight distribution of (a) ResNet and (b) Transformer. 82
5.2 Blocked-wise structured sparsity learning. 87
5.3 Sparsity visualization of Transformer encoder with BSSL. 90
5.4 Sparsity visualization of Transformer decoder with BSSL. 91
5.5 Comparing the $L_1$ and the Hoyer regularizer of a 2-D vector. [7] 94
5.6 Comparison between SSL and SHS regularizers with low pruning sparsity. 98
5.7 Comparison between SSL and SHS regularizers with high pruning sparsity. 99
5.8 Training loss with different line-wise pruning strategies under different sparsities. 100
5.9 Layer-wise sparsities of Transformer encoder under different model sparsities. 101
5.10 Layer-wise sparsities of Transformer decoder under different model sparsities. 101
5.11 Execution time on with different string lengths. 103
5.12 Execution time on different mobile devices. 103
5.13 Execution time on different numbers of threads. 104
5.14 Time consumption when executing pruned models of different sparsity. 105
5.15 BLEU score during the fine-tuning procedure. 106
Acknowledgements

During my Ph.D. Study, I received many precious ideas, advices, and helps from my advisors, mentors, colleagues, friends and family. First of all, I want to express my sincere gratefulness to my advisor Yiran Chen and co-advisor Hai Li. When I applied for the Ph.D. position of Prof. Chen and Prof. Li’s lab, neither my GPA nor my TOEFL English is outstanding among all the applicants. However, Prof. Chen and Prof. Li still gave me the opportunity of being a full-time Ph.D. student. During my research, Prof. Chen and Prof. Li consistently provides me with very important guidance. For example, the entire idea of my first work (distributed testing on mobile platforms (MoDNN)) is from my advisor Yiran Chen. After giving me the high-level idea, Prof. Chen continuously supply me with guidance about how to implement the system, how to tackle the obstacles, and how to conduct the experiments. When there exist some different opinion between Prof. Chen and me. Prof. Chen and Prof. Li always respect my opinion and give me enough trust. The caring from my advisors is not only expressed in my study and research. They also provide me enormous help in my daily life. During the pandemic of COVID-19 this year, Prof. Chen and Prof. Li distributed masks to all members of our lab to ensure our health. Secondly, I want to thank my committee members of my qualification, preliminary, and final defense. Prof. Benjamin C. Lee, focuses on computer architecture and systems, who is both my teacher of computer architecture course and my committee member, giving me tremendous ideas and suggestions during my research. Prof.
Maria Gorlatova, focusing on Internet of Things systems and applications, who not only guides my research works but also provided me with papers in the field of embedded system. Prof. Qiang Qiu and Prof. Galen Reeves, who are experts of signal and information processing and machine learning. They provide me with many insights and ideas in their professional field. I received many precious advice and comments from my committee members, which let me aware of the pros and cons of my research and have a more comprehensive view of my works. Thirdly, I want to thank my colleagues, co-authors, and friends. They are the ones who are accompanied with me the longest of times during my daily P.h.D. study. My mentors of embedded system group of our lab, Xiang Chen and Kent W. Nixon, leads me toward the world of research. They teach how to do paper review and conduct experiments. Before submissions, they always do a detailed proof-reading for me, point out the problem of my paper and help me to solve them. I also want to thank my co-authors, Linghao, Qin, Wei, Chunpeng, Huanrui, Ang, Chuhan, Bing. They give me a lot of support during the process of my research. We study and improve with each other. I still remember the time when Linghao and I talked about our collaboration work excitedly from lab to home in the mid-night. I am so grateful to have so many colleagues, co-authors, and friends in my Ph.D. life, with whom I could exchange ideas, collaborate, and study. Last, I want to say thanks to my cousin, Yaojun, who is also graduated as a Ph.D. under the guidance of Prof. Yiran Chen, gives me a lot of tips on how to work as a Ph.D. and teaches me to always be positive and active in both research and daily life. I also want to thank my parents, Junliang and Zongxiu, who always support and care about me. Thanks to Chuhan, my colleague, friend, co-author, who finally becomes my wife last year. In a word, I am so lucky to have all my advisors, committee members, mentors, friends, and family members appear in my life, especially in my Ph.D. life. Thanks a lot.
1

Introduction

1.1 Testing on Mobile Devices

With the emergence of massive mobile network bandwidth, novel interactive applications in mobile devices utilize more multimedia processing and boost the performance requirement of object recognition, classification, language processing, etc.

Deep Neural Network (DNN) is a widely used technique in object recognition and classification tasks due to its high accuracy, scale flexibility, and data-driven property. However, optimal performance incurs huge computing costs and memory consumption. A representative example is VGG [8], which achieved satisfactory performance on ImageNet Large Scale Visual Recognition Challenge 2014 (ILSVRC14), using 15M neurons, 144M parameters and 3.4B connections. If deployed on Nexus5, VGG consumes approximately 16 seconds for a single image identification procedure, which is intolerable in practical usage.

A significant gap exists when we want to apply DNN models to mobile devices. The contradiction is between the limited mobile resource and the high requirement of DNNs, which requires substantial energy consumption. Many research works
have been done to fill this gap. To efficiently offload the huge computing cost and memory consumption to outside infrastructure, the client-server computing paradigm is the most straight-forward solution. [9] research the data offloading scheme in a pipelined machine learning structure; [10] focuses on the establishment of an efficient distributed parameter server framework for the training phase of DNNs.

To enable local execution of DNN models on mobile devices, attempts have also been made in model compression during the training phase of DNN. [11] deeply compressed the DNN models using a three-stage pipeline: pruning, trained quantization, and Huffman coding; [12] used a low-cost hash function to group weights into hash buckets for parameter sharing purpose. Because the convolutional layer consumes intensive computation cost. Many efficient layer structures are invented to substitute the original convolutional layers. MobileNet [13] decompose convolutional layer to a $D \times D$ deep-wise convolution followed by a $1 \times 1 \times M$ pixel-wise convolution. Because $D \times D$ deep-wise convolution only contributes a tiny part of the total computation cost, the main cost is on $1 \times 1 \times M$ pixel-wise convolution. Therefore, the theoretical speedup of MobileNet is about $D \times D$. MobileNetV2 further optimize MobileNet by adding linear bottleneck layer [14]. Instead of decoupling the convolutional layer into multiple concatenated layers in depth. SqueezeNet decouples the convolutional layers both in depth and width [15]. SqueezeNet first uses $1 \times 1$ pixel-wise convolution to reduce the input channel number. Then, the output feature maps will go through two convolutional layers, which are $1 \times 1$ pixel-wise convolution and $3 \times 3$ convolution. The whole procedure is named as a Fire Module in SqueezeNet. In [16], ShuffleNet identifies the key idea of ShuffleNet: for each feature map in the neural network, they are divided into many groups of feature maps in the channel dimension. For example, if the original channel size is 512 and the group size is 4, then the convolution filters’ channel size is $512/4$ (128). The computation cost is one-quarter of the original model. The inspiration of ShuffleNet is that the author thinks that
not all the features are correlated with each other. However, by using ShuffleNet, the output features are derived by part of the previous feature map. Inspired by the work called "Deep networks with stochastic depth" [17], the authors of DenseNet find that 2 important facts. One is that the feature of \( n \) layer may not only rely on \( n - 1 \) layer but also rely on previous layers. The other is that the feature of different layers consists of much redundancy. Therefore, in the structure of DenseNet, the current output feature map of layer \( n \) is the output of all the previous feature maps. In this way, each layer can learn fewer features than before. One obvious defect of DenseNet is its high memory consumption. The reason is that current neural network frameworks realize concatenate operation by allocating a new memory space. In order to deal with high memory consumption, CondenseNet further optimize DenseNet by adding the idea of group convolution in ShuffleNet [18].

Convolutional layers and fully-connected layers are two key layers in modern DNNs. To further dig out their individual properties, Fig. 1.1 summarizes the layer-wise experiment results performed on smartphones (Nexus 5) from the perspectives of computing cost and memory consumption. Without loss of generality, three famous models of different scales are tested, which include Lenet [19], Inception-BN [20] and VGG [8]. Two common properties are derived as follows:
• Convolutional layers drain approximately 87% to 98% of the over-all computing cost.

• Fully-connected layers cost from at least 94% of the memory consumption caused by large model weights.

Sparse fully-connected layers are another emerging research topic arousing tremendous attentions [21], which achieves high layer sparsity with marginal or even zero accuracy drop [11]. For example, the sparsity of fully-connected layers in VGG-16 reaches over 96% without accuracy drop [22].

Different from previous works on network design, we focus on how to partition and deploy a single large DNNs on several mobile devices and execute it on those devices in parallel. In section 2, a more dedicated illustration will be presented about the framework of our proposed local distributed mobile computing system. Furthermore, algorithms used in our proposed model partition schemes will be provided in detail so that the reason why our partition schemes can efficiently decrease the data delivery time will be clarified.

1.2 Training on Mobile Devices

Triggered by the unprecedented development in neural networks, a broad range of applications have emerged because of their state-of-the-art performance, e.g. object classification, language processing, and motion estimation [8, 23, 24]. Due to the plethora of embedded sensors available on the platform, mobile devices represent the largest market where these applications are fed into practical use, enabling mobile navigation systems [25], speech-based human-device interaction, and augmented reality games [26]. However, a significant limitation of neural networks is the requirement for the neural network models to be trained before their practical use as
testing. In comparison with the testing phase, the training phase has significantly higher computational complexity and memory consumption than the training phase.

As a subset of machine learning, deep neural networks (DNNs) have come to outperform many traditional algorithms, even surpassing human-level performance, e.g. AlphaGo [27]. However, a significant limitation of DNNs is that they require the DNN model to be trained in order to learn the characteristic features and patterns in the underlying data. During training, taking supervised learning as an example, the weights of the DNN are iteratively updated along the opposite direction of gradients in the loss function w.r.t. the training data and corresponding labels until convergence. In comparison with the testing phase, the training phase has significantly higher computational complexity and memory consumption.

Many works have been done to minimize the time required for the training phase, ranging from algorithm-level optimization (e.g. low variance Stochastic Gradient Descent (SGD) [28]) to hardware-level acceleration (e.g. Resistive Random Access Memory (ReRAM) [29]), allowing DNN training to be applied to increasingly challenging real-world problems. Recent developments in deep learning software also shift the focus away from frequency scaling and towards massive parallelization. In [30], Abadi et al. build TensorFlow, which enables distributed training by converting the neural network into an executable graph, with each device responsible for a unique subgraph. In [10], Li et al. propose parameter server, featuring many architecture-level optimizations resulting in excellent execution scalability during the training phase of neural networks.

However, these previous works focus mainly on designing around powerful, dedicated server clusters, neglecting that it is mobile devices that catch the primary source for the majority of data collected from the outside world. Thus, in this paper, we explore an important research topic that is barely touched: distributed learning on multiple mobile devices under a wireless local area network (WLAN). By utiliz-
ing such a solution, neither private data nor trained models will be accessible by the outside world, and no extra cost is needed for the establishment of cloud-based resources. Unlike previous works regarding distributed mobile computing systems for neural networks [31], we advocate a distributed mobile system that can not only test but also train a neural network. In order to train a model, it is expected that large quantities of privacy-sensitive data will be transmitted to the data center, incurring inevitable privacy problems. Moreover, this client-server paradigm requires dedicated infrastructure and strong network support, incurring additional implementation costs, and lower user experience. In section 3, the detailed methods and implementation of our distributed mobile training system will be discussed.

1.3 DNN Models for Video Detection System

Deep neural networks (DNNs) are now widely utilized in many cognitive tasks, such as automated speech translation [32], Natural Language Processing (NLP) [23], object detection [33] and classification [34], facial recognition [35], etc. In this work, we focus on the optimization of the DNN-based video detection application. Video detection is tremendously adopted in the mobile context, which aims at detecting the objects in continuous computer vision captured by the mobile cameras for the purpose such as video analytics. Video detection is one of the most computationally-demanding DNN applications because of many reasons. First, compared to the voice or text signal in NLP applications, the inputs (image) of the video detection system usually get a larger size. Second, as mentioned before, convolutional layers in video detection models are more computation-intensive than fully-connected layers in NLP models. Last, compared to object detection, video detection involves a sequence of video frames to be detected in real-time.

Many attempts have been made to accelerate the video detection system using DNNs. The works could be concluded to model compression and system optimiza-
tion. For the method of model compression, it designs a compact model structure by minimizing model parameters from the over-parameterized DNNs. Model compression methods can effectively decrease the inference time of DNNs by reducing the number of network weights. Structure search [36, 37, 38], weight pruning [39], low rank approximation [40], and quantization [41] are three representative methods. Another solution is system optimization, which accelerates the DNN-based system through system-level or application-oriented methods. In [9], Hauswald et al. divide the computation pipeline of computer vision tasks into a pipeline and examine the tradeoff between different workload partition schemes on local mobile devices and outside server; In [42], Zhu et al. utilize the motion data from Image Signal Processor (ISP) to relax the number of expensive CNN inferences during video detection. In most cases, system optimization can reach a better performance than model compression method because it focuses on the optimization of a certain kind of application such as NLP or video detection. In this work, we propose three application-oriented optimizations for the DNN-based video detection system, the details of which will be discussed in Section 4.

1.4 DNN Models for Neural Machine Translation

Transformer comes to be an emerging network structure in the Neural Machine Translation (NMT) tasks as a promising paradigm for sequence modeling [3]. Empowered by the stacking of attention and large feed forward layers, Transformer successfully achieves state-of-the-art performance in many NMT benchmarks as well as related language tasks [23]. Figure 1.2 depicts the architecture of Transformer model in [3]. Transformer is a sequence-to-sequence model which are composed of decoder (left part of Figure 1.2) and encoder (right part of Figure 1.2). Both encoder and decoder are composed of several identical Transformer blocks (e.g. 6 blocks in [3]). The input and output of each block in Transformer are of the same dimensions ($d_{model}$).
which is the same as the embedding dimension ($d_{\text{embed}}$) of the source/target language dictionary. In each Transformer block, there exist two main components, which are Multi-Head Attention (MHA) and Feed Forward Network (FFN). In [3], translation tasks are adopted in the experiment on NMT English to German translation task.

In recent years, the performance of many pre-trained attention-based models has been further enhanced by larger datasets as well as larger model sizes [23, 43, 44]. However, the ever-growing model size of the transformer models introduces tremendous memory consumption and computational cost, making the models hard to be deployed onto real-time embedded systems [45]. Running the transformer models on remote servers may potentially cause latency, privacy, and security problems [46, 47]. In order to solve these problems, the execution of a transformer on mobile platforms
becomes an important research field [4, 48, 49, 50]. In order to make the family of Transformer models more execution-friendly on resource-constrained platforms (e.g. mobile devices), many efforts have been done to compress the Transformer model size.

Transfer Learning method aims at using the knowledge from a pre-trained large model to guide the training of a smaller model structure. In [51], Sanh et al. initially remove 1 out of every 2 layers of the teacher model to form the student model. Then, the student model is trained based on the KL-divergence of the logits from the student and teacher model. In [52], Sun et al. transfer the knowledge of the Transformer model using both the MSE of feature maps and the KL divergence of per-head self-attention distribution between the student and teacher models as the loss.

Efficient Transformer Variants method tries to substitute the original costly Transformer modules with more efficient operators. For example, in [53], Zhang et al. introduce average layer and gating layer to summarize history attention via a cumulative average operation over previous positions to replace the original self-attention scheme. In [54], Wu et al. introduce Long-Short Range Attention, which designates different heads for local heads modeling and a long-distance relationship to broaden the functionalities of attention structure in Transformer.

Model Pruning has been proven as an effective way to reduce the resource needs of transformer model execution. Current model compression techniques for Transformer models mainly fall into three categories – Model Pruning, Transfer Learning, and Efficient Transformer Variants.

Model Pruning method fine-tunes the original pre-trained model to force the weights [55, 39, 56] or activations [57, 58, 59] to be zeros as many as possible. A transformer model may be pruned at four different pruning granularity levels, which are layer-wise pruning [60], head-wise pruning [1, 61], line-wise pruning [2]
and element-wise pruning [62].

Model Pruning is a straight-forward technique among the model acceleration methods. Compared with transfer learning and efficient Transformer variants, model pruning neither requires extra human effort in model structure design nor needs to train the model from scratch. Therefore, in this paper, we propose TPrune, which analyzes and prunes Transformer so that it could be better accommodated in mobile devices. Section 5 will present the details of TPrune, an analyzing and pruning pipeline for Transformer models.
Distributed Testing on Mobile Devices

2.1 Preliminary

2.1.1 Testing on Mobile Devices

With the increasing research of efficient testing of neural networks, the industry gets involved in machine learning development for mobile and cloud platforms. Some widely-used deep learning toolkits includes PyTorch [63], MxNet [64], Caffe [65], and TensorFlow [30]. In order to reach better DNN execution, we propose MoDNN in this section, a distributed DNN testing system for mobile devices.

![Diagram](image)

**Figure 2.1**: GEMM steps: (a) Patch to column step. (b) Matrix multiplication step.
2.1.2 Traditional Convolutional Neural Networks

We first characterize the computing overhead of convolutional layers in Neural Networks (NNs). Modern deep learning libraries leverage GEneral Matrix to Matrix Multiplication (GEMM) to realize convolution operations. GEMM achieves consistent memory access patterns and instruction-level optimization. As shown in Fig. 2.1(a) and Fig. 2.1(b), GEMM consists of two steps. The patch to column step is shown in Fig. 2.1(a) and the matrix multiplication step us shown in Fig. 2.1(b). Patch to column step reshapes the input feature map into columns while matrix multiplication steps multiply the reshaped matrix with the filter matrix. When performed on mobile CPU, the patch-to-column step consumes 71.6% of the computing time while the matrix multiplication step consumes the remaining 28.4%. A conclusion could be made that the long execution time of DNN on mobile devices is because of both the memory intensity of patch-to-column steps and computing intensity of matrix-to-matrix multiplication steps.

2.1.3 Model Pruning

Besides efficient model structure, another hot area to speed up model execution of neural network is model pruning, especially for convolutional layers. As mentioned before, the convolution operation is essentially a matrix-to-matrix multiplication. The left matrix is the input feature map while the right one stands for the convolution kernel, also known as model parameters. Therefore, to prune a neural network model, we can either try to decrease the size of the model weights or the size of the input feature map.

Structured Weight Pruning

Structured weight pruning means the model weights are discarded in the unit of groups [39]. The group granularity is manually defined. The weights of convolutional
layers can be represented as a collection of 4-D tensors: \( T_{[l]} \in \mathbb{R}^{F_l \times C_l \times H_l \times W_l} \), where \( F_l, C_l, H_l, \) and \( W_l \) denote the filter number, channel size, spatial height, and spatial width of the \( l \)-th layer in the model of \( L \) total layers. To learn the structured sparsity of the DNN model, the error function is defined as:

\[
E(T) = \sum_{i=1}^{n} V(f(x_i), y_i) + \lambda R(T) + \sum_{l=1}^{L} \lambda_g R_g(T_{[l]}),
\]

(2.1)

where function \( V(\cdot) \) generates the loss induced by the training data corresponding to their labels and the term \( \lambda R(T) \) represents the original regularization method (e.g., L2-norm) utilized to avoid the overfitting problem. \( R_g(\cdot) \) is the additional structured sparsity regularization. More specifically, \( \lambda_g R_g(T_{[l]}) = \lambda_g \sum_{g=1}^{G} \| T_{[l]}^g \|_g \), meaning the sum of the \( G \) group lasso regularization terms in layer \( l \) with their corresponding weight decay \( \lambda_g \). Mind that \( T_{[l]}^g \) represents a collection of partial tensors in group \( g \) of layer \( l \). Here we denote \( \| \cdot \|_g \) as the group lasso operator, which is defined as: \( \| T_{[l]}^g \|_g = \sqrt{\sum_{i=1}^{\text{Size}(T_{[l]}^g)} t_i^g} \), in which \( \text{Size}(T_{[l]}^g) \) is the number of tensors in \( T_{[l]}^g \) and \( t_i^g \in T_{[l]}^g \). Compared to weight pruning, feature map pruning is less exploited. However, there is a growing trend about such research field because of the extensive research on weight pruning. The concept of channel-wise feature map pruning is the same as the filter-wise pruning for model weights. The only difference lies in that channel-wise feature map pruning prunes different channels based on different inputs because different inputs contain different features. Thus, the extracted features are dynamically decided.

**Element-wise Weight Pruning**

On the other hand, weight can also be pruned element-wise [66]. In [66], the author describes a method to reduce the storage and computation required by neural networks by an order of magnitude without affecting their accuracy by learning only the
important connections.[66] prunes redundant connections using a three-step method. First, a network is trained to learn which connections are important. Next, the unimportant connections are pruned. Finally, the network is retrained to fine-tune the weights of the remaining connections. On the ImageNet dataset, a factor of $9\times$ of parameters of AlexNet is reduced, from 61 million to 6.7 million, without incurring accuracy loss. The feedback of element-wise weight pruning is that it will result in a sparse matrix instead of a smaller dense matrix. Although the theoretical Floating-point Operations (FLOP) is largely decreased, no much practical computation time is saved except when the matrix is sparse enough to get a computation benefit.

2.2 System Framework of MoDNN

Although pruning effectively accelerates model execution. There still exist some times where we want to deploy a large DNN on mobile devices with high performance. In order to deal with such a case, we propose MoDNN, which executes a single DNN on multiple mobile devices under the local network. We call our system MoDNN. Fig. 5.2 presents an overview of MoDNN which includes three main components: 1) A local distributed network cluster formed by Group Owner (GO) and several worker nodes; 2) A model processor that partitions the DNN model onto the worker nodes; and 3) A module that performs data delivery and identification services of the DNN.

We note that the computing cost of Convolutional Layers (CLs) is primarily dependent on its input size. Hence, we introduce a Biased One-Dimensional Partition (BODP) scheme to partition the CLs. On the contrary, the memory usage of Fully-connected Layers (FLs) is mainly decided by the number of weights in the layer. As a result, a weight partition scheme that consists of Modified Spectral Co-Clustering (MSCC) and Fine-Grain Cross Partition (FGCP) is introduced specifically for sparse FLs. It is worth noting that here the DNN model partition only needs to be performed once in the application once the DNN is trained. Thus, the partition cost
can be amortized over the execution of the system as long as the trained DNN keeps the same.

2.2.1 Definition of Terminologies and Variables

We define the terminologies and variables that are referred to in the following sections here:

- **Total Worker Nodes** ($k$): Total number of the available worker nodes within the computing cluster;

- **Workload** ($W_{[i]}$): The workload assigned to node $i$;

- **Estimated Time** ($ET_{[i]}$): Estimated time for node $i$ to execute workload $W_{[i]}$ plus data delivery time;

- **Computing Ability** ($CA_{[i]}$): The normalized performance of node $i$. Computing ability is used to decide the workload partition imbalance ratio for node $i$.

- **SpMV Time** ($SPT_{[i]}(n)$): Time for node $i$ to do Sparse Matrix-Vector multiplication (SpMV) in which the matrix is represented by a linked list of size $n$;

- **GEMV Time** ($GET_{[i]}(r,c)$): Time for node $i$ to perform General Matrix-Vector multiplication (GEMV) in which the matrix is represented by $r \times c$ array;

- **Sparsity Threshold** ($Thld_{[i]}(r,c)$): Sparsity threshold of node $i$ that achieves equivalent computing time of the $r \times c$ matrix using SpMV and GEMV;

- **Data Delivery Time**: Data delivery time denotes the total time consumption for the data being transmitted between nodes. Data delivery time includes two
parts: wakeup time and transmission time. Wakeup time denotes the time for the head of the data traveling from the sender to the receiver and transmission time denotes the amount of time for the receiver receiving from the first bit to the last bit of the data.

2.2.2 Network Establishment and Setup

Network Topology of MoDNN

In MoDNN, each worker node is assigned a subset of the layer inputs. After the execution of a certain layer, the outputs are reduced back to the GO, which generates the inputs for the new layer in the next iteration.

In order to form a computing cluster, GO first enables its WiFi module to act as an Access Point (AP) that is prepared for responding to potential worker nodes. In the meantime, the available worker nodes with extra computing resources are searching for the GO in the same local network.

Connection and Registration

In our design, after getting permission from the user, a mobile device will be permanently trusted by the GO and automatically connected to the group when it is within the reachable WiFi range. Once connected, in addition to the device IP address, performance-related metadata are also sent to the GO for later utilization in partition schemes. The meta data includes the previously defined variables and functions like $CA_{i[j]}$, $SPT_{i[j]}(n)$, $GET_{i[j]}(r,c)$, $Thld_{i[j]}(r,c)$, etc. The motivations to define and generate $SPT_{i[j]}(n)$, $GET_{i[j]}(r,c)$ and $Thld_{i[j]}(r,c)$ will be discussed in the following methods.

2.2.3 Input Partition for CL

Traditional partition method of CLs on other platforms maintains a symmetry pattern for the input feature map of CLs. As shown in Fig. 2.2(a), in [67], Coates et
arranged a GPU cluster into 2D-grids and partitioned the input neurons along the two-dimensional space evenly. However, such a 2D partition is not satisfactory for our MoDNN system.

Unlike in the GPU cluster, the wakeup time, rather than the transmission time, dominates the data delivery time in the transmission among mobile devices. It is because of the Opportunistic Power Save Protocol that supports the sleep mode of the clients. If a mobile device has not been active for a certain time period, it will turn off its radio modules automatically [68]. Turning on the radio modules and establish the transmission channel takes a time period significantly longer than the data transmission time.

In MoDNN, BODP is proposed to partition the input feature map along the longer edge of the input matrix according to the computing abilities \( CA_i \) of an individual node, as illustrated in Fig. 2.2(b). There are two sets of neurons in the figures: input neurons and output neurons of a CL. Taking node 3 as an example: the input feature map of node 3 overlaps all the other three nodes in the 2D partition in Fig. 2.2(a). However, the feature map of node 3 in Fig. 2.2(b) only overlaps with that of node 2. Note that only the overlapped parts of the feature map need to be transmitted among devices during DNN execution.
Because the wakeup time in MoDNN is greatly impacted by the number of the established transmission channels, reducing the number of the neighbor nodes from 4 (in the conventional 2D partition) to 2 (in BODP) will decrease the high propagation time effectively.

2.2.4 Weight Partition for Sparse FLs

For FLs, if executed and partitioned in a conventional way, not only exhaust unnecessary computing cost and memory consumption but also cost redundant transmission size in a local distributed mobile computing system. The partition of FLs is straightforward, where the weight matrix is divided by columns. Beyond FLs, we focus on the optimization space for Sparse FLs. Because of the comparatively short execution time of FLs, mobile devices will keep the wireless radio in an active state and the transmission time dominates the data delivery time. The object of the proposed partition scheme is to reduce the size of the data to be transmitted for reducing the transmission time between mobile devices.

Hybrid Matrix Representation in MoDNN

There are two ways to compute matrix-vector multiplication for FLs: General Matrix-Vector multiplication (GEMV) and Sparse Matrix-Vector multiplication (SpMV). GEMV is used to compute a dense matrix that often uses arrays as the data struc-
ture. SpMV is adopted for sparse matrix that can be efficiently stored as linked-list, as illustrated in Fig 2.3. In MoDNN, we assign different operations to different sub-matrix based on their sparsity level. The sparsity level is defined as:

$$Thld_{r,c} = \frac{SPT^{-1}(GET_{r,c})}{r \times c}.$$  \hspace{1cm} (2.2)

Here $SPT_{[i]}(n)$ and $GET_{[i]}(r, c)$ are the time spent on the computation of the matrix-vector multiplication using SpMV and GEMV, respectively. They can be obtained from real measurements on mobile devices via a linear regression method. The selection of the appropriate data representation is based on the comparison between target matrix sparsity and pre-defined threshold $Thld_{r,c}$. When the sparsity of the matrix is larger than $Thld_{r,c}$, SpMV will be used for the computation; otherwise, GEMV will be applied.

**Modified Spectral Co-Clustering (MSCC)**

Generally, GEMV is more efficient per matrix element than SpMV due to its higher computing parallelism. Hence, it will be beneficial to partition the weight matrices onto the worker nodes in a dense structure. In the weight partition scheme of FLs
in MoDNN, a clustering algorithm is leveraged to group the nonzero weights into several clusters and minimize the number of the nonzero weights outside the clusters. Considering the weight matrix as an undirected graph, generating \( k \) clusters with minimal connections between them is an NP-hard problem [69]. In MoDNN, we use spectral clustering to find the solution heuristically.

Targeting at minimizing between-cluster similarities [70], Spectral clustering technique is widely used in graph partition problems. In MoDNN, the sparse FLs are treated as undirected graphs where the graph vertices represent the input and output neurons and the edges represent the network weights. Hence, we re-define the similarity in spectral clustering technique as the number of between-clusters connections. However, traditional spectral clustering technique works only on a matrix with the same row and column size, which greatly limits its generality of DNN type. Therefore, spectral co-clustering algorithm is introduced to address this drawback by normalizing the original connection matrix \( A \) to \( A_{\text{norm}} \) and then performing Singular Value Decomposition (SVD) on \( A_{\text{norm}} \) [71]. Here the elements of weight matrix \( A \) are binary, where '1' represents there exists a connection between two neurons and '0' otherwise. This algorithm converts \( r \) rows and \( c \) columns of the original matrix \( A \) to a matrix \( Z \) of \( r + c \) rows using the results generated in obtaining \( A_{\text{norm}} \). Each column of \( Z \) is an eigenvector of \( A \) so that we can cluster matrix together based on the rows of \( Z \). Then, we apply appropriate data structure to each cluster based on their sparsity for computing time reduction. We name this clustering procedure as modified spectral co-clustering (MSCC). A rationale of the proposed MSCC is depicted in Fig. 2.4(a) and 2.4(b). After applying MSCC, \( k \) dense clusters are generated and the corresponding input neurons are transmitted to the assigned worker nodes for parallel executions.
Fine-Grain Cross Partition (FGCP)

Spectral co-clustering focuses on reducing the external connectivity between the clusters without considering the internal cluster density. To solve this problem, we propose FGCP to partition the remaining outliers in the weight matrix after MSCC to balance the workloads between the GO and the worker nodes. The basic idea here is to identify the sets of weights with the minimal number of nonzero elements and keep them computed on the GO rather than sending to the worker nodes to avoid the high cost introduced by the long data delivery time.

For the sparse outlier matrix shown in Fig. 2.4(c), for example, because the number of the columns is smaller than rows, FGCP initially assigns the elements on the same rows where the cluster $C_{ri}$ (obtained in MSCC) resides to node $i$. Then, FGCP iteratively finds the worker node $i$ with the maximum $ET_{ri}$ and offloads the initially assigned weights on the same column in the outlier matrix with the minimal number of non-zero elements from the worker node $i$ to the GO. In addition, FGCP needs to consider the discrepancy of execution time between the GO and the worker nodes during the offloading process, especially the data delivery time on the network between the worker node $x$ and the GO, which can be conceptually formulated by:

$$\text{Initial } ET(x) = \frac{\left(\sum_{i=0}^{r} C_{\text{column}[i]} + \sum_{i=x}^{k} C_{\text{row}[i]}\right)}{\text{TPT}}, \quad (2.3)$$

where $TPT$ is the mobile network throughput; $\sum_{i=0}^{r} C_{\text{column}[i]}$ and $\sum_{i=x}^{k} C_{\text{row}[i]}$ describe the total non-overlapping data size of the input and output neurons to be transmitted during the execution. Obviously, higher sparsity of the outlier matrix leads to more elements to be offloaded to the GO and thus balance the workload.
2.2.5 Processing Flow of DNN

Fig. 2.5 and Fig. 2.6 illustrates the proposed pre-processing pipeline of CLs and FLs in MoDNN and how the two parts are integrated. Given an original trained DNN model, the model processor extracts each layer and identify its type. If a CL is detected, the layer input will be partitioned by BODP into small pieces, which are then combined with the subsequent non-overlapping layer structures e.g., ReLu layers, pooling layers, normalization layers, etc. for computation. If a sparse fully-connected layer is detected, MSCC and FGCP will be applied in sequence to assign the workloads to the worker nodes in clusters and the workloads for outliers, respectively, in order to achieve the minimum total execution time.
2.3 Experiments

2.3.1 Environment Setup and Test bench Selection

The implementation of MoDNN is based on MXNet, which is a deep learning framework developed by the Distributed Machine Learning Community (DMLC) team for the desktop platform using C++ [64]. We modify and recompile the MXNet libraries
so that it can support Android systems with ARM architecture with JAVA Native Interface (JNI) [64]. We adopt a pre-trained DNN model called VGG-16 [8], as the target DNN model to be partitioned in our experiments. The model is trained on ImageNet dataset [72]. VGG is a famous Convolutional Neural Network (CNN) model that includes all mainstream layer types so that the significance of each component of MoDNN can be distinctly evaluated. In our experiments, VGG are executed locally or distributed to different numbers of worker nodes by MoDNN; the adopted mobile
devices are LG Nexus 5 running Android 4.4.2 with a 2.28 GHz processor and 2GB RAM. The experiment setup is depicted in Fig. 2.7.

Fig. 2.8 presents the results of $SPT_{i}[n]$ and $GET_{i}[r,c]$ on Nexus5. Linked list and array structures are used in characterizing $SPT_{i}[n]$ and $GET_{i}[r,c]$, respectively. Here x-axis denotes the amount of computations, i.e., $n$ non-zeros for $SPT_{i}[n]$ and $r \times c$ matrix for $GET_{i}[r,c]$, respectively. The results show that the calculation time of the worker nodes is proportional to the calculation number. For the same workload, SpMV is much slower than GEMV. Hence, we set $Thld_{i}[r,c]$ to 15.8% in our scheme. The measured average WLAN wakeup time and transmission throughput are 54.7ms and 43.8Mbps, respectively.

2.3.2 Data Delivery Time Evaluation of BODP

Fig. 2.9 shows the computing times of 13 CLs in VGG-16 during the testing phase, without considering the data delivery time. The results of running locally and on 2, 3, and 4 worker nodes in MoDNN are depicted. For comparison purpose, the results of using conventional 2D-grids partition scheme for 4 worker nodes is also included in this figure. When the number of worker nodes increases, the execution time of each CL keeps reducing, proving the effectiveness of MoDNN in parallel computing. The results of BODP with 4 worker nodes and 2D-grids partition are very close, implying little impact of the input shapes of the CLs on the computing time. As also illustrated by the dot lines in Fig. 2.9, compared to 2D-grids partition, BODP also slightly increases the average data transmission size of each CL from 41048 bytes to 59856 bytes and hence, increases the average transmission time from 7.15ms to 10.43ms. Nonetheless, when taking into account that the total wakeup time contributes to approximately 30% of the total data delivery time in each data sharing procedure, BODP still achieves shorter total data delivery time than 2D-grids partition for 4 worker nodes as fewer transmission channels need be established.
Table 2.1: Overall evaluation of MoDNN with 2-4 worker nodes.

<table>
<thead>
<tr>
<th></th>
<th>Execution (ms)</th>
<th>Delivery (ms)</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>15809</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 Workers</td>
<td>8509</td>
<td>1819</td>
<td>1196</td>
</tr>
<tr>
<td>3 Workers</td>
<td>6884</td>
<td>2563</td>
<td>2257</td>
</tr>
<tr>
<td>4 Workers</td>
<td>5208</td>
<td>2567</td>
<td>3336</td>
</tr>
<tr>
<td>4 Workers 2D-grids</td>
<td>6324</td>
<td>3073</td>
<td>2256</td>
</tr>
</tbody>
</table>

2.3.3 Transmission Size Evaluation of MSCC & FGCP

In order to evaluate the effectiveness of MSCC and FGCP on large-scale, sparse FLs, the FLs in VGG-16 are sparsified by $L_1$-norm group lasso with a predefined discarding threshold to control the sparsity [11]. We define the evaluation baseline as the one-dimensional partition that divides the weight matrix along its longer side without any overlaps between the partitioned parts. The experiments are performed on the layer sparsity varying from 70% to 96% and the results of the transmission size reduction are normalized by the baseline, as shown in Fig. 2.10.

According to the results, MSCC and FGCP effectively reduce the transmission size by at least 22.6% compared with the baseline implementation. In most cases, the transmission size reduction ratio keeps increasing with the layer sparsity, e.g., reaches as high as 49.3%, 69.2%, and 69% for FC_6, FC_7, and FC_8, respectively at different numbers of worker nodes. One exception occurs in the FC_8 with 2 worker nodes, which shows a decrease in the transmission size reduction ratio with when the layer sparsity increases. It is because of the residual unbalance in the clustering due to the limited solution space of the small-scale layers (e.g. $1000 \times 4096$ for FC_8). Nonetheless, following the increase of the number of the worker nodes, the effectiveness of MSCC and FGCP also increases, demonstrating a good design scalability.
2.3.4 **Overall Evaluation of MoDNN**

In table 2.1, we summarize the overall execution time to compute the whole *VGG*-16 model in MoDNN over different numbers of mobile devices. Following the increase of the number of the worker nodes, the overall execution time reduces significantly, demonstrating excellent computing parallelism: the computation time improves by 2.17-4.28× with 2 to 4 worker nodes. We also summarize the data delivery time and the data transmission size of different scenarios in table 2.1, indicating the extra cost introduced by the distributed computing mechanism of MoDNN. MoDNN outperforms the conventional 2D-grids partition scheme by substantially reducing the data delivery time though the data transmission size is slightly increased.

2.4 **Conclusion**

In this section, we introduce MoDNN, a local distributed mobile computing system to enable the parallel computation of DNN on mobile platforms. As convolutional layers and fully-connected layers are identified as the major DNN components that contribute to the total execution time, several advanced partition schemes, i.e., BODP, MSCC, and FGCP are proposed to well balance the workloads of each worker nodes and minimize the data delivery time. Experiments show that MoDNN can achieve better than linear performance speedup on DNN computations, demonstrating the great potential of mobile platforms in DNN-related applications.
3

Distributed Training on Mobile Devices

Unlike previous works regarding distributed mobile computing systems for neural networks [73, 31], in this section, we advocate a distributed mobile system that can both test and train a neural network in a distributed fashion. We term our system AdaLearner, because we need to adaptively tailor DNN hyper-parameters during the training phase for the distributed mobile scenario from the perspectives of system architecture and training configuration scheduling.

Concretely, our contributions include:

- We propose AdaLearner—an adaptive distributed mobile system for neural networks training, consisting of two core distributed training architectures;

- We design an adaptive scheduler for AdaLearner in Section 3.5, which adapts the training configuration (e.g. number of worker nodes (WNs), batch size, transmission data size) to heterogeneous mobile resources and network circumstances.

- We adopt and improve 1-bit quantization in AdaLearner, which can increase the system scalability by fiercely compressing the transmission data size to only
1-bit with little or no accuracy drop;

- We implement and experiment AdaLearner on ARM-based smartphones by modifying and combining the state-of-the-art libraries for neural network training.

We evaluate AdaLearner on three neural network models that are trained from two well-known image classification datasets (MNIST, CIFAR-10). Take the experimental results of LeNet on MNIST as an example, compared with local training, AdaLearner accelerates the training by $1.75\times-3.37\times$, when increasing the WNs from 2 to 8.

3.1 Preliminary

3.1.1 Backpropagation with SGD

When training a neural network on $n$ examples: $(x_1, y_1), \ldots, (x_n, y_n)$, we often target at optimizing the following problem:

$$\text{Min}\{F(w)\}, \quad F(w) = \frac{1}{n} \sum_{i=1}^{n} L(w; x_i, y_i), \quad (3.1)$$

where $L(\cdot)$ denotes the loss function and $w$ denotes the weights. For instance, if we use a square loss in the training phase, then the loss function will be: $L(w) = (w^T x_i - y_i)^2$. In order to update the weights for the minimization of $F(w)$, a partial derivative with respect to each weight in $w$ is calculated by backpropagating through all the layers in the neural network. The update procedure from iteration $l$ to $l + 1$ can be formulated as:

$$w_{l+1} = w_l - \frac{\partial F(w)}{\partial w}, \quad (3.2)$$

where $\eta$ is the learning rate. However, in each iteration of gradient descent, backpropagation is applied to all the $n$ training examples, encountering expensive computation
cost. Stochastic Gradient Descent (SGD) is thus introduced where each iteration only consists a mini-batch of the whole training examples. Hence, the $F(w)$ in Eq. 3.2 can be approximated as:

$$F'(w) = \frac{1}{t} \sum_{i=1}^{t} L(w; x_i, y_i),$$

(3.3)

where $t$ is the minibatch size for a single update.

3.1.2 Parallel Schemes for Distributed NN Training

There are two common ways to train a model distributedly in a centralized system, which are model parallelism and data parallelism. From the perspective of communication network topology, there are centralized topology and decentralized topology.

Model parallelism

In model parallelism, all the nodes are provided with the same input data while each node holds only a part of the full model [74]. As exemplified in Fig 3.1(a), each node contains a single layer of the model and the nodes need to communicate
with each other as long as there is a connection between them. Model parallelism is generally used when training a large-scale model so that the model parameters can be scattered on each node with less node-wise memory consumption. However, this scheme requires frequent transmission due to its smaller computation granularity (need to communicate layer-wise).

**Data parallelism**

In data parallelism, as illustrated in Fig 3.1(b), each node holds a model replica but is fed with a different subset of total dataset (minibatch) [10]. After finishing the minibatch round of all the worker nodes, there will be a node in charge of gathering all the computation results and send them back. The model will be updated on that node. Whether there is an independent node acting as the scheduler varies in different training scenario [74][75]. In most cases, data parallelism enjoys a looser communication pattern and simpler partition complexity than model parallelism.

**Network Topology for Distributed Training**

Model parallelism and data parallelism are main-stream distributed neural networks training paradigms [76] [31] [4]. Both of these two methods can be accommodated to different distributed system network topologies. Fig 3.2(a) illustrates centralized
topology. In centralized topology, there exists a central controller called parameter server, taking care of the whole workload partition and scheduling. All the nodes beside the central controller are worker nodes. They receive the command from the central controller and return the computation results back. Fig 3.2(b) gives an example of decentralized topology, which is more flexible in connection. Different from the centralized topology, the nodes in decentralized topology play the same role among the whole system. Each node of the system only communicates with its neighbors. The benefit of decentralized topology is that it enables better scalability and flexibility of the system while the drawback is that it requires that all the nodes are secure and trustable.

3.1.3 Quantization Methods

Scalability is a critical optimization target of distributed systems, where the communication between nodes is always the main bottleneck that restricts the system scalability. In a distributed system, the communication costs grow quadratically with the increase of the number of the nodes grows. In order to decrease the communication cost during neural network training, quantization methods are widely used.

Quantization is a very popular method, which can be applied to both testing and training procedures. Here, I introduce two quantization methods, namely 1-bit quantization and TernGrad quantization.

1-bit Quantization

Among many data compression technologies targeting at efficient neural network training [75][77]. 1-bit quantization has two important features: (1) The compression rate is the highest among all the existing technologies. Each number to be transmitted is quantized to 1 bit, the compression ratio of which is 1/32 because the
original data are represented as floating-point numbers, which are 32 bits in length; 
(2) 1-bit quantization enables less extra computation cost because of its simple quantiza-
tion function. In 1-bit quantization, 0 is set as the quantization threshold. All 
the number larger than or equals 0 is set as 1 while all the negative numbers are set 
as 0.

In order to make up of the loss introduced by 1-bit quantization, the quantization 
error of last minibatch will be added to the current one [75]. Assume the gradients 
calculated by k-th minibatch is $G[k]$, the quantization can be expressed as:

$$G_{quantized}[k] = Q(G[k] + \Delta[k-1]),$$

where $Q(\cdot)$ stands for the quantization function. $\Delta[k-1]$ represents the quantization 
error of $(k-1)$ minibatch and $G_{quantized}[k]$ is the quantized 1-bit results to be transmit-
ted. After the quantization procedure in Eq 3.9, a new quantization error feedback 
is calculated for next minibatch.

$$\Delta[k] = G[k] - Q^{-1}(G_{quantized}[k]).$$

With the effect of error feedback, the quantization errors are accumulated and 
expressed in the subsequent iterations of minibatch. Therefore, it can losslessly 
compress the gradients between nodes to 1 bit with little or no accuracy drop and 
the experimental results will be presented in the experiment section.

*TernGrad Quantization*

Although 1-bit quantization reduces the transmission amount by the highest factor 
and requires tiny computing overhead, it inevitably exists little accuracy degradation. 
Therefore, we adopted another quantization technology, namely, TernGrad [78], to 
tradeoff computing resources and quantization bits with a higher neural network 
model accuracy.
Different from 1-bit quantization, TernGrad maps the original gradients to ternary values: $-1$, $0$, and $+1$. The quantization function for training iteration $k$ can be expressed as:

$$G_{\text{quantized}[k]} = \max(\text{abs}(G^i_{[k]})) \cdot \text{sign}(G_{[k]}) \circ B_{[k]}.$$  \hspace{1cm} (3.6)

The final quantized gradients are the product of three parts, which represent scalar, sign, and probability respectively. As shown in Eq 3.6, the first part $\max(\text{abs}(G_{[k]}))$ is the scalar of all the layer-wise gradients, which is defined as the maximum number among all the absolute values of each layer in $G_{[k]}$. The layer-wise scalar introduces negligible transmission overhead while brings tremendous benefit for convergence speed. For the purpose of distinguishing the gradient descent direction, the second part, $\text{sign}(G_{[k]})$, denotes the sign of each gradient in $G_{[k]}$. The third parts, $B_{[k]}$, is either 1 or 0 for each gradient in the neural network model. The probability of the $i$-th element $G^i_{[k]}$ to be equivalent to 1 follows the Bernoulli distribution:

$$Pr(B^i_{[k]} = 1) = \frac{G^i_{[k]}}{\max(\text{abs}(G_{[k]}))}.$$  \hspace{1cm} (3.7)

Hence, the probability of the $i$-th element $C^i_{[k]}$ to be equivalent to 0 is $1 - Pr(B^i_{[k]} = 1)$. Mind that $\text{sign}(G_{[k]})$ and $P_{[k]}$ apply to individual gradient, meaning that the results are vectors. So, the operator $\circ$ is the Hadamard product.

### 3.2 Methods

In parallel training, the total execution time consumed in the training phase with $N$ WNs can be expressed as:

$$T_{\text{total}} = \max\left(\frac{W_{\text{comp}[i]}}{C_{[i]}}\right) + \frac{W_{\text{trans}}}{B} \cdot N + \sum_{j=1}^{N} D_{[j]}, i = 1, 2,...N.$$  \hspace{1cm} (3.8)

Where $W_{\text{comp}}$ and $W_{\text{trans}}$ respectively denote the device-wise minibatch size and the transmission data size of node $i$ in a single training iteration; $C_{[i]}$ describes the
computing capability of node \( i \), which illustrates the minibatch size that can be completed per unit of time; \( B \) is the available WiFi bandwidth; while \( D_{[j]} \) stands for the wakeup time for WN \( j \). In AdaLearner, we set \( W_{\text{comp}} \) and \( W_{\text{trans}} \) in Eq. 3.8 as our main optimization target.

**Adaptive Configuration for Computation:** To minimize \( \max\left(\frac{W_{\text{comp}}[i]}{c_{[i]}}\right) \) in Eq. 3.8, we need to excellently balance the computing workload within each WN in Eq. 3.3. From Eq. 3.3, we can find that there is a significant solution space w.r.t. the optimal size \( t=W_{\text{comp}} \) of the minibatch. Therefore, in AdaLearner, we adaptively determine the minibatch size allocated to each WN as a function of the mobile device’s available computing resources for optimal parallelism.

Beyond this, it can be seen from Eq. 3.8 that as the number of WNs increases, the contribution of the actual training computations decreases due to their parallel nature, causing the total execution time to be dominated by the transmission overhead. In such cases, we also need to adaptively choose the optimal number of WNs.

**Adaptive Configuration for Communication:** Communication is always a big bottleneck for a distributed training system because of the tremendous amount of data to be transmitted between WNs, which are unavoidable for each update iteration. Such data includes model gradients, parameters, and etc. Traditionally, each number in the model is a full-precision, 32-bit floating-point value, which is actually unnecessarily accurate in certain situations.

In response to this, AdaLearner adopts a fierce quantization technology—1-bit quantization [75], aimed at compressing the precision of model parameters in order to significantly reduce transmission size. Besides greatly minimized transmission data size, 1-bit quantization need little extra compression overhead.
3.2.1 Software Components

Fig. 3.3 gives an overview of AdaLeaner including 3 components:

**Model Initializer:** The model initializer generates the necessary files for training a model: symbol files and parameter files. Symbol files record the structure of the layers while parameter files define connection weights. Additionally, the model initializer of AdaLeaner generates a configuration file that contains metadata such as learning rate, batch size, and etc.

**Execution Middleware:** Execution middleware is the fundamental component in AdaLeaner. This execution middleware includes both the computation and communication scheduler, which is deployed to each WN and be responsible for their identical computation and communication works in the training phase.

**Computing Cluster Generator:** When training a neural network, the mobile device held by the user is designated as the GO and all the other assisting devices are designated as WNs. The computing cluster generator runs on the GO, which first enables its Wi-Fi hotspot feature and determines the number of WNs which subsequently connect to it. The GO then collects the IP addresses of all connected WNs for future communication. Last, the GO collects metadata, which includes the computing capabilities from each of the WNs.
3.2.2 System Architecture of AdaLearner

In AdaLearner, we provide two training architectures to deal with the tradeoff between computation cost and communication cost.

Architecture for Efficient Computation

Fig. 3.4 details the distributed architecture in AdaLearner for efficient computation, which is leveraged when the communication network is less congested, all WNs first load the training data and model into memory. Then, the model is trained on a minibatch of data using forward and backward propagation. This results in the generation of model gradients, which are then transmitted back to the GO. As data is being transmitted to the GO, each WN begins to load the training data for the next minibatch. After gathering all the gradients from the WNs, GO merges the gradients, updates its local model parameters, and transmits the updated model parameters to all the WNs. Once the WNs have received the updated model parameters, a new iteration is triggered. The total training procedure here is similar to the local training scheme with high parallelism and computation efficiency.
Architecture for Efficient Communication

As mentioned, scalability is a critical optimization target for distributed systems, with communication costs always being the main bottleneck. Compared with the training in data centers, communication in mobile networks suffers greater limitations due to the limited radio channels and comparatively low transmission bandwidth. Under such a circumstance, an alternative training architecture is proposed in Fig. 3.5, which is tailored to minimize communication overhead. Different from Fig. 3.4, there are additional steps of data encoding and decoding performed by both the GO and WNs before and after every data transmission. The encoding process is utilized to compress the data which will be transmitted, thus reducing transmission overhead. The detailed compression technology utilized and the corresponding tradeoff between the transmission data size and the extra cost for encoding decoding procedures will be described in Section 3.4.

Synchronization Mechanism

Because of the hardware limitations of the radio module in mobile devices, the communication between GO and WNs is performed sequentially. Only one pair of devices can communicate at any time. To maintain high parallelism, the GO maintains a
First In First Out (FIFO) synchronization mechanism. For example, in Fig. 3.4, as WN0 is the first to transmit gradients to GO, and is followed by WN1, the GO sends back weights in the same sequence, first to WN0, then to WN1.

This FIFO scheme establishes the communication sequence by considering the execution parallelism yet neglects the fault tolerance of the system where there exists potential disorder during message passing. Hence, the communication establishment in our system is set as non-blocking. In such a case, it will neither pause nor fail when any worker loses its connection to the GO.

**Message Format**

The message format utilized in AdaLearner is detailed in Fig. 3.6. The head of the messages in AdaLearner is the ID of the sender and the receiver of the message. The following number of bytes contain the metadata such as the learning rate for the current minibatch. If the data is compressed, the compression parameter for decoding is also included. The last part is the main body of the message, containing the gradients or the weights of the neural network.

### 3.3 Computation Cost in Adaptive Scheduler

In Eq 3.8 and Eq 3.3, we assume the computing time is linearly related to minibatch size, which is consistent with our experimental results. Fig 3.7 shows the time consumption for the forward and backward procedures with increasing minibatch size on Nexus 5X. For LeNet and ConvNet, the time consumption is proportional to the
minibatch size. An exception occurs on MLP, where the computation efficiency keeps increasing with the increase of minibatch size. It is because that MLP is a tiny neural network with low computation cost and memory usage. When the minibatch size is small, MLP does not utilize the maximum resources (e.g., memory bandwidth, CPU frequency) in the mobile device and thus showing a lower efficiency. However, when the minibatch size reaches around 50, it, too, begins to reflect the expected linear relationship. For example, in Fig 3.7, MLP only costs $200 \, ms$ for the minibatch size of 150 while LeNet and ConvNet consume approximately 10 seconds with the same batch size. Hence, when the minibatch size is small, MLP does not utilize the maximum resources and thus showing a lower efficiency.

Without loss of generality, we utilize a lookup table to record the device-wise relationship between minibatch size and their corresponding time consumption on heterogeneous mobile devices. The table in Fig. 3.8 presents an example of such a lookup table containing computing capability metadata from three WNs. For example, in order to train a model with a total minibatch size of 75, our adaptive scheduler searches the lookup table for WN combinations for each number of WN, prioritize devices with higher computing capability. First, our adaptive scheduler searches the worker node with the best computation capability, which is WN3 in Fig. 3.8 and the corresponding estimation time is $400 \, ms$. Secondly, it checks the second line of Fig. 3.8, where the total minibatch size of WN2 and WN3 is 75. In this scenario, the total computation time is $300 \, ms$. Last, the first line in Fig. 3.8 will be searched and the corresponding result is $200 \, ms$. In this way, AdaLearner can quickly estimate that $400 \, ms$, $300 \, ms$, $200 \, ms$ is needed for one, two, and three WNs, respectively.
Figure 3.7: Time consumption with different minibatch sizes.

<table>
<thead>
<tr>
<th>Minibatch Size</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Time Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>25</td>
<td>30</td>
<td>200 ms</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>35</td>
<td>40</td>
<td>300 ms</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>55</td>
<td>75</td>
<td>400 ms</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.8: Lookup table for minibatch size and computing time.

3.4 Communication Cost in Adaptive Scheduler

Transmission efficiency is critical for mobile networks scalability. Among many data compression technologies targeting at efficient neural network training [75][77]. Using the lookup table, we only estimate the total execution time on different WNs, neglecting the cumulative effect of transmission time to and from each WN. To combat this, we introduce an alternative training technology in AdaLearner to allow for compression of the data associated with each update. We adopt the concept of 1-bit quantization in our communication efficient architecture in Fig. 3.5. The choice is based on two reasons. First, the compression rate is the highest among all the existing technologies. Each number to be transmitted is quantized to 1 bit, the compression ratio of which is 1/32 because the original data are represented as 32 bits floating-point numbers. Second, 1-bit quantization requires almost no extra computation cost because of its simple quantization method.
3.4.1 1-bit Quantization with Error Feedback

As mentioned, among many data compression technologies targeting at efficient neural network training \[75\][77], 1-bit quantization achieves the highest compression rate. Concretely, in 1-bit quantization, we set 0 as the quantization threshold. All the number larger than or equals 0 is set as 1 while all the negative numbers are set as 0. In order to make up of the loss introduced by 1-bit quantization, the quantization error of last minibatch will be added to the current one \[75\]. Assume the gradients calculated by \(k\)-th minibatch is \(G[k]\), the quantization can be expressed as:

\[
G_{\text{quantized}}[k] = Q(G[k] + \Delta[k-1]), \tag{3.9}
\]

where \(Q(\cdot)\) stands for the quantization function. \(\Delta[k-1]\) represents the quantization error of \((k-1)\) minibatch and \(G_{\text{quantized}}[k]\) is the quantized 1-bit results to be transmitted. After the quantization procedure in Eq 3.9, a new quantization error feedback is calculated for next minibatch:

\[
\Delta[k] = G[k] - Q^{-1}(G_{\text{quantized}}[k]). \tag{3.10}
\]

With the effect of error feedback, the quantization errors are accumulated and expressed in the subsequent iterations of minibatch. Therefore, we can losslessly compress the gradients between nodes to 1 bit with little or no accuracy drop and the experimental results will be presented in the experiment section.
3.4.2 Gradients Aggregating Scheme

Conventionally, 1-bit SGD is realized in GPUs cluster without central scheduler and each GPU in \( N \) GPUs is responsible for aggregating a \( 1/N \) subset of the model parameters [75]. Such training topology is not suitable in our distributed mobile system. Different from conventional realization, AdaLearner incorporates 1-bit SGD to the training architecture in Fig. 3.5. After deriving the gradients on each WN, the gradients experience the quantization procedure in Eq 3.9 and then transmit them to GO. GO gathers all the gradients in 1-bit format and sums them up in the unquantized form. After post-processing the gradients in GO, a second quantization is applied before transmitted back to each WN. Each WN will then decode the received 1-bit gradients and updates the model parameters locally. The quantization and unquantization modules are embedded in the encoder and decoder in Fig. 3.5. In order to conduct unquantization procedure in GO, two numbers are required to be transmitted along with the 1-bit gradients for each column of the original weight matrix, which represents the multiplication scalar for positive and negative gradients. They are calculated by averaging all the original positive and negative gradients. Mind that original 1-bit quantization is utilized in Recurrent Neural Networks (RNNs), which only contains fully-connected layers. However, AdaLearner targets at all the neural network structures, including convolutional layers, for which we innovatively calculate the positive and negative scalar for each filter.

3.4.3 Computation / Communication Tradeoff

Such a gradients aggregation scheme introduces the tradeoff between computation cost and communication cost. The procedures with red square background in Fig. 3.9 detail the extra computation cost yielded by reduced communication cost utilizing 1-bit quantization. Before and after each communication, quantization and unquantization procedures are necessary. In order to reach high parallelism, the procedure
for calculating the new error feedback is executed in parallel without influencing the total execution time.

If we transmit the 32-bit floating-point numbers between $N$ nodes, the communication overhead can be formulated as:

$$T_{\text{comm}} = N \cdot \frac{W_{\text{trans}}}{B} + D,$$

where $W_{\text{trans}}$ means the original transmission data size under the network of bandwidth $B$. Here $D$ denotes the total wakeup time during the communication, which is assumed to be a constant in the formula. Alternatively, by applying 1-bit quantization technology, the communication can be expressed as:

$$T_{\text{1bit comm}} = 2 \cdot \frac{W_{\text{trans}}}{S_q} + (N + 1)\frac{W_{\text{trans}}}{S_{uq}} + \frac{T_{\text{comm}}}{Z} + D,$$

where $S_q$ and $S_{uq}$ respectively denote the quantization and unquantization speed (bits per unit time), which are profiled on each mobile device. In Eq 3.12, $Z$ equals 32 here because the transmission data size is 1/32 of the original transmission data size. Because the unquantization procedure of GO in Fig. 3.9 is done sequentially, the time consumption on unquantization is linearly related to the number of WNs. Thus, the decision function of whether to leverage communication efficient scheme with $N$ WNs is given by:

$$\text{Decision} = T_{\text{1bit comm}} < T_{\text{comm}} ? \text{True} : \text{False},$$

which compares between the communication costs for these two training schemes and chooses the one with less communication time. If Eq 3.13 holds, the scheduler will use communication efficient training because the extra computation overhead is smaller than saved communication overhead and vice versa.
3.5 Incorporate Adaptive Scheduler to AdaLearner

In this section, we give the comprehensive scheme, which adapts the optimal configuration (number of WNs, minibatch size, communication scheme) to the current mobile resources. The outside loop of our scheduler algorithm iteratively estimates the execution time with different WNs. For each iteration, our scheme estimates the time consumption with and without data compression and pick the one with less time estimation. The scheme keeps running until it reaches the max WNs number or the minimal total estimated time. In AdaLearner, we focus on the partition scheme with maximal parallelism of total minibatch size \(W_{\text{comp}}\), where \(W_{\text{comp}} = \sum_{i=1}^{N} W_{\text{comp}[i]}\). The total minibatch size in AdaLearner is predefined by the users. In the experiment, we test our system with the most commonly used configuration where \(W_{\text{comp}} = 128\).

3.6 Experiments

3.6.1 System Implementation and Experiments Setup

In AdaLearner, we adopt and modify three software libraries for the computation of neural network training and data communication. Targeting at efficient computation, the local training procedure is implemented based on MXNet [64], which is built under the concept of [10]. MXNet is a high-performance framework for neural network training for desktops and servers realized in C++. However, MXNet only provides the mobile platforms with testing capability. In this work, we modify the
neural network training code in MXNet and recompile them to ARM-based Android devices. In order to realize efficient communication between nodes, Message Passing Interface (MPI) is adopted in AdaLearner. However, none currently existing MPI implementations explicitly offer support for Android devices. As such, we modify the source code and build process for an existing MPI implementation called Open MPI [79] such that it compiled and ran on Android devices. Open MPI relies on an SSH server and client implementation to serve as its underlying communication channel between nodes. Again, while multiple SSH implementations exist for standard Linux distributions, none of them offer direct support for Android platforms due to the difference in security and account implementations. To get around this, we modify the source code of Dropbear [80], an open-source SSH implementation that specifically targets embedded devices with low storage, computing, and memory resources.

By merging the aforementioned libraries with the implementation logic in AdaLearner, we realized a local distributed mobile system for neural network training on multiple mobile devices. Our experiments are conducted on LG Nexus 5X running Android 6.0.1 with a 1.8 GHz processor and 2GB RAM. The experimental setup is depicted in Fig. 3.10. We conduct the experiments on two image classification datasets: MNIST [81], CIFAR-10 [82]. In order to show the robustness of AdaLearner on different model complexities, three neural network model with increasing scales are tested based on these two datasets, which are called Multi-layer Perceptron (MLP), LeNet [19], and ConvNet [72], respectively. MLP only contains fully-connected layers while LeNet and ConvNet are two representative Convolutional Neural Networks (CNNs) on MNIST and CIFAR-10.
3.6.2 Evaluation of Efficient Computation Architecture

The bars in Fig. 3.11 show the relationship between the number of WNs and their corresponding execution time utilizing efficient computation training architecture. For efficient computation architecture, the total execution time for training is made up of the computation time denoted by the blue bars and the communication time which is presented as the orange part of the stacked column in Fig. 3.11. Here we define the baseline as the training time on local devices. Additionally, the total training time in the experimental results indicate the time consumption as the time spent in a whole distributed training procedure of a single minibatch iteration. The total number of iterations is defined by users. We can find that these three neural network models show different results compared with the baseline:

Fig. 3.11(a) shows the distributed training time on MLP, which keeps increasing with the increase of the number of WNs. On the one hand, MLP is a model with only fully-connected layers and is of extremely small scale and thus requiring less computation cost. On the other hand, the model size is comparatively big (0.44MB), increasing the communication overhead. Therefore, compared with the local training time (183ms), the communication overhead accounts for the main part in the total training time, which occupies from 76% to 93% of the total training time for 2 to 6 WNs. In this scenario, our scheduler assigns all the work to the local devices for the shortest training time.

The experimental results on LeNet in Fig. 3.11(b) show an obvious reduction in
Figure 3.12: Time consumption on three neural network models with efficient communication architecture.

The bars in Fig. 3.12 demonstrate the total training time using efficient communication architecture with 1-bit quantization technology. Besides computation cost and communication cost, Fig. 3.12 also includes the time of the encoding and decoding procedures. It can be clearly viewed that the extra overhead for encoding and decoding procedures is low when compared with computation cost and communication cost, which is at most 10%, 8.5%, and 1.3% of the total training time for MLP, LeNet, and ConvNet.

Mind that the communication time is the summation of transmission time and wakeup time. The wakeup time in our experiments is tested to be about 100ms in average, which affects the parallelism of small-scale neural network model like MLP.
In Fig. 3.12(a), although the transmission time decreases from $1572\text{ms}$ to $630\text{ms}$, the communication overhead is still the bottleneck with 1-bit quantization.

Thanks to the effectiveness of 1-bit quantization, when performing LeNet in AdaLearner with efficient communication architecture, the scalability is better than that of efficient computation architecture. As illustrated in Fig. 3.12(b), the total training time keeps decreasing from 2 to 8 WNs. In 8 WNs scenario, the total training time comes to $3276\text{ms}$, boasting the execution by $3.37\times$, which is better than the optimal scenario of efficient computation architecture ($1.78\times$). The same result occurs for ConvNet in Fig. 3.12(c), where the total training time is decreased to $3262\text{ms}$, which is lower than $3976\text{ms}$ under efficient computation architecture, achieving $3.53\times$ speedup.

3.6.4 Performance / Accuracy Tradeoff

The use of 1-bit quantization will may cause the tradeoff between training performance and potential accuracy drop. Fig. 3.13(a), Fig. 3.13(b), and Fig. 3.13(c) presents the accuracy results with the increase of training epoch with 2, 3, and 4 WNs on MLP, LeNet, and ConvNet, respectively.

Fig. 3.13(a) depicts the accuracy on MLP. Compared with the baseline accuracy without 1-bit quantization (97.8%), 1-bit quantization results in the accuracy of 97.5%, 97.5%, and 97% respectively for 2, 3, and 4 WNs. The accuracy drop is limited in less than 0.8%. For LeNet on MNIST, the corresponding accuracies of 2,
3, and 4 WNs in the 20th epochs are 99.12%, 99.00%, and 99.01%. The accuracy drop is constrained within 0.2% in comparison to the original baseline of 99.2%. However, the accuracy drops higher in ConvNet in Fig. 3.13(c). Compare with the baseline accuracy of 81.2% without using 1-bit quantization, our communication efficient training with 1-bit quantization on 2, 3, and 4 WNs can only reach the accuracy 79.4%, 78.1%, and 77.3%, incurring 2.9% accuracy loss in average. The high accuracy loss is due to the complexity of the dataset and the comparatively simple model structure adopt in our experiment. Hence, whether to utilize 1-bit quantization is according to the user real demand of performance-accuracy tradeoff.

3.7 Conclusion

In this work, we propose AdaLearner - an adaptive distributed mobile learning system for neural networks to enable parallel training of neural networks on mobile platforms. In AdaLearner, we first design the architectures tailored for distributed mobile devices: efficient computation architecture and efficient communication architecture. Then, 1-bit quantization technology is adopted and extended to convolutional layers in efficient communication architecture to largely compress the transmission data size for better system scalability. Additionally, an adaptive scheduler is designed to adapt the training configuration to the mobile resources so as to realize high execution parallelism. Finally, we realize all the functionality of AdaLearner on several Android devices and provide the total execution speedup with respect to a different number of worker nodes.
4.1 Preliminary

4.1.1 DNN Model Profiling on Mobile Devices

Although DNNs allow for high accuracy, the limited computing resources available on embedded platforms (e.g., smartphones) limit DNN accuracy and viability [76] [4]. Following the research of efficient inference of DNNs on mobile platforms, the industry develops machine learning toolkits for the mobile platforms. Some representative toolkits targeting at embedded platforms includes CoreML [83], MxNet [64], Caffe2 for mobile [65], and TensorFlow Lite [84]. Many prior works have focused on the speed-accuracy tradeoff between different DNN model structures. Table 4.1 illustrates the inference time of two representative DNNs [13][85] when run on flagship smartphones, as well as their realized Top-1 accuracy on the ImageNet dataset [34].

Table 4.1: Profiling of 2 state-of-the-art DNN models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Pixel 2</th>
<th>iPhone 8</th>
<th>Top-1 Acc</th>
<th>Param</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNet</td>
<td>166.5 ms</td>
<td>32.2 ms</td>
<td>70.9%</td>
<td>17 MB</td>
<td>224 × 224</td>
</tr>
<tr>
<td>Inception-v4</td>
<td>3180 ms</td>
<td>611 ms</td>
<td>80.2%</td>
<td>171 MB</td>
<td>229 × 229</td>
</tr>
</tbody>
</table>

Efficient Video Detection
Those analyzing results are gathered from the official site of TensorFlow [30], which are highly representative. In Table 4.1, MobileNet [13] is an mobile-friendly network structure designed for mobile devices, while Inception-v4 [85] represents the state-of-the-art in accuracy. We can find that even the execution of highly-optimized MobileNet is still limited to only 6fps on Pixel 2. Mind that the inference times in Table 4.1 are measured from image classification tasks, the input image resolution of which is around $200 \times 200$. However, typical input image resolution of video detection is about $600 \times 1000$. The corresponding computation cost increases by more than 10 times. Because deploying large video detection DNN on local mobile devices suffers from overwhelming computation costs and energy consumption, our detection system targets a cloud-based video detection system that can achieve higher accuracy and lower latency compared to local execution on mobile devices.

### 4.1.2 DNNs in Object Detection

Current main-stream DNN-based object detection frameworks can be divided into two categories: region-based detection and region-free detection.

Region-based detection methods function by performing two distinct steps: first generating region proposals, then performing classification on each of the proposed regions to create final detection results. An example of these types of detection methods is presented in [86], where Dai et al. proposed R-FCN, which combines both position-sensitive and classification features for higher detection performance. In [87], Kong et al. presented HyperNet, which aggregates hierarchical feature maps into a uniform space referred to as a hyper feature. In [88], RON is established, which leverages reverse connection to merge the features of different scales.

Conversely, region-free detection methods treat the object positioning and classification as a single problem. For example, YOLO framework [89] is proposed to detect the object position and class confidence directly from a single shot. SSD [90]
further improves it by utilizing different feature map scales to semantically represent different object scales.

Generally, region-based methods achieve higher accuracy than region-free methods because of their dedicated detection pipeline while region-free methods get faster execution speed because both localization and classification are done simultaneously. Both region-based detection and the region-free detection are mainly composed of a deep stack of convolutional layers as feature extractors, introducing tremendous computation cost. To get state-of-the-art accuracy, we incorporate a region-based detection method to our system, which is R-FCN in specific [86].

4.1.3 Motion Vector in H.264 Video Codec

Considering the network bandwidth, video contents captured from mobile cameras are compressed before being distributed. Among all the video coding formats, H.264 is one of the most widely-used formats, which is also known as MPEG-4 Part 10, Advanced Video Coding [91]. One key reason for the fierce compression rate of H.264 is its utilization of motion compensation. During encoding, H.264 protocol gathers certain successive video frames in a single group, which is named as Group of Pictures (GoP). The size of GoP can either be a constant value (e.g., 30) or be dynamically set based on the video content. Each GoP is made up of one I frame ("I" for Intra) and multiple P and B frames ("P" for Prediction or "B" for Bidirectional). I frame independently encodes a complete frame, which serves as the reference point for P and B frame in the same GoP. P frames are predicted by analyzing the difference between themselves with the previous P frames or I frames. Similar to P frames, B frames are also predicted by the frame displacement except that they are allowed to be compared with the later frames. Hence, P and B frame store fewer data with high resolution. The video frames are divided into several macroblocks, serving as the basic unit for predicting the frame difference. The macroblock size is typically
set to $16 \times 16$ in H.264. The difference between frames is described by motion vector, including the source and destination of all macroblocks. In MobiEye, we adopt H.264 codec for efficient communication between the mobile camera and the cloud under wireless networks such as Wi-Fi or LTE networks.

4.1.4 Feature Map Sparsity in DNNs

Spatial-wise feature map pruning is a more straight-forward idea. In many image-recognition problems, not all the space in the input image is of the same importance. Such fact inspires some researchers to consider how to eliminate the meaningless spatial area so that these areas do not need to be executed. Different from traditional model compression which focuses on the zeros in weights, feature map sparsity leverages the zeros in feature maps to accelerate DNN inference. Many priors works accelerate DNN execution based on the feature map sparsity derived from the semantic content of input images. In [92], Li et al. convert a DNN model into a cascaded structure for image segmentation so that some simple objects can be identified in earlier cascaded layers. In [93], the author proposed a novel deep layer cascade (LC) method which uses a single deep neural network to cascade different objects in a single frame. Fig. ?? shows the architecture of Layer Cascade IRNet (IRNet-LC). We can tell that, for IRNet, it does not compute all the spatial areas in all the layers. For example, the background of the image has already been extracted in stage1. All the following layers only calculate the area of the horse and person, which are detected in stage2 and stage3, respectively. In [59], Figurnov et al. proposed PerforatedCNNs. PerforatedCNNs uses interpolation to deal with the spatial area that is not calculated in the convolutional layer. The benefit of PerforatedCNNs is that it does not need to change the model structure while derives satisfactory speedup in testing with tiny accuracy loss. In SBNet [58], gather and scatter operators are designed to compute only the sub-blocks when a non-zero number in the feature map surpasses
Compared to model compression, the advantage of feature map sparsity with semantic properties (e.g., saliency) lies in the reduced amount of modification to the underlying DNN structure. In this paper, we design Spatial Sparsity Inference (SSI), which spatially skips the feature map based on the proposed computation masks to achieve speedup target with tiny accuracy loss. The idea and implementation of incorporating spatial sparsity in video detection system from DNN-level are inspired by PerforatedCNNs [59] and SBNet [58] with certain necessary modifications.

4.2 Methods

4.3 System Framework

Figure 4.1 shows an overview of our DNN-based video detection system. The upper part of Figure 4.1 represents the edge user part, which captures the frames, encodes them to video and transmits them to the cloud, waiting for the final detection result. The bottom part denotes the execution flow of cloud processing. The received video is first decoded and then fed to the DNN-based detector frame-wise.

Figure 4.1 also depicts the position where we add our three system-level optimization methods:

method to asynchronous execute the DNN inference for lower latency.

(2) **Video-based Dynamic Scheduling (VDS):** A system optimization which dynamically adjusts key frame detection frequency based on the metadata (motion vector) that has already presented in H.264 encoded video. The video decoder component extracts the metadata and sent them to the detector.

(3) **Spatial Sparsity Inference (SSI):** A DNN-related optimization that accelerates DNN inference by focusing only on visual saliency areas which can be manually decided or with dynamic computation mask. More specifically, SSI accelerates DNN by re-implement the convolutional layers in DNNs so that only the designated spatial positions are convoluted.

4.3.1 Video Detection Framework (DFF)

We adopt a state-of-the-art video detection system: Deep Feature Flow (DFF) [6] as DNN-based detector in our video detection system. The essence of DFFlies in that it divides video frames into two kinds: key frames and inter frames. Key frames are detected using traditional object detection DNNs, which is R-FCN in DFF, consisting of a feature extractor network, Region Proposal Network (RPN), and classification network. But, for the detection of inter frames, DFF compare themselves with the previous key frame using a smaller network (optical flow network) and then generate their feature map based on interpolation from the key frame feature map. According
to the default setting of DFF, a key frame occurs every 10 frames, with the other 9 frames being inter frames. As a result of that, the feature extractor network is substituted by an optical flow network for all the inter frames, saving a lot of inference time per video frame.

Figure 4.2 shows the execution flow in DFF detection framework for key frame and inter frame, respectively. The red arrow shows the execution flow of key frame while the green arrow shows the execution flow of inter frame. More specifically, DFF includes four network components:

- Feature Extraction Network (FeatNet): Feature extraction network takes the responsibility to extract the high-level feature from the raw image through stacks of convolutional, activation, and pooling layers. FeatNet is the most computation-intensive component among all the four components in DFF, which utilizes ResNet-101 [94].

- Optical Flow Network (FlowNet): For the detection of inter frames, Optical flow network computes the optical flow between the previous key frame and themselves. In order to enable end-to-end training, such optical flow is also realized by DNN called FlowNet [95].

- Propagation Function (Propagation): Propagation function takes the optical flow between key frame and inter frame as well as the feature map of key frame as input. Then, the propagation function interpolates the feature map of key frame using optical flow via bilinear interpolation.

- Region Proposal Network (RPN) and Classification Network: Region proposal network is applied on the feature map of each frame to locate the potential area of the objectiveness in the frames. The candidate Regions of Interest (RoIs) are then fed into the classification network, get the final object category by
pooling and voting procedures.

4.3.2 Asynchronous Computation (ADFF)

One limitation of DFF is that although it can achieve high framerate on average, inference requires different computation time depending on whether the current frame is a key frame or an inter frame. This is a result of the different network structures utilized by each type of data. When running the original DFF code on GTX 1080 GPU, the time of key frames is 0.09s, while the time of inter frames is 0.02s. It means if the key frame interval is set as 7, the first key frame detection latency is 0.09s followed by 6 inter frames that consume 0.02s per image. Such sequential execution leads to unbalanced inference time, which is not desirable for real-time applications.

Therefore, in our system, we execute the feature network and optical flow network asynchronously. Figure 4.3 compares the original sequential (top) with proposed asynchronous (bottom) execution of DFF when the key frame interval is set as 7. In sequential execution, the first and eighth frame require 0.6s, while asynchronous execution allows for an inference time of 0.2s for every frame. In asynchronous execution, we set utilize two threads: one to execute key frame inference and the
other to execute inter frame inference. Before the current key frame feature map is generated, the inter frame uses the feature map of the last key frame and thus asynchronously gets the detection results. By performing computations asynchronously, detection latency only depends on the execution time of the optical flow network and the transmission time of video frames. As shown in the bottom part of Figure 4.3, the first frame and the eighth frame are both key frame and inter frame for two parallel threads. In the example of Figure 4.3, the feature map delay is 3 frames as FeatNet is around 3× slower than FlowNet. More formally,

\[ D_f = \left\lfloor \frac{T_{\text{FeatNet}}}{T_{\text{FlowNet}}} \right\rfloor \]  

(4.1)

where \(T_{\text{FeatNet}}\) and \(T_{\text{FlowNet}}\) are the inference times for FeatNet and FlowNet, respectively, \(\left\lfloor \cdot \right\rfloor\) rounds up to the nearest integer, and \(D_f\) is the feature map delay in terms of frame count. In practice, \(D_f\) can be even smaller because of the communication overhead. For example, under the scenario where \(T_{\text{FeatNet}}\) is 0.6s and \(T_{\text{FlowNet}}\) is 0.2s and we assume that the communication cost and other operations time (e.g. RPN) equals 0.2s, the feature map delay \(D_f\) will be 2 \((0.6/(0.2 + 0.2))\).

The correctness of the multi-threaded execution of ADFF is assured by mutex to indicate when the current key frame feature map is available or not. Concretely, if the current frame to be detected is a key frame, a new thread will be created, executing the FeatNet. Once the thread is created, it will acquire a thread lock to prevent the later inter frames from getting the wrong key frame feature map. In the meantime, the current frame is also treated as inter frame so as to get the detection results immediately and transfer the results back to edge users. When the FeatNet of the newest key frame is generated, the thread lock will be released so that the subsequent inter frames acquire the lock can compare themselves with the newest generated feature map.
Another difference between sequential and asynchronous DFF is that, in order to reach the same accuracy, asynchronous DFF requires a smaller key frame interval. This is due to an increased time between key frames and the inter frames which utilize their feature maps, and is an unavoidable result of $D_f$. For example, the distance of sequential execution in Figure 4.3 is 6 while 9 for asynchronous one. During experiments, we find the two threads works better when they are deployed on different GPUs. The reason lies in that if GPU memory are shared by two thread executing two networks, the total GPU kernel is dynamically changed for each thread, resulting in the fluctuated inference time. If we execute two threads on two separate GPUs, the inter frame inference time will not be negatively affected by the parallel execution of key frame. Hence, in our experiment in Section 4.6, we set up our server with two GPUs, running key frame and inter frame separately.

4.4 Video-based Dynamic Scheduling

As part of our detection system, we target the elimination of redundant calculations within the video detection system. Our system attempts a similar goal by dividing the input into key frames and inter frames, only performing higher-cost inference on key frames when necessary. In DFF, key frames are selected at a set interval, regardless of the underlying video data [6]. This overlooks the case where video clips may contain long runs where there is very small displacement from one frame to the next few frames. In such a case, key frame interval can be greatly increased without loss of accuracy, reducing the requirement for expensive key frame inference calculations. In our system, we design Video-based Dynamic Scheduling (VDS) to dynamically determine whether the current frame is a key frame or not. By adopting VDS, the feature map network is frequently executed when the movement in video is fast, and rarely executed when movement is slow.
4.4.1 Motion Vector and Optical Flow

The idea of Video-based Dynamic Scheduling (VDS) is inspired by the similarity between motion vector and optical flow.

Motion vector in H.264 works in discrete domain, the values of which represent the displacement of sub-blocks in the video frame. Optical flow is the motion of image brightness in continuous domain, which could be caused by brightness change that is not caused by real motion. But in 2-dimension images, optical flow is the same as motion vector in ideal case [96].

One straight-forward idea is directly substituting motion vector in H.264 for the FlowNet in our system. However, such a solution leads to several problems:

- Because H.264 utilizes variable block-size motion compensation, the resolution in H264 motion vector could not be guaranteed to be identical with the output dimension of FlowNet.

- During our experiment, we find that the sub-block size in H.264 motion vector is not guaranteed to be the same for different video frames, making it hard to be converted to pixel-wise optical flow.

- The motion vector in H.264 is realized targeting video encoding and decoding so that the accuracy of H.264 motion vector is much lower than FlowNet.
Unlike the work in [42], which utilizes motion vector from hardware-level, our work focuses on software level optimization. Therefore, the aforementioned problems prevent us from using motion vector in H.264 directly.

Additionally, we also tried to replace FlowNet with a more light-weighted optical realization such as Lucas Kanade optical flow [97] or Gunnar Farneback optical flow [98]. As can be seen in Figure 4.4, the accuracy of Lucas Kanade optical flow is far worse than Gunnar Farneback optical flow although Lucas Kanade method is much faster than Gunnar Farneback method. The experimental results show that, even if we utilize Gunnar Farneback optical flow to replace original FlowNet, the Mean Average Precision (mAP) still drops 9.6% when the frame interval is set as 10, which is unacceptable. This result is consistent with the statement made in [6] that the accuracy of FlowNet is the key to achieve a better speed-accuracy tradeoff.

### 4.4.2 H.264 Motion Vector Extraction

The key idea of capturing inter-frame movement quantity information is based on the H.264 video codec. During server-side video decoding, the motion vector of the frame is extracted at the same time. The H.264 GoP pattern adopted in our system is described in the bottom of Figure 4.5, the top part of which shows a standard GoP pattern. The GoP pattern of our system does not include B frames as the nature of live video streaming means the required information from future frames is
never available. Targeting dynamic key frame scheduling, the number of reference frames is set to 1. In this way, all P frames motion vectors are calculated based on a single prior frame, as shown in the bottom of Figure 4.5. Note that motion vector extraction incurs no additional computation cost as the motion vector is already encoded as part of the H.264 video stream.

4.4.3 Video-based Dynamic Scheduling (VDS)

In order to achieve high efficiency, VDS is a simple scheduling scheme. VDS initializes the dynamic frame interval ranging from $f_{i_{\text{min}}}$ to $f_{i_{\text{max}}}$, and the motion vector magnitude from $mv_{\text{min}}$ to $mv_{\text{max}}$. For each video frame, VDS calculates a scalar $mv_{\text{mean}}$, representing the mean of the magnitude of the incoming $mv_{\text{cur}}$. Following this, VDS maps $mv_{\text{mean}}$ to a new key frame interval ($f_{i_{\text{cur}}}$) using Min-Max linear mapping. We utilize $mv_{\text{acc}}$, the accumulated motion vector over several continuous frames, to denote the total motion from the last key frame until the current frame. When $mv_{\text{acc}}$ is larger than $mv_{\text{max}}$, the minimum frame interval ($f_{i_{\text{min}}}$) will be assigned to the current frame interval ($f_{i_{\text{cur}}}$). We can find that the only computation in VDS is vector summation and averaging, incurring little computation cost compared with DNN inference. Furthermore, unlike [99], VDS does not require extra learning procedures.

4.5 Inference with Spatial Sparsity

In Section 4.4, temporal redundancy in video detection system is eliminated via VDS. In this section, a method for removal of spatial redundancy in video detection system is also detailed.
4.5.1 Spatial Sparsity Inference (SSI)

The key idea of Spatial Sparsity Inference (SSI) lies in skipping unimportant pixels in order to accelerate DNN execution. In SSI, we design two computation masks, indicating the spatial position that could be skipped: (1) **Computation mask based on feedback detection results**: Note that, in DFF, the inter frame is detected based on the key frame and the optical flow between these two frames. Therefore, the background part of both frames need not be examined by FlowNet in DFF, as background movement would not affect final detection results. Figure 4.6 shows an example of a feedback computation mask, where only the region within the red bounding box is executed. The red bounding box is defined by the detection results from the key frame (green box), with an additional margin of constant size. In MobiEye, we set the default margin as 64 due to the fact that, in FlowNet, the ratio between the spatial size of the feature map before the first convolution and
the smallest feature map after convolution is $64 : 1$. The ratio can be inferred from Figure 4.11, which is $w/2$ or $w/2$ divided by $w/128$ or $h/128$. For the feedback computation mask in SSI, the non-zero mask area grows with the increase in interval frames between the last key frame and current inter frame due to the expected increase in movement. (2) **Computation mask based on brightness error**: Feedback computation masks reduce computational redundancies due to reprocessing of the image background. However, they can not be applied when feedback detection results cover the entire video frame. To deal with such a case, we design a computation mask based on brightness error calculated by subtracting inter frame from key frame, which is defined as:

$$\text{Mask} = 0 \text{ where } |B_{\text{key}} - B_{\text{inter}}| < \text{thd}, \text{ else } 1; \quad (4.2)$$

where $\text{thd}$ stands for threshold of brightness error to be skipped and $B_{\text{key}}$ and $B_{\text{inter}}$ represent pixel value of key frame and inter frame, respectively. Figure 4.7 shows the brightness error mask of an elephant video with a brightness threshold of 15. It can be seen that a large region is unnecessary for re-calculation even though it falls within the detection bounding box.

### 4.5.2 Realization of SSI Index Layer

The task of SSI index layer is to convert a binary computation mask to a non-zero index array as well as the total size of non-zero numbers. The output of SSI index layer will be fed to SSI convolutional layer so that the computation of zero activations...
could be efficiently skipped. The realization of SSI index operation on CPU is very simple with the help of high-level API in existing scientific computing libraries such as Numpy [100]. So, in this section, we focus on the GPU realization of SSI index operation. Figure 4.9 depicts how to realize an efficient SSI index operation on GPU in parallel. Each position of the binary computation mask is assigned to one GPU kernel for parallel execution. Then, for each GPU kernel, it decides whether the assigned number is 0 or 1. If the assigned number is 0, the kernel will end its execution thread. Otherwise (1), the kernel will first do an atomic addition to the global position counter. Atomic addition assures the correctness of parallel execution of SSI index operation so that two GPU kernel will not access and update the same position in the output non-zero index array. After that, the thread will get the old value before the atomic addition, indicating the position that can be filled for the current GPU kernel. Last, such GPU kernel will update the position in the non-zero index array with the index of the non-zero number in the original computation mask. As shown in the example of Figure 4.9, the final non-zero index array contains 0, 2, and 5, which are the position index of the input binary computation mask. Such implementation is identical with SBNet [58] except that SBNet does such index procedure block-wise while our work does it pixel-wise.

Figure 4.9: Realization of SSI index operation on GPU.
4.5.3 Realization of SSI Convolutional Layer

Figure 4.8 presents the original layer modules (left) and their corresponding SSI modules (right), except that SSI skips the spatial pixels for fast execution. Compared with the original layer module with the feature map as input, the SSI module is fed the same spatial size computation mask (as described in Section 4.5.1) and feature map. To deal with the scale change in different DNN layers, we borrow the idea from SBNet [58] which uses a pooling operation followed by a threshold to downsample the input computation mask. The convolution in the original layer module is replaced with a SSIConv Layer, which is similar to the perforated convolutional layer in PerforatedCNNs.

Because SSI belongs to structured sparsity, it can better accelerate DNN execution with fully-optimized, dense GEMM in both CPU and GPU mode. Figure 4.10 details the procedures of the SSIConv operation layer. Before matrix to matrix multiplication in step 2, the input feature map is first expanded from a 3-dimension to a 2-dimension tensor with a patch to column operation. Patch here means a subset of feature map which is of the same dimension of the convolution kernels. Originally, each patch in the feature map are flattened to a single matrix column iteratively through X-axis and Y-axis. In SSIConv, patch to column operation takes the non-
zero index as input and iteratively assigns the column base on the patches with the non-zero index. Compared to the original patch to column implementation, the output matrix is smaller with fewer columns, leading to less memory consumption. Take step 1 in Figure 4.10 as an example, the output matrix column is 3 because the non-zero index only includes three index (0,2,5). Compared with original implementation with 9 columns in the output matrix, SSI_Conv saves $2/3$ memory consumption. Consequently, step 2 in Figure 4.10 does the matrix multiplication with a smaller matrix, saving $2/3$ computation cost compared with original matrix multiplication. Step 3 is an extra step for SSI_Conv operation, which expands the output matrix to the original size based on the non-zero index.

### 4.5.4 Applying SSI on FlowNet

After using ADFF, the latency bottleneck comes to the inference time of FlowNet. Thus, in our video detection system, we apply SSI on FlowNet to further accelerate the inference latency per video frame. The FlowNet adopted in DFF is similar to the FlowNetSimple model architecture in original work with little modification [95] [6].
The detailed network architecture is shown in Figure 4.11. The input of FlowNet is a frame with height \( h \) and width \( w \), which is first compressed by half for both dimensions in DFF [6]. There are totally six convolution groups, each of which decreases the original input feature map size by half. There is one convolution operation in the groups of Conv1, Conv2, while there are two convolution operations in the groups of Conv3, Conv4, Conv5, Conv6. After six convolution groups, the output feature map dimension comes to \( h/128 \times w/128 \), which is fed into the refinement layers. The refinement layers finally generate the optical flow with the spatial dimension of \( h/16 \times w/16 \) with 2 channels, representing the displacement of two frames from \( x \) axis and \( y \) axis. Note that SSI index operation contains atomic addition, which needed to be executed sequentially even on GPU. The time consumption of SSI index operation increases quadratically with the increase of computation mask spatial scale because of the atomic addition in Figure 4.9. As a result of that, applying SSI on Conv1 and Conv2 could not achieve speedup in general. Therefore, in our system, we apply SSI after two convolution groups.

4.5.5 Comparison of SSI with Related Works

The realization of SSI is jointly inspired by PerforatedCNNs [59] and SBNet [58], with necessary modifications. More specifically, SSI is a combination of PerforatedCNNs and SBNet. The reason why we combine the idea of PerforatedCNNs with SBNet in SSI is due to their functional limitations when taken separately:

If individually adopted:

- PerforatedCNN supports efficient pixel-wise skipping but the skip index is static with the same computation mask for all input images. PerforatedCNNs could not be directly applied to our system because, in our system, the computation mask is dynamically changing for the video frames.
SBNet supports dynamic computation mask but only realizes convolution operation speedup in block-wise situations. However, SBNet could not achieve speedup with a small scale feature map as input, which is common for the latter layers in FlowNet. This is due to the computation overhead incurred by Gather and Scatter when the feature map is of low dimension but high channel size.

The reason why SBNet could not realize speedup under pixel-wise setting lies in that SBNet adds two operations before and after original convolution operation, which are Gather and Scatter. Gather operation slices an input feature map into \(B\) tensors with the height and width of a single sub-block, where \(B\) stands for the total number of dense sub-blocks. Scatter operation is the inverse operation of Gather operation. Due to the complexity of tensor conversion, it could not achieve speedup when the sub-block is small.

Therefore, SSI adopts the dynamic computation mask method from SBNet and adopts efficient convolution from PerforatedCNNs, yielding an efficient convolution infeference with spatial sparsity.

4.6 Experiments

4.6.1 Experimental Setup

**Test Benches:** We implement our video detection system using the DFF [6] framework, with all proposed optimization schemes compared to DFF as baseline. For fairness of comparison, we directly utilize trained models from DFF without fine-tuning for all experiments. We adopt the ImageNet VID dataset for evaluation, which includes 5354 annotated videos. The model structures adopted for the feature extraction network and optical flow network are ResNet101 [94] and FlowNet [95], respectively. Video frames are resized to 600 pixels on the shorter side as the input.
of the feature network, and 300 pixels on the shorter side for the optical flow network. In our experiments, the accuracy-related performance of the video detection system is reported using a mean average precision (mAP), speed-related performance is evaluated in terms of sparsity and milliseconds (ms).

**System Environment:** For the client side, we adopt the Nexus 5, and Pixel 2, representing two popular Android-based smartphones. The Nexus 5 is powered by a Quad-core Krait CPU with an Adreno 330 GPU and 2GB of RAM. The Pixel 2 is equipped with an Octa-core Kryo CPU with an Adreno 540 GPU and 4 GB RAM. We implement and deploy a H.264 video encoding application on both devices which utilizes the EGL interface for efficient GPU-accelerated encoding. For the server side, we deploy our system on a server running Ubuntu 16.04, with a 16-core, 2.4GHz Intel Xeon CPU, two NVIDIA GPU’s (GeForce GTX 1080 and GeForce GTX TITAN), and 128GB RAM. Corresponding to the video encoding performed on the Android devices, we establish a video decoding application on the server with low-level ffmpeg and x264 libraries. For the DNN video frame inference module on the server side, we extend the existing DFF project with our optimizations. DFF itself is based on MXNet [64], a powerful deep learning framework developed by DMLC team. Figure 4.12 (a) shows the histogram of the motion value derived from ImageNet VID. For ease of visualization, all motion values larger than 10 are truncated. From
the figure, it can be seen that the distribution of motion values is polarized: 29% of the video motion values are smaller than 1 while 23% of the values are larger than 9. The mean of the motion values is 6.83, and the median is 3.05. Figure 4.12 (b) shows the relationship between different image scales and their corresponding encoding/decoding time. Generally, Pixel 2 is faster than Nexus 5 for video encoding, which consumes 14.9 ms and 22.8 ms on average, respectively. When the image scale reaches 246K on the Pixel 2, the encoding time becomes shorter than when encoding lower-scale images. This is due to the highly parallel nature of both the encoding operation and of the mobile GPU itself, resulting in more efficient computation when performed at scale. The video decoding time from server side is 3.9 ms per frame on average.

4.6.2 Evaluation of Communication Cost

Figure 4.13 illustrates the average communication time per frame of transmitting the H.264 encoded video of different resolutions under different uploading bandwidths. The upload bandwidth we choose in Figure 4.13 is from 6 Mbps to 21 Mbps, which is the common uploading bandwidth range in LTE [101], which is currently the most popular communication standard for mobile devices. From Figure 4.13, we find that H.264 could fiercely compress the video frame with high resolution. Even
with 1280 × 960 video resolution, the average frame size is 1.98 Mb while the original raw data is 9.83 Mb. However, under low uploading bandwidth, the communication latency is still high (e.g. 330 ms under 6 Mbps for 1280 × 960 video resolution). Because the communication latency is linear related to the image resolution, different resolutions could be used for different user requirements under different uploading bandwidth. Note that we could achieve even higher uploading bandwidth using other communication protocols such as 5G network or Wi-Fi.

4.6.3 Evaluation of ADFF

Figure 4.14 compares DNN inference time between original sequential DFF and our proposed ADFF. As indicated in Figure 4.14, the time consumption of each video frame always equals the time for inter frame in ADFF. We find that the inference time for key frame and inter frame are 93 ms and 20 ms for GTX TITAN, and 89 ms and 19 ms for GTX 1080. Given this, we deploy the key frame thread on GTX TITAN and the inter frame on GTX 1080, as the latter card results in lower latency for ADFF. In all cases, the time consumption per frame in the original sequential DFF baseline is much higher than ADFF, which is due to the bottleneck of key frame execution. Note that delay in ADFF is set as 3, and that delay should be no bigger than the key frame interval. Hence, Figure 4.14 shows the key frame interval from 6.
From the dotted line in Figure 4.14, we find that ADFF achieves $1.05 - 1.47\times$ and $1.1 - 1.5\times$ speedup compared with sequential DFF on GTX 1080 and GTX TITAN, respectively. Another benefit of ADFF is realized as uniform frame timings, whereas sequential DFF is highly fluctuating, tremendously affecting user experience.

4.6.4 Evaluation of VDS

Figure 4.15 presents the results of VDS scheme in our system, where the gray dots show the baseline speed-accuracy tradeoff with a static key frame interval. For VDS, we try three max motion values: 10, 20, and 30, the results of which are colored as orange, yellow, and green. For each max motion value setting, we set the minimum key frame interval as 5, 10, 15 and max key frame interval as 20, 30, 40, forming a totally 9 dynamic interval ranges. As shown in Figure 4.15, when the key frame interval is small (e.g., $\leq 12$), VDS does not show an advantage compared to static key frame scheduling. The reason for this is that the dynamic interval range is small and thus there exists little optimization space for higher accuracy. The maximum accuracy that can be reached is the situation where all the video frames are considered as key frame. With the increase of key frame interval, we derive a higher accuracy than baseline when adopting VDS. For example, when the average key frame interval is 21.6, the mAP reaches 70.6%. Meanwhile, the mAP of static key frame scheduling
only achieves 69.9% when the key frame interval is 21. Therefore, we find that VDS does help the prediction of key frame scheduling to achieve better accuracy with less computation.

4.6.5 Sparsity-accuracy Tradeoff Evaluation of SSI

To evaluate SSI, we set three brightness error mask thresholds: 10, 20, and 30 to show the tradeoff between the spatial sparsity and accuracy. Figure 4.16 shows the sparsity-accuracy tradeoff when adopting SSI in our system. The average spatial sparsity equals 52.8% when only applying the feedback computation mask and the mAP drop is controlled within 1% for all the key frame interval settings. Furthermore, the mAP drop is negligible (e.g. < 0.5%) when the key frame interval is small (e.g. < 9). With the increase of the key frame interval, the spatial sparsity keeps decreasing because more computation is needed due to feedback computation mask. The reason lies in that larger key frame interval incurs larger movement between key frame and inter frame and thus the bounding box in the feedback computation mask is larger. As illustrated in Figure 4.16, the spatial sparsity increases with the increase of brightness error mask thresholds. Take the key frame interval 9 as an example, the 53%, 68%, and 75% for threshold 10, 20, and 30, respectively. Their corresponding mAP is 72.53%, 72.18%, and 71.85%, incurring 0.35% and 0.68% mAP drop. Therefore, we find that setting the brightness error mask threshold as a small
number (e.g., 10) leads to a better sparsity-accuracy tradeoff. One limitation of SSI is that it could not achieve speedup when the input feature map is large in height and width. So, we apply SSI after 2 pooling operations in FlowNet. In our experiments, SSI achieves $1.06 \times - 4.25 \times$ speedup on convolutional layers of FlowNet with 60% – 90% spatial sparsity. Because FlowNet only contains 12 convolutional layers after 2 pooling operations, the impact of SSI on speedup of the whole inference operation is limited to $1.01 \times - 1.15 \times$ under 60% – 90% spatial sparsity due to operations such as region proposal network, deconvolution, etc.

4.6.6 Speed Evaluation of SSI Index

Figure 4.17 depicts the time consumption of SSI index operation with different scales of computation mask and spatial sparsity using GeForce GTX 1080. From Figure 4.17, we could find that the time consumption of SSI index operation is linearly related to the sparsity and quadratically related to the scale of the input image. Here, the scale means the side length of the input frames. In general, the total time consumption of SSI index operation is linearly related to the number of non-zeros in the computation mask, which is due to the atomic operation in SSI index operation. For example, the time consumption of SSI index reaches 0.94 ms when the computation mask is $80 \times 80$ with 0.7% sparsity, which is unacceptable because the standard convolution operation just costs about one millisecond. Therefore, in order to get
4.6.7 Speed Evaluation of SSI Convolution

Figure 4.18 demonstrates the speedup of SSI convolution operation using GeForce GTX 1080. Figure 4.18 is derived from running standalone layer-wise speedup tests, where the layer setting is 128 channels and the number of filters, $3 \times 3$ convolution kernel with 1 padding and stride. The implementation detail is similar to PerforatedCNNs. So, in our video detection system, if the computation mask is static or predefined, the speedup in Figure 4.18 is the final speedup that SSI could achieve.

SBNet claims that it achieves $0.88 - 3.39 \times$ speedup under 90% sparsity with different input image scale [58]. Compared with SBNet, SSI convolution achieves higher speedup under the same input image scale, which achieves $1.25 - 3.79 \times$ speedup under 80% sparsity and $1.32 - 5.76 \times$ speedup under 90% sparsity, when the input image scale is ranging from $50 \times 50$ to $500 \times 500$. The reason lies in SSI require less memory allocation and memory copy, which is required in SBNet. Moreover, similar to SBNet, with the decrease of input image scale, the speedup becomes lower than that of large input image scale in SSI convolution. This phenomenon is due to that the total parallel threads size of tested GPU could already cover the parallel capacity.
Table 4.2: Configuration of convolutional layers in FlowNet.

<table>
<thead>
<tr>
<th></th>
<th>Input Dim</th>
<th>Output Dim</th>
<th>Kernel</th>
<th>Stride</th>
<th>Filter Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1</td>
<td>300×500</td>
<td>150×250</td>
<td>7</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>Conv2</td>
<td>150×250</td>
<td>75×125</td>
<td>5</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>Conv3</td>
<td>75×125</td>
<td>38×63</td>
<td>5</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>Conv3_1</td>
<td>38×63</td>
<td>38×63</td>
<td>3</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>Conv4</td>
<td>38×63</td>
<td>19×32</td>
<td>3</td>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td>Conv4_1</td>
<td>19×32</td>
<td>19×32</td>
<td>3</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>Conv5</td>
<td>19×32</td>
<td>10×16</td>
<td>3</td>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td>Conv5_1</td>
<td>10×16</td>
<td>10×16</td>
<td>3</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>Conv6</td>
<td>10×16</td>
<td>5×8</td>
<td>3</td>
<td>2</td>
<td>1024</td>
</tr>
<tr>
<td>Conv6_1</td>
<td>5×8</td>
<td>5×8</td>
<td>3</td>
<td>1</td>
<td>1024</td>
</tr>
</tbody>
</table>

![Layer-wise speedup on FlowNet with SSI convolution.](image)

Figure 4.19: Layer-wise speedup on FlowNet with SSI convolution.

of GEMM operation when the feature map resolution is low [102]. So, higher spatial
sparsity could hardly achieve proportional speedup.

4.6.8 Evaluation of FlowNet with SSI

Table 4.2 shows the configuration of each convolutional layer in FlowNet used in
our video detection system with input video frame dimension of 600 × 1000. Figure
4.19 their corresponding speedup with different sparsity with static computation
mask using GeForce GTX 1080. The speedup results is quite similar to that in
Figure 4.18. Later layers get lower input dimensions of the feature map. So, their
corresponding speedup is limited compared with prior layers. Furthermore, SSI could
achieve marginal speedup with comparatively low sparsity such as 50%. So, it is recommended that SSI could be used when the spatial sparsity is high enough. Mind that the speedup results in Figure 4.19 only considers the SSI convolution operation. Therefore, if we adopt dynamic computation mask for each video frame, the overall speedup will be lower because of the overhead of SSI index operation. Fortunately, for the computation mask based on feedback detection results, the SSI index operation could be done once the feedback bounding boxes of the last video frame are available, which will not affect the total latency.

4.6.9 Memory Evaluation of SSI

Figure 4.20 shows the memory consumption for original convolution, SSI convolution, and SBNet convolution, respectively. We calculate the memory consumption with different non-zero sparsity under the setting of 128 3×3 convolution kernels and a 100×100 feature map of 128 channels as input. In order to show the memory consumption of SBNet with different block granularity, we shows 3 numbers for each sparsity level, which are named as SBConv10, SBConv30, and SBConv50. 10, 30, and 50 represents the number of blocks from each side of the feature map. For example, SBConv10 means that the feature map is divided into 10×10=100 blocks while SBConv50 means the feature map is divided into 50×50=2500 blocks. We could find from Figure 4.20 that SSI and SBNet require similar or even more memory than
original convolution when the spatial sparsity is under 40%. So, Neither SSI nor SBNet achieve very satisfactory speedup with low spatial sparsity. Additionally, SSI requires less memory then SBNet because SBNet needs to make a copy of the input feature map during the Gather operation. Moreover, with the decrease of block granularity in SBNet, the memory consumption keeps increasing, which is due to the number of total blocks and the overlap between each block.

4.6.10 Overall Evaluation

We evaluate our video detection system under the setting of key frame interval = 20, video scale = 800 × 600, network throughput = 10Mbps, video size per frame = 47KB, and H.264 Bitrate = 2Mbit/s. The corresponding communication and codec latency sums up to 30ms.

The baseline inference latency from the server side is 89.8ms for FeatNet and 19ms for FlowNet. After adopting ADFF, the execution of FeatNet is hidden so that the execution bottleneck becomes only FlowNet latency. With the same key frame interval, VDS increases the mAP by 0.6%. Finally, If we want to further exchange accuracy with computation cost, SSI saves another 0.2ms on FlowNet by sacrificing 0.6% mAP. Note that the average time saving of 0.2ms is derived with both SSI index and SSI convolution. If the computation mask is asynchronously generated, SSI could achieve more speedup.

4.7 Conclusion

In this work, we propose an efficient cloud-based video detection system for real-time applications and we call it MobiEye. Our system adopts the state-of-the-art video detection framework DFF with several optimizations. From system level, we design ADFF, which utilizes multi-thread technology to asynchronously execute the DNNs in DFF while insure the functionality correctness via Mutex. From algorithm level,
we propose VDS to dynamically decide the key frame based on H.264 motion vector and design SSI for spatially partial inference based on both feedback detection results and brightness error. Our video detection system is able to reach real-time requirement of video detection applications on mobile platforms with marginal accuracy drop.
5

Efficient Neural Machine Translation

5.1 Preliminary

5.1.1 Model Profiling on Mobile Devices

Although Transformer could be categorized as deep neural network which is full of linear weight matrix that seems amiable to deep compression [55]. The weight magnitude of Transformer models is actually much larger than that of CNNs. Figure 5.1 compares the weight distribution of Transformer on WMT dataset [103] and ResNet [94] on CIFAR-10 dataset [82]. Transformer and ResNet are targeting at language processing and computer vision, respectively. For the ease of comparison,

![Figure 5.1: Weight distribution of (a) ResNet and (b) Transformer.](image-url)
we visualize the weights from -1 to 1 for both models. Comparing with ResNet in
Figure 5.1, Transformer’s weights are distributed more smoothly in a wider range.
Therefore, pruning Transformer model is more complicated than pruning CNNs like
ResNet, which is also proved in many previous works.

5.1.2 Transformer Model Pruning

As mentioned in Section 5.1.2, Many previous works have been done in the field of
Transformer model compression. In the field of NMT tasks, besides efficient Trans-
former variance [54, 53, 104], Transformer pruning is the most popular topic for
Transformer compression due to its simplicity and practicality [60, 61, 1, 2, 62].
These Transformer pruning works could be classified in terms of different pruning
granularities.

Layer-wise pruning removes Transformer layers as the smallest pruning granu-
ularity. For example, in [60], Fan et al. propose LayerDrop, which selects sub-network
of any depth from the original Transformer network during inference time. In Lay-
erDrop, an drop rate is learned for each Transformer layer during training and only
highest-scored layers are adopted during testing.

Head-wise pruning removes attention heads in Transformer as the smallest
pruning granularity. For example, in [1], Voita et al. apply gates on each head
of MHA for pruning using stochastic relaxation, which prunes half of all heads in
Transformer with less than 0.25 BLEU degradation. In [61], Michel et al. also
mask the heads in MHA iteratively based on the proposed proxy score to expect the
sensitivity of the model to the masked variable.

Line-wise pruning removes rows or columns in Transformer layer matrices as
the smallest pruning granularity. For example, in [2], Kenton et al. utilize auto-
sizing to prune Transformer model weights with regularization and proximal gradient
descent. However, auto-sizing severely affects the performance with pruned model.
For example, the BLEU score drops from 27.9 to 20.9 when 20% of parameters are removed in Arabic to English translation task.

**Element-wise pruning:** removes random weights as the smallest pruning granularity. For example, in Transformer.zip [62], Cheong *et al.* implement iterative magnitude pruning, which masks out weights less than a certain threshold and iteratively retrain the model. Although element-wise pruning in general potentially leads to higher sparsity than other pruning methods because of its randomness, it only achieves real speedup under specific hardware settings.

Moreover, BERT is a self-supervised approach to train a pre-trained model for many downstream language tasks with Transformer structure [23]. Many model compression works have also be done targeting at BERT utilizing the technologies ranging from transfer learning [51, 52], pruning [105, 106, 107], and quantization [108, 109].

TPrune falls in the category of line-wise pruning. In this work, we mainly focus on the speedup of the execution of NMT tasks of Transformer models using pruning methods when deployed on resource-constrained platforms. Line-wise pruning is the most fine-grained pruning method that could achieve real speedup in general mobile settings.

### 5.1.3 Model Pruning with Regularization

When pruning a model, the ideal training loss is the $L_0$ regularization of model weights. In such a case, $L_0$ represents the number of non-zero weights. However, $L_0$ is non-convex and thus difficult to minimize using gradient-based methods. In [110], Han *et al.* iteratively prune a fixed percentage of weights with the smallest magnitude. Such a heuristic method could hardly achieve the optimal compression rate. As a result of that, $L_1$ regularization is commonly used in model pruning as an alternative. In [111], Liu *et al.* apply $L_1$ regularization on weights to achieve element-wise
sparsity. However, the drawback of $L_1$ lies in its scale-variance property, which means the gradient varies with respect to the scale of original weight magnitude.

Structured pruning extends element-wise pruning to group-wise. By removing groups of weights from original model weights, real speedup could be achieved and thus more feasible for the deployment on mobile platforms. In [39], Wen et al. group each column and row of the weight matrix using $L_2$ norm and minimize the $L_1$ of all these grouped $L_2$ norm. Similarly, In [2], Kenton et al. apply $L_1$ to the $L_\infty$ of all the grouped weights, where the scale variance problem still holds. In [1], Gate method is proposed to be applied on each head of MHA component in Transformer. The gate value is drawn independently from a parameterized concrete distribution. However, such a method incurs extra overheads in the gate approximation, making it less scalable.

In TPrune, we adopt Structured Hoyer Square (SHS) for structured Transformer model pruning [7]. Previously, the effectiveness of SHS on CNNs has been demonstrated [7]. In this work, we adopt it to Transformer models for sequence-to-sequence applications.

5.2 Methods

5.3 Transformer Redundancy Analysis

5.3.1 Analyzing Model Components

As mentioned in Section 5.1, Transformer is mainly made up of two layer components, of which we want to decrease the parameter size:

Multi-Head Attention (MHA): Multi-head attention components take three matrices: Query (Q) of dimension $d_q$, Key (K) of dimension $d_k$, and Value (V) of dimension $d_v$ as input and generate the transformed matrices of dimension $d_o$. In [3], $d_q=d_k=d_v=d_o=d_{\text{model}}/n$, where $n$ represent the number of heads in MHA.
components, which is set as 8. Hence, $d_q=d_k=d_v=d_o=d_{model}/h=512/8=64$ For multi-head self-attention, $Q$, $k$, $V$ are identical. Formally, MHA is calculated as:

$$MHA(Q, K, V) = Concat(h_1, ..., h_n)W^o,$$

where $h_i = Attn(QW_i^Q, KW_i^K, VW_i^V)$. 

In Equation 5.1, the attention function ($Attn$) equals:

$$Attn(Q, K, V) = softmax(QK^T/\sqrt{d_k})V.$$ (5.2)

As expressed in Equation 5.2, attention operation basically multiplies $Q$ and $K$ (scaled by $\sqrt{d_k}$), applying softmax on the results and multiply it with $V$, where no model parameters are involved. Therefore, our pruning targets of MHA are $W^Q, W^K, W^V$, and $W^o$, where $W^o$ is called output transform matrix. Because $Q$, $K$, $V$ matrices are of the same shape, $W_i^Q$, $W_i^K$, and $W_i^V$ ($i = [1...8]$) could be combined into single weight matrix and computed in parallel. $W^Q, W^K, W^V$ are of shape $(d_{model}, d_{model})$ and $W^o$ is also of shape $(d_{model}, d_{model})$.

**Fully-connected Feed-forward Network (FFN):** Fully-connected feed-forward networks take the output of MHA as input and consist of two subsequent fully connected layers with ReLU activation function in between, which is formulated as:

$$FFN(x) = max(0, xW_{ffn1} + b_1)W_{ffn2} + b_2,$$ (5.3)

where $b_1$ and $b_2$ are biases. Therefore, $W_{ffn1}$ and $W_{ffn2}$ are the pruning targets of FFN components, which are of shape $(d_{model}, d_{ffn})$ and $(d_{ffn}, d_{model})$, respectively. Following [3], $d_{ffn}$ equals 2048, which is 4 times of $d_{model}$.

### 5.3.2 Analyzing Target

Although we target at fully-connected layers in Transformer, some challenges still exist about learning the compact structure of Transformer. Transformer is composed of
encoder and decoder, which may have different layer properties due to their different functionalities. Moreover, unlike the fully-connected layers in CNNs, the input and output dimension of the weight matrix represent different meanings. For example, the input dimension of $W_{\text{ffn}1}$ denotes the dimension of the input word embedding ($d_{\text{embed}}$) while the output dimension represents the linear transformations ($d_{\text{ffn}}$). We need to prune the model with enough understanding of which dimension is better for pruning. In other words, to better prune a Transformer model, several questions are required to be answered about the Transformer properties:

- For each of our pruning targets ($W^Q, W^K, W^V, W^o, W_{\text{ffn1}}$ and $W_{\text{ffn2}}$), should we prune them row-wise, column-wise or from both directions?
- Should we treat encoder and decoder the same during pruning?
- Is the pruning properties the same for the Transformer from shallow layers to deep layers?
- Should $W^Q, W^K, W^V$ be pruned with the same sparsity in MHA?

### 5.3.3 Block-wise Structured Sparsity Learning (BSSL)

In order to answer the questions in our analyzing target, we propose Block-wise Structured Sparsity Learning (BSSL). In BSSL, we divide the original weight matrix...
into several sub-blocks with the same shape. We apply group lasso penalty row-
wise and column-wise on each sub-block with identical weight decay. Row-wise and
column-wise BSSL regularization on a single sub-block of shape \((r, c)\) is formulated as:

\[
L_{\text{row}}(W) = \sum_{i=1}^{r} \left[ \sum_{j=1}^{c} (W[i, j])^2 \right]^{1/2} \tag{5.4}
\]

and

\[
L_{\text{col}}(W) = \sum_{i=1}^{c} \left[ \sum_{j=1}^{r} (W[j, i])^2 \right]^{1/2} \tag{5.5}
\]

For an original weight matrix \(W\), which is divided into \(x \times y\) sub-blocks, the final
training loss for a model with \(l\) layers is defined as:

\[
L = L_D + \lambda \sum_{i=1}^{l} \sum_{j=1}^{x} \sum_{k=1}^{y} (L_{\text{row}}(W[i,j,k]))^i + L_{\text{col}}(W[i,j,k]), \tag{5.6}
\]

where \(\lambda\) is the weight decay, which makes the tradeoff between the loss on training
data \(L_D\) and BSSL regularization penalty. Here, \(W[i,j,k]\) means the sub-block with
index \((j, k)\) in the original weight matrix of layer \(i\).

Essentially, BSSL is a generalization of weight pruning with different granular-
ities. For example, when sub-block shape equals the original weight matrix shape,
BSSL is line-wise pruning. When sub-block shape is \((1, 1)\), it is element-wise prun-
ing. In this work, we propose BSSL to analyze the best pruning granularity for
Transformer models. Specifically, BSSL conforms the following specification to de-
rive better understanding and analyzing target about Transformer properties for
pruning:

- In order to understand which dimension of the weight matrix is more likely to
be pruned, all sub-block of each layer has an identical size. The penalty on
each row and columns of the sub-block is thus the same.
For $W^Q$, $W^K$, and $W^V$ in MHA layer, the dimension of sub-block is equal to head width. In this way, we could understand how many heads are needed for each MHA components and what is the dimension that is required for each head.

- For $W^o$ in MHA layer, we divide them into $2 \times 2$ sub-blocks, which means each sub-block is of shape (256, 256).
- For $W_{ffn1}$ and $W_{ffn2}$ in FFN, we set the sub-block of the same shape as $W^o$. Therefore, $W_{ffn1}$ and $W_{ffn2}$ are divided into $2 \times 8$ sub-blocks and $8 \times 2$ sub-blocks, respectively.

Figure 5.2 depicts the implementation procedure of BSSL training. For a target Transformer model we want to analyze, we first train an original full-size model. Then, we build a new Transformer graph, where each original layer matrix is replaced with tiled sub-blocks. Because the combined shape of all the sub-blocks is the same as the original weight matrix, we initialize all the sub-blocks with the original model values. The input dimension of the original model is split and fed into the blocked model and the output dimension of the blocked model is concatenated so as to be fed into the next layer. In this way, the blocked layers will not affect the other layers in the Transformer. Hence, the blocked Transformer models could achieve the same accuracy as the original model. Last, we apply row-wise and column-wise group lasso on each sub-block and do the training with Block-wise SSL training. During the training, rows and columns of each sub-blocks are expected to become zeros gradually. Finally, we visualize the sparsity properties of the model after BSSL training for a better understanding of the Transformer weights redundancy.
5.3.4 Observation using BSSL Analysis

We apply BSSL analyzing method on Transformer trained with WMT English-German dataset. The weight decay for BSSL regularization term $\lambda = 10^{-3}$ and the learning rate equals $10^{-2}$, which is one-tenth of the original learning rate. In order to better understand the property of each layer and component of Transformer, we visualize the line-wise sparsity of the Transformer model after BSSL analysis. Following previous works [7, 39], the zero weights are determined by a pre-defined threshold, which is set to $10^{-4}$ in our case.

In Figure 5.3, we present the sparsity of 6 encoder layers of Transformer with BSSL. For each encoder layer, there exist $W^Q$, $W^K$, $W^V$, $W^o$ for MHA (left part of Figure 1.2) and $W_{ffn1}$, $W_{ffn2}$ for FFN. The blocks in red mean that such sub-blocks contain no rows or columns that are zeros, or the sparsity is 0% after BSSL pruning. All the other green blocks indicate that they are of certain sparsity, either row-wise,
column-wise, or both after BSSL training. The areas of green denote the non-zero weights in the blocks and the white areas represent the weights that are zero. In other word, the smaller area colored by green in a block, the larger the sparsity for that block. For $W_{ffn2}$, we visualize it by its transpose ($W_{ffn2}^T$) for the clearance of visualization. From Transformer decoder in Figure 5.3, we observe that:

- All the weight matrices are pruned to certain extent for the encoder.
- $W_Q$, $W^K$, $W^V$, and $W_{ffn1}$ should be pruned column-wise while $W^o$ and $W_{ffn2}$ should be pruned row-wise.

Figure 5.4: Sparsity visualization of Transformer decoder with BSSL.
In Figure 5.4, we present the sparsity of 6 decoder layers of Transformer with BSSL. For each decoder layer, there exist $W^Q, W^K, W^V, W^o$ for both self-MHA and encoder-decoder MHA (right part of Figure 1.2) and $W_{ffn1}, W_{ffn2}$ for FFN. From the visualization of Transformer encoder after BSSL in Figure 5.3, we observe that:

- For both self-MHA and encoder-decoder MHA, $W^Q, W^K, W^V$ are pruned to some extent.
- $W^o, W_{ffn1},$ and $W_{ffn2}$ hardly get any sparsity even applied them with line-wise and column-wise group lasso regularization.

### 5.3.5 Conclusion of BSSL Analysis

Here, we summarize the properties derived from BSSL analysis by looking into both the encoder and decoder of Transformer in Figure 5.3 and Figure 5.4.

First, with the help of BSSL analysis, we could find some properties of Transformer models, which have been found in previous works on Transformer understanding [1, 112].

- Some heads in Figure 5.3 and Figure 5.4 are removed entirely, which means that only a subset of heads is important for translation.
- The column-wise sparsity of self-attention MHA in Figure 5.3 is high (53.5% on average) while the sparsity of decoder-encoder MHA in Figure 5.4 is much lower (22% in average), which implies that model prefers to prune encoder self-attention heads. Decoder-encoder attention heads are more important.
- For the self-attention heads in Figure 5.4, the heads in shallower layer (e.g. layer 1, 2) are retained more readily compared with deeper layers (e.g. layer 5, 6). For example, the column-wise sparsities of layer 1 and 2 are 13.3% and 23.3%, respectively while 36.7% and 93% for layer 5 and 6, respectively.
○ For the decoder-encoder attention heads in Figure 5.4, we observe that more heads are retained in deeper layers (e.g. layer 5, 6). For example, the column-wise sparsities of layer 1 and 2 are 68% and 63%, respectively while 10% and 16.7% for layer 5 and 6, respectively.

Besides the properties which have been discovered before, we derive more fine-grained understanding from the sparsity visualization in Figure 5.3 and Figure 5.4 using BSSL:

○ Even if we apply block-wise penalty to Transformer, the model redundancy is still structured in most cases. That is the main reason why we propose line-wise pruning for Transformer.

○ $W^Q$ and $W^K$ are always pruned to the same column-sparsity level while $W^V$ is be pruned to different sparsity levels as $W^Q$ and $W^K$. This fact could be proved from Equation 5.2, where $Q$ and $K$ transpose is multiplied while $V$ is related to the softmax of the multiplication result.

○ Dimension could be further compressed for FFN in Transformer encoder but not for Transformer decoder.

○ All the weights are pruned either row-wise or column-wise because all the sub-blocks want to utilize the whole embedded dimension and only the internal dimension could be pruned. In other words, the pruned dimension is always the internal dimension of each component (MHA and FFN).

Thanks to BSSL analysis, we are able to answer the question in Section 5.3.2 regarding to the properties of Transformer pruning. Those understandings of Transformer model help us deciding the subsequent pruning strategy in Section 5.4.
5.4 Transformer Pruning

5.4.1 Structured Hoyer Square (SHS)

$L1$ regularization is a popular way to achieve weight sparsity [110, 39]. However, $L1$ regularization generally applies the same gradient magnitude during training to all the weights without any consideration of the original weight magnitude. Such behavior is contradictory to our original intention that we want to remove the ‘unimportant’ connections.

In order to penalize more on the weights that ‘need to be pruned’, we leverage the property of Hoyer regularization. Figure 5.5 compares the Hoyer regularizer and the $L1$ regularizer. The left two subplots of Figure 5.5 respectively shows the contour line of 2-D vector of $L1$ and Hoyer regularization [113] and the right two subplots of Figure 5.5 demonstrate their corresponding negative gradients. We could observe that $L1$ regularization method only has a single minimum at the origin. Moreover, $L1$ is proportional to the scale of the variables, making the gradients the same in the whole space except where the variables are zeros. However, Hoyer regularization is scale-invariant, namely, $R(\alpha X) = R(X)$, where Hoyer regularization $R(X)$ equals:

$$R(X) = \frac{\sum |x_i|}{\sqrt{\sum x_i^2}}.$$  \hspace{1cm} (5.7)

The minima structure of Hoyer regularization is thus close to $L0$, which meets the
intention of weight pruning. The aforementioned features of Hoyer regularization make it a more stable and reasonable pruning method than $L_1$ regularization \cite{114}.

In order to align Equation 5.7 of scale $(0 - \sqrt{N})$ with $L_0$ of range $(0 - N)$, Hoyer-Square (HS) regularizer is proposed for element-wise sparsity \cite{7}:

$$HS(W) = \frac{\left(\sum_i |w_i|^2\right)^2}{\sum_i w_i^2}.$$  \hspace{1cm} (5.8)

The HS regularization is also differentiable as well as scale-invariant. It has the same range and minima structure as the $L_0$ norm. Thus, the HS regularizer presents the ability to turn small weights to zeros while protecting and maintaining larger weights.

In this work, we want to prune the model line-wise to realize real speedup-up when executed on mobile devices. In order to realize it, SSL could also be used, which applies $L_1$ on $L_2$ norm of group weights to generate structured sparsity \cite{39}. However, pruning using SSL has its disadvantage regarding the group scale. For example, consider adding SSL regularization on groups of weights: $W_g$, where the $L_2$ norm of this group of weights equals $n_g$. The corresponding gradient for single weight $w_i$ in $W_g$ equals $w_i/n_g$. The gradients largely depend on the scale of original weight value as well as the $L_2$ norm of the group the weight belongs to.

In response to the scale-variant SSL, the scale-invariant HS to Structured Hoyer Square (SHS) is proposed in \cite{7}, which could be expressed as:

$$SHS(W) = \frac{\left(\sum_{g=1}^{G} ||w^{(g)}||_2^2\right)^2}{\sum_{g=1}^{G} ||w^{(g)}||_2^2},$$ \hspace{1cm} (5.9)

where $||w^{(g)}||_2$ means the $L_2$ norm of group of weight $w^{(g)}$. Therefore Equation 5.9 is equivalent to:

$$SHS(W) = \frac{\left(\sum_{g=1}^{G} ||w^{(g)}||_2^2\right)^2}{||W||_2^4},$$ \hspace{1cm} (5.10)

95
The effectiveness of SHS on CNNs has been shown in [7]. In this work, we adopt SHS in Transformer and each group of weights is defined as a row or a column of the layer matrices.

5.4.2 Pruning Strategy

Based on the pruning properties of Transformer derived from BSSL analysis, we apply SHS on our target Transformer model for line-wise pruning. The detailed pruning strategies are as follows:

- The visualization of BSSL in Figure 5.3 shows that the Transformer could achieve high sparsity for all the layer components. Therefore, for Transformer encoder, we prune $W^Q$, $W^K$, $W^V$, $W^{ffn1}$ in column-wise and prune $W^o$ and $W^{ffn2}$ in row-wise.

- The visualization of BSSL in Figure 5.4 shows that the sparsity for FFN and $W^V$ in MHA are very limited. Therefore, for the Transformer decoder, we only prune $W^Q$ and $W^K$ in column-wise.

In the experimental section, we will compare our pruning strategy derived from BSSL analysis with naive pruning strategies to show its advantages. Same as previous works [39, 7], the pruning procedure includes two steps: pruning followed by fine-tuning. In the pruning step, the model is trained with sparsity regularizer (e.g. SSL, SHS), aiming at removing the redundant weights as much as possible. In the fine-tuning step, the trained model from the pruning step is further fine-tuned without sparsity regularizer. The gradients are masked-out for all the zero weights during the fine-tuning step.
5.5 Experiments

5.5.1 Experimental Setup

Framework: We implement TPrune with TensorFlow [30], a widely used deep learning framework for training and testing on various hardware platforms. More specifically, for the training and pruning of the Transformer model, we use tensor2tensor (T2T) [115], which is a TensorFlow library designed for machine learning research. It incorporates many datasets, pre-trained models, and pre-defined hyper-parameters to accelerate research. For the profiling of real-time execution of Transformer on mobile devices, we adopt TFLite Model Benchmark Tool, which is a TensorFlow benchmark tool for embedded software on both IOS and android devices. It converts the original model structure to TFLite format targeting at embedding systems for profiling purposes.

Benchmarks and Settings: For the whole experiments, we use WMT English-German dataset provided by T2T to validate the effectiveness of TPrune. English-German dataset consists of around 32K tokens with shared source-target vocabulary. During the pruning step, the effective batch size is set to 6k tokens and trained for 40k steps. Following the tip in [116], the effective batch size and training steps to achieve promising performance are set larger than that of the pruning step, which is 16k tokens and 800k steps, respectively. The same as [3], We report the BLEU score as the metrics to evaluate model performance and the testing dataset is newstest2014.

Transformer Model: In all the following experiments, we adopt the same model structure as the base Transformer structure in the original work [3]. The base Transformer model include 6 layers for both encoder and decoder with $d_{\text{model}}$ and $d_{\text{embed}}$ equals 512 and $d_{\text{ffn}}$ equals 2048. Because this work is about Transformer model pruning, the final model sparsity is derived by only considering the weights of the main body of Transformer. The embedding matrix is not considered, which is rea-
reasonable because the size of the weight matrix largely depends on the vocabulary size of the source and the target languages. For the fairness of comparison, we start the model pruning from a pre-trained Transformer model provided by T2T framework. The BLEU score of the original model on newstest2014 is tested to be 29.2.

Environment: In order to speed up the training, the models are trained under Linux system with 4 NVIDIA TITAN RTX GPUs in parallel. Therefore, the batch size of each GPU is 4K in order to reach 16k effective batch size. For time profiling on mobile devices, we test the Transformer models on 4 popular Android devices with different hardware performance, which are Nexus 5, Pixel2, Pixel3, and LG G8 ThinQ.

5.5.2 Comparisons of Different Pruning Regularizers

Figure 5.6 shows the BLEU score and corresponding sparsity during the pruning step of two regularizers, which are SHS and SSL, respectively. In Figure 5.6, the weight decay $\lambda$ are set as $10^{-5}$ and $10^{-4}$ for SSL and SHS regularizer, respectively. Because of the comparatively small $\lambda$, the pruned models after $400k$ steps still maintain a good performance. In order to reveal the properties of different pruning regularizers, we only show the metrics change during the pruning step.

As demonstrated in Figure 5.6, under similar BLEU score after $400k$ steps train-
Figure 5.7: Comparison between SSL and SHS regularizers with high pruning sparsity.

ing, the model pruned with SHS achieves higher sparsity (8.16%) than the model pruned with SSL regularizer (2.2%). This is due to the reason that, when the pruning penalty $\lambda$ is small, most of the weights are still non-zeros. In this case, SHS regularizer only penalizes the weights which are close to zeros while SSL regularizer adds a larger penalty on larger weights. Therefore, SHS outperforms SSL regularizer when we prefer high-performance model with smaller sparsity.

Figure 5.7 shows the BLEU score and corresponding sparsity of the models trained with larger weight decay, $5 \times 10^{-5}$ and $10^{-3}$ respectively for SSL and SHS. Because of the higher $\lambda$ setting, the pruned model sparsity reaches around 45% for both regularizers after 400k steps. The corresponding BLEU scores are 21.97 and 21.66, respectively for SSL and SHS regularizer.

Different from the situation in Figure 5.6, the performance of SSL pruned model is similar to the performance of SHS pruned model. Moreover, the larger penalty settings of SSL and SHS regularizers lead to different performance trends during model pruning. On the one hand, the sparsity of SSL pruned model keeps increasing with the decrease BLEU score in the pruning process. On the other hand, the sparsity of SHS pruned model tends to drop dramatically at the beginning steps (e.g. 40k) with the large increase of sparsity and then gradually rescues the performance back in following steps (e.g. 40k-400k). Therefore, SHS regularizer could converge to a
high sparsity with shorter updates while SSL regularizer needs more steps to realize similar sparsity.

5.5.3 Evaluation of Pruning Strategies

In Figure 5.8, we compare different pruning strategies of Transformer to show the effectiveness of BSSL analysis. Besides the decided pruning strategy marked as Strategy, which is mentioned in Section 5.4, we add two line-wise strategies for comparison:

- **Coarse Strategy**: In the pruning strategy derived from BSSL analysis, we treat the encoder and decoder differently. In coarse strategy, we treat them the same as the pruning strategy for the original encoder strategy.

- **Naive**: In naive strategy, we simply apply both row-wise and column-wise penalties to all the target weight matrix. This strategy represents the baseline when we have no understanding of the Transformer structure properties derived from BSSL analysis.

As shown in Figure 5.8, our proposed fined-grained pruning strategy performs best among all the strategies and the naive strategy baseline performs worst. For example, the training loss under 20% sparsity are 1.8, 1.831, and 1.937 for Strategy, Coarse
5.5.4 Evaluation of Layer-wise Sparsity

Figure 5.9 and Figure 5.10 present the layer-wise sparsities under different model sparsities for Transformer encoder and decoder, respectively. Due to the computation logic of Transformer, the following rules always hold:

- \( W^Q \) and \( W^K \) are always of the same column-wise sparsity.
- The column-wise sparsity of \( W^V \) and the row-wise sparsity of \( W^o \) are the same.
- \( W_{ffn1} \)'s column-wise sparsity is the same as \( W_{ffn2} \)'s row-wise sparsity.

**Strategy**, and **Naive**, respectively. This means that our designed pruning strategy is useful for removing the useless regularization penalties during Transformer pruning.
Therefore, we demonstrate the sparsities of them together in Figure 5.9 and Figure 5.10. For Transformer encoder in Figure 5.9, $W^Q$ and $W^K$ are more likely to be pruned at the beginning. For example, when the model sparsity equals 9.76%, the sparsities of $W^Q$ and $W^K$ are 14.65% while the sparsity of $W^{ffn1}$ and $W^{ffn2}$ are 2.08%. With the increase of model sparsity, the sparsity of all target layer types comes to be similar. For example, when the model sparsity equals 45.07%, the sparsities for $W^Q$, $W^K$, $W^V$, $W^O$, and $W^{ffn1}$, $W^{ffn2}$ are 79.07%, 79.52%, and 77.33%, respectively.

Because no regularization penalty is added to $W^V$, $W^O$, $W^{ffn1}$, and $W^{ffn2}$ for Transformer decoder, the sparsities of $W^Q$ and $W^K$ in self-attention and encoder-decoder attention are shown in Figure 5.10. As illustrated in Figure 5.10, the sparsities of $W^Q$ and $W^K$ in encoder-decoder attention are always higher than $W^Q$ and $W^K$ in self attention. This means that encoder-decoder attention is more important than self attention for the Transformer encoder.

When comparing Figure 5.9 with Figure 5.10, we find that, under the model sparsity of 9.76% and 45.07%, the sparsities of $W^Q$ and $W^K$ in Transformer decoder is a little higher than that of encoder. We think this is due to that $W^{ffn1}$, $W^{ffn2}$ are not pruned for Transformer decoder, leaving more pruning space for $W^Q$ and $W^K$.

5.5.5 Evaluation of Speedup on Mobile Devices

In this section, we evaluate the speedup of our pruned Transformer models on mobile devices. For all the profiling experiments on mobile devices, we set the warm-up runs as 5 times and the test runs as 80 times.

Impact of string length

Figure 5.11 presents the execution time on Pixel 3 with different model sparsities and translation string lengths. All the time consumptions in Figure 5.11 are tested
under 4 threads. As shown in Figure 5.11, with the same sparsity, the execution time increases with the string length. For example, for the original transformer model, the time consumptions are 50, 65, 100, and 184 with the string length from 10 to 80. Different from stated in [3], the time consumption on Pixel 3 is not proportional to the square of string length. This is due to that the time complexity of the algorithm could not be directly applied to the time consumption on real mobile devices. Many other factors may also affect the time consumption on mobile devices such as memory bandwidth, on-chip cache size, and memory latency. For the observation in Figure 5.11, we think the memory latency is the main reason that causes a large time consumption even with short string length.
Impact of mobile devices

Figure 5.12 depicts the time consumption of Transformer models on different popular mobile devices. All the time consumptions in Figure 5.12 are tested under 4 threads with string length equals 40. As shown in Figure 5.12, executing Transformer models on different mobile devices largely impacts the time consumption, which holds for all the Transformer models with different sparsities. For example, when executing original models on Nexus 5, Pixel 2, Pixel 3, and LG G8 ThinQ, the corresponding time consumptions are 190ms, 159ms, 100ms, and 70ms, respectively. If we execute the Transformer model pruned to 45% sparsity, the time consumption are 121ms, 81ms, 57ms, and 40ms. Therefore, we could conclude that pruning Transformer model could achieve a similar speedup on mobile devices with various hardware settings.

Impact of multi-thread

Figure 5.13 presents the impact of the multi-thread when executing Transformer models on time consumption. All the time consumptions in Figure 5.13 are tested with string length equals 40 on Pixel 3 (left) and LG G8 ThinQ (right). For the result of both Pixel 3 and LG G8 ThinQ in Figure 5.13, we could observe that pruned models could achieve speedups under different number of threads. For example, when executing Transformer models on Pixel 3, the speedups of the 45% sparsity...
Figure 5.14: Time consumption when executing pruned models of different sparsity.

Transformer model compared with original model are 1.6, 1.5, 1.57, 1.63, and 1.19, respectively, for 1, 2, 4, 8, and 16 threads. Moreover, we observe from Figure 5.13 that simply increasing the number of parallel threads could not guarantee speedup. For Pixel 3, the time consumptions of the original Transformer are 116 ms and 125 ms under 4 and 8 threads. For LG G8 ThinQ, the time consumptions of the original Transformer are 77 ms and 164 ms under 4 and 8 threads. When executing the models under 16 threads, both Pixel 3 and LG G8 ThinQ execute the model with even longer time consumption, which is due to the poor parallelism when dividing a model into too many threads. Therefore, too many threads (e.g. 16) is not suitable when executing a single model on mobile devices. In most cases, 4 parallel threads is a good setting to achieve satisfactory speedup.

**Speedup of pruned models**

Figure 5.14 shows the time consumption when executing pruned models of different sparsity. The blue line in Figure 5.14 shows the results of the models pruned by our proposed SHS regularizer and the yellow line depicts the results of head-wise pruning in previous works [61, 1]. All the time consumptions in Figure 5.14 are tested with string length equals 40 and 4 parallel threads on Pixel 3. For line-wise pruning using SHS regularizer, it achieves 1.65 to 1.92 speedups for the models with the sparsity from 10% to 45%. For head-wise pruning, the speedups are range from 1.07 to 1.75.
Figure 5.15: BLEU score during the fine-tuning procedure.

for the models with the sparsity from 4% to 28%. We observe from Figure 5.14 that head-wise pruning achieves better speedup than line-wise pruning under the same sparsity level due to its larger pruning granularity. However, the sparsity of head-wise pruning is very limited. Even when we remove all the heads in the Transformer model, it only accounts for 28% of the total parameters. In such a case, the model could not even work. While for line-wise pruning, we could achieve 45% parameter decrease with a reasonable BLEU score (around 24.78).

5.5.6 Evaluation of Sparsity-accuracy Trade-off

Figure 5.15 shows how the BLEU score changes during the fine-tuning procedure of TPrune. The BLEU score of the original model is 26.92, marked as the red dash line in Figure 5.15. In Figure 5.15, we train all the models for 800k steps with 16k batch size. We could derive from Figure 5.15 that the pruned model with SHS regularization could quickly converge when the pruned model sparsity is low. For example, the model with sparsity 9.76% restores its BLEU score back in 200k fine-tuning steps to 26.89, only incurring 0.03 BLEU degradation compared to the original model. For the pruned model with sparsity 45.07%, it takes approximately 400k to converge, the BLEU score of which is 24.38. As shown in Figure 5.15, when the model sparsity is below 30%, the fine-tuning step could always restore the performance back and make it close to the original level.
Table 5.1: Summary of our line-wise pruning results, previous head-wise pruning results [1] and line-wise pruning results [2].

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>BLEU</th>
<th>Degradation (%)</th>
<th>Sparsity(%)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMT_EnDe</td>
<td>Baseline</td>
<td>26.92</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Model1</td>
<td>27.14</td>
<td>0</td>
<td>9.76</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>Model2</td>
<td>26.93</td>
<td>0</td>
<td>15.63</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>Model3</td>
<td>26.78</td>
<td>0.52</td>
<td>20.29</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>Model4</td>
<td>26.14</td>
<td>2.9</td>
<td>30.65</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td>Model5</td>
<td>25.94</td>
<td>3.58</td>
<td>34.27</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td>Model6</td>
<td>25.63</td>
<td>4.79</td>
<td>36.93</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>Model7</td>
<td>24.78</td>
<td>7.95</td>
<td>45.07</td>
<td>1.92</td>
</tr>
<tr>
<td>WMT_EnRu</td>
<td>Baseline</td>
<td>29.6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
<td>29.62</td>
<td>0</td>
<td>13.4</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
<td>29.36</td>
<td>0.81</td>
<td>16.7</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
<td>29.06</td>
<td>1.82</td>
<td>21</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>[1]</td>
<td>28.9</td>
<td>2.36</td>
<td>23</td>
<td>1.56</td>
</tr>
<tr>
<td>IWSLT_AraEng</td>
<td>Baseline</td>
<td>27.9</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[2]</td>
<td>24.7</td>
<td>11.47</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>[2]</td>
<td>20.9</td>
<td>25.09</td>
<td>20</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 5.1 summarizes the metrics of TPrune models with the original model as a baseline. Additionally, we also present the results of other two state-of-the-art works with head-wise pruning and line-wise pruning of Transformer model on WMT_EnRu dataset and IWSLT_AraEng, respectively [1, 2]. Because no speedup is tested in [1] and [2], all the speedup results are tested by us on Pixel 3. As shown in Table 5.1, our pruned models could retain good performance with satisfactory speedup on mobile devices. Compared with the baseline model, the pruned model could achieve around 16% sparsity without any degradation in BLEU score with 1.21 speedup on Pixel 3. When the sparsity is increased to 20.29%, the BLEU score comes to 26.78, which is about 99.48% of the original performance. The BLEU score still achieves 26.14 with 30.65% sparsity, incurring 2.9% BLEU score decrease compared with the baseline. If we further remove the weights to 45% sparsity to achieve 1.92 speedup, the BLEU score comes to 24.78, which is 92.05% of the original performance. Similar to our
results, for the result of head-wise Transformer pruning in [1], the model could also get promising performance under low sparsity. For example, in [1], 1.22 speedup is achieved with 13.4% model sparsity with no performance degradation. However, when the model sparsity is increased to 23% with 1.56 speedup, the BLEU score is 28.9, suffering from 2.36% BLEU degradation. What is worse is that the model could not be further pruned because of the limited heads in Transformer. When comparing with [2], another line-wise pruning work, TPrune still demonstrates better sparsity-accuracy tradeoff. For example, the model performance in [2] declines from 27.9 to 20.9 in order to achieve 20% sparsity and 1.25 speedup. While in TPrune, only 0.14 BLEU drop is incurred with 20.29% sparsity.

5.6 Conclusion

In this work, we propose TPrune, which includes a Transformer model analysis method Block-wise Structure Sparsity Learning (BSSL) and a line-wise pruning method using Structured Hoyer Square (SHS). In BSSL, we utilize structured sparsity learning on sub-blocks of Transformer weight matrix to understand the model properties of Transformer. Then, based on the analysis derived from BSSL, we design the pruning strategy. Last, we propose SHS regularizer to prune the Transformer model in either row-wise or column-wise for different Transformer components. In a nutshell, TPrune innovatively analyzes the Transformer model properties as well as accelerates the model execution on mobile devices by pruning method. Experimental results show that TPrune achieves satisfactory speedup with no or marginal performance degradation.
Conclusion

In this dissertation, I first study the efficiency neural network based systems on mobile and cloud platforms. On multiple mobile devices under local network, we propose and design the systems for distributed testing (MoDNN) and distributed training (AdaLearner). These two systems enable the user to perform efficient inference and learning of neural networks in local distributed mobiles. In such a case, the sensitive user data does not need to be uploaded to outside server. For mobile testing, I wish the future researcher could focus more on the applications that leverage neural networks to solve the real-world challenges and promote life happiness such as AR/VR games, real-time simultaneous localization and mapping, and causal inference applications. For mobile training, in order to maximize the utilization of mobile resources and heterogeneous data, more studies could be conducted on the topic of Federated Learning, Meta Learning, and Life-long Learning. Then, we study the application of video detection and propose MobiEye to accelerate the inference latency of an existing video detection framework called DFF. We emphasize the importance of real-time video detection applications because it is very important in the field of unmanned vehicle, surveillance system, etc. Last, we study the compression
of Transformer models, which is a state-of-the-art model structure in the language-related applications. We propose TPrune as the pruning framework for Transformer models. The compression of Transformer is an important field to enable the future demand of on-device translation, speech recognition, and text understanding on mobile platforms.
Bibliography


Biography

Jiachen Mao is a Ph.D. candidate at Electrical and Computer Engineering at CEI lab of Duke University. His research is efficient neural network based systems on mobile and cloud platforms. He received his Master’s degree from electrical and Computer Engineering school of the University of Pittsburgh and his Bachelor’s degree from the software engineering school of Shanghai Jiao Tong University, Shanghai, China.

During his Ph.D. study, he worked as a Research Intern at Adobe, digital experience department for two consecutive summers. He has authored one Best Paper Award in DATE [31]. He has authored or co-authored papers in many conferences, including HPCA [49], ICTAI [57], DAC [45, 117], ICCAD [76, 4, 46], DATE [31], and ASP-DAC [50, 48], etc. After graduation, Jiachen will start his position as a full-time research scientist of the machine learning team, at Facebook.