COMPILATION AND OPTIMIZATION TECHNIQUES FOR MACHINE LEARNING WORKLOADS

Byung Hoon Ahn[1]
Department of Computer Science and Engineering
University of California, San Diego
bhahn@eng.ucsd.edu

ABSTRACT

Similar to how performance improvement of general-purpose computing has enabled diverse applications, achieving faster execution of machine learning algorithms can foster more pervasive use of machine learning in various application domains. While there exist some hand-optimized libraries to enhance efficiency in a narrow range of hardware, there is an increasing need to bring machine learning to various devices ranging from cloud to edge. As such, conventional compilation stacks require revision to enable higher levels of performance and efficiency among a wide range of devices. Recently, multiple new software libraries and compilers have been introduced to meet the above needs. On the other hand, there are some algorithmic characteristics that we can leverage to optimize the machine learning workloads for more efficiency. For example, by exploiting the intrinsic error tolerance of machine learning models to near-threshold, through quantization and pruning, we can improve the efficiency of the machine learning algorithms. To this end, this report surveys these two directions: *compilation* and *optimization* approaches for machine learning workloads.

1 INTRODUCTION

Deep learning, especially Deep Neural Networks (DNNs), have pushed the boundaries in image classification (Krizhevsky et al., 2012; Simonyan & Zisserman, 2015; Szegedy et al., 2015; He et al., 2016), natural language processing (Vaswani et al., 2017; Devlin et al., 2018), automatic speech recognition, and autonomous decision making (Mnih et al., 2015). To cope with the enormous computational intensity of DNNs and the emergence of dark silicon phenomenon (Esmaeilzadeh et al., 2011), the community developed domain-specific architectures that can execute the emerging workloads (Chen et al., 2016; 2014; Han et al., 2016a; Judd et al., 2016; Gao et al., 2017; Parashar et al., 2017; Sharma et al., 2018). For example, these deep learning accelerators such as TPU (Jouppi et al., 2017) and Brainwave (Fowers et al., 2018) have permeated into the servers in large-scale data centers and in edge devices (NVIDIA, b). Concurrently, some software libraries such as cuDNN, TensorRT, and MKL have been integrated as backends for a variety of programming environments such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). However, computer architectures and compilers build an intimate relationship where one requires another for maximized utilization and performance. As such, to cope with the computational requirements of the deep learning models as well as its volatile topology, it calls for developing automated compilation frameworks that can optimize the execution of deep models. For example, TensorComprehensions (Vasilache et al., 2018), Glow (Rotem et al., 2018), and TVM (Chen et al., 2018a) are specifically designed to optimize deep learning workloads through optimization and graph transformations.

On the algorithm front, however, quantization (Wang et al., 2019; Esser et al., 2020) and pruning (Zhu & Gupta, 2018; Anwar et al., 2017) have been highly sought for as it provides many benefits in terms of optimizing the algorithm. For example, Deep Compression (Han et al., 2016b) shows that this dimension of optimization can bring over $30\times$ reduction in storage requirements and $3\times$ speedup and energy efficiency. The community is also exploring new dimensions to optimize the execution of deep learning algorithms such as activation compression (Jain et al., 2018). To this end,

[1] Second year PhD student at Alternative Computing Technologies (ACT) Lab. (Prof. Hadi Esmaeilzadeh).

this report summarizes the community's effort to *compile* and *optimize* machine learning workloads (esp. DNNs) and the remaining challenges, then it also describes some interesting directions for future investigation. Section 1 describes the basics of machine learning and DNNs with some of the prominent applications. Then, Section 2 and Section 3 enumerates the recent works on compilation and optimization of DNNs, respectively. Lastly, Section 4 describes directions for future research.

2 Compilation for Deep Neural Networks

As DNNs have became a integral feature for various applications, enhancing programmability to expedite the algorithm development as well as improving the speed of execution are important tasks. As such, TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019) has been developed on top of software libraries such as cuDNN (NVIDIA, a), TensorRT (NVIDIA, 2017), and MKL (Intel) to improve both programmability and performance. However, the compute-intensive nature of DNNs combined with the volatility of its operations call for developing ways to further optimize DNN execution on variety of hardware platforms (not limited to CPUs or GPUs). Recent works such as TVM (Chen et al., 2018a) and TensorComprehensions (Vasilache et al., 2018) show that optimizing compilation over the deep learning workloads provide significant benefits in terms of execution speed on a wide range of devices. This section surveys different categories of programming interface to develop DNNs and the recent innovations in optimizing the execution.

2.1 Programming Interface

As the deep learning models became more complex, many frameworks to speed up development of these networks have been proposed. These frameworks are provided in form of libraries that are expressive enough to cover variety of deep learning models and flexible enough to quickly support new operations. These libraries or programming interfaces are provided as internal Domain Specific Languages (DSLs) (Fowler, 2010), and are embedded in general purpose languages such as C++ and Python. Then, these DSLs are powered by the backend libraries that efficiently perform General Matrix to Matrix Multiplication (GEMM) which is part of the Basic Linear Algebra Subprograms (BLAS) (Lawson et al., 1979). For example, cuDNN (NVIDIA, a), TensorRT (NVIDIA, 2017), and MKL (Intel) are good examples of these libraries.

The programming interfaces to design DNNs are largely categorized into two families depending on their execution models: (1) first build a symbolic graph from the program then execute, and (2) directly execute the program imperatively. This subsection explains these program interfaces in detail and highlight the representative deep learning frameworks.

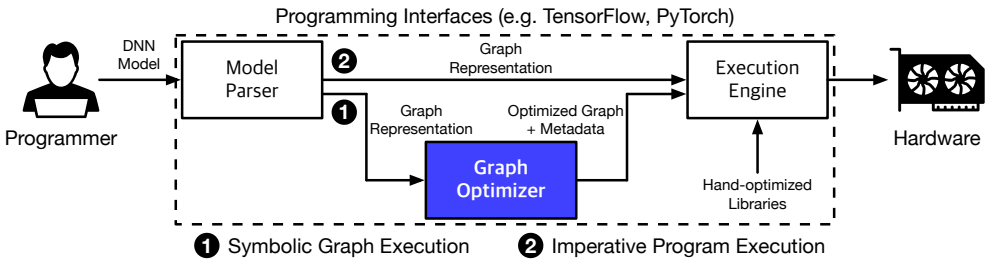


Figure 1: Illustration of high-level structure of different class of programming interfaces: *Symbolic Graph Execution* and *Imperative Program Execution*.

Symbolic graph execution. Frameworks such as TensorFlow (Abadi et al., 2016), Caffe2 (Facebook), CNTK (Seide & Agarwal, 2016), and MXNet (Chen et al., 2015) first builds an internal representation (dataflow graph) of the DNN model defined by the programmer. The nodes in the graph are the operations (e.g., convolution, pooling, activations, etc.) and the edges denote the flow of data. By implementing such representation internally, it allows the framework to exploit various levels of parallelism while scheduling the nodes or enables various well known compiler optimization such as node fusion. Furthermore, this form of execution makes it more suited to implement

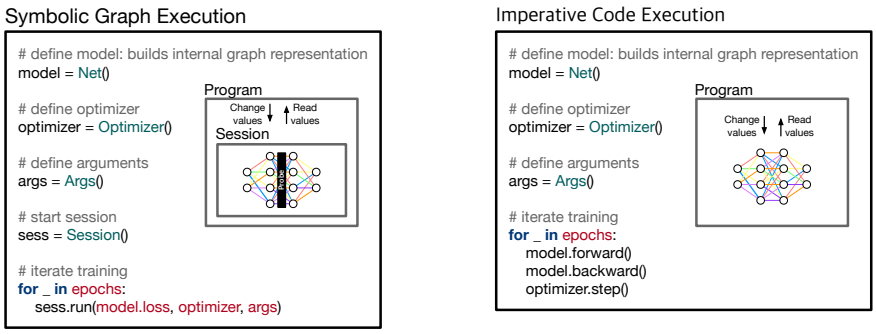


Figure 2: Code example of *Symbolic Graph Execution* and *Imperative Program Execution*. Symbolic graph execution requires *probe* to read or modify values while it can be done very easily using imperative program execution.

various optimized backends for wide range of devices. Figure 2 illustrates the workflow of this class of frameworks. As illustrated in the figure, one important component of symbolic graph execution is the *Graph Optimizer*. Since this graph optimizer is fed with a representation of the entire graph of the input DNN model, it is able to not only perform various graph optimization but also take advantage of target specific techniques. For example, TensorFlow (Abadi et al., 2016) takes advantage of the graph representations to assign, distribute, schedule, or even interleave the computation and communication. Another upside of symbolic graph execution is that it allows more straightforward way of saving or visualizing the models. On the other hand, in imperative code execution (while there are tricks to enable this), however, lack of concrete internal representation makes it more difficult to save or visualize the models.

Imperative program execution. On the contrary, Figure 2 illustrates the workflow of another class of frameworks: imperative execution. While this class of deep learning frameworks includes a similar language constructs, internal execution of the graphs differ significantly in its philosophy. Unlike symbolic graph execution which takes advantage of various graph optimizations, the imperative program execution directly executes the computations. Therefore, these frameworks makes it harder to optimize the code. For example, PyTorch (Paszke et al., 2019) or Eager Mode of TensorFlow (Abadi et al., 2016) takes this approach, and these frameworks are relatively slower compared to the symbolic frameworks mentioned above. However, although these frameworks give up a little on the performance in terms of speed, their easy-to-use interface has been gathering large population of users from academia. He (2019) shows that PyTorch has been attracting many researchers, and now even surpassed that of TensorFlow. This trend can be attributed to the easy interface of PyTorch to try wacky ideas that require modification of the gradients, weights, or activations.

Recent and future directions. As these different frameworks each implements different format to save and deploy DNN models, ONNX is an initiative that aims to build an open standard for machine learning models. ONNX is now supported by various deep learning frameworks and has even permeated into deep learning pipelines used internally at Samsung, Microsoft, and Qualcomm[1].

Another interesting direction is to develop hybrid of the two aforementioned directions in developing a hybrid approach that has benefits not only in the user interface and straightforward execution but also speed. For example, Janus (Jeong et al., 2019) aims to achieve both performance of the symbolic execution and the user-friendly programming of the imperative execution.

One more inspiring dimension of research is using the insights that we accrued while developing these programming interfaces to different domains such as robotics, digital signal processing, and etc. Also, one looming direction of research is on developing polymorphic languages that can target subsets of these different application domains, enabling easy programming and acceleration of cross-domain applications.

[1]According to personal communications (Lee; Saarikivi), these companies have employed ONNX as input interface for their in-house compilation infrastructures.

Table 1: Summary of supported programming interfaces and target hardware for frontend compilers.

	TVM	TC	Glow	nGraph	XLA
Supported programming interfaces					
TensorFlow	✓	✗	✗	✓	✓
PyTorch	✓	✓	✓	✗	✓
MXNet	✓	✗	✗	✗	✗
Caffe2	✓	✓	✓	✗	✗
ONNX	✓	✗	✓	✓	✗
Supported target hardware					
CPU	✓	✓	✓	✓	✓
GPU	✓	✓[3]	✓	✓	✓
FPGA	✓	✗	✗	✗	✗
TPU	✗	✗	✗	✗	✓
Custom	✓	✗	✓	✗	✓

Returning to programming for deep learning, while easy implementation and efficient training is of the utmost importance to the neural architects, faster inference is an important problem when it comes to deploying these deep learning models. As such, an imperative problem is how to deploy the models trained on these frameworks to various devices. This challenge brings us to the next section on compilation.

2.2 COMPILATION

While the aforementioned programming interfaces fostered fast development of deep learning models, increasing complexity and the volatility of the models make it difficult to optimize its execution using the these frameworks. For example, TensorFlow (Abadi et al., 2016), while it has been extended with XLA (Google, b) to provide higher performance on CPUs and GPUs, mainly relies on hand-optimized General Matrix Multiply (GeMM) libraries. While these are practical solutions to execute some deep models, it is not flexible enough to consistently provide high performance over operations other than 2D convolution and fully connected layers. Therefore, it calls for developing automated compilation frameworks that can provide high performance for novel operations to cope with an increasing demand for speed in this unprecedented rate of innovation (LeCun, 2019).

Frontend compiler. Frontend compilers[2] in Figure 3 take a deep learning model from the existing deep learning programing interfaces as input and parses them into Intermediate Representations (IR). These IR enables various optimizations such as operator fusion, layout transformation, and various other simplification that can reduce computation of the models. These optimization passes are usually categorized into two distinct classes, where one is independent of the target hardware, whereas the other is takes advantage of the architectural characteristics of the target device to maximize performance. Representatively, TVM (Chen et al., 2018a), TensorComprehensions (Vasilache et al., 2018), Glow (Rotem et al., 2018), nGraph (Cyphers et al., 2018), and XLA (Google, b) have developed infrastructures that enables the translation (from above programming interfaces to IR) and also some common optimizations. Table 1 summarizes the supported programming interfaces and the target hardware for these frontend compilers.

Optimizing compilation. Optimizing compilers (Kennedy & Allen, 2001) refers to compilers try to maximize the gains of a computer program. In current deep learning compilers, optimizing compilers take a black-box approach and use hardware measurements to configure the optimization based on a measure of fitness f of each solution. For example, the compiler develops a template τ with configurable knobs $\Theta = (\theta_1, \theta_2, ..., \theta_n)$ from tiling, loop unrolling, and thread bindings. The

[2]While there may be different ways of classifying different parts of the compiler, this report separates optimizing compiler as the backend compiler while categorizing all other parts as the frontend. This is to emphasize the optimizing compilation process that is an important research topic, following the practice in Ahn et al. (2020b). Another way of classifying the components is illustrated in Li et al. (2020).

[3]TensorComprehensions (Vasilache et al., 2018) do not support AMD GPUs.

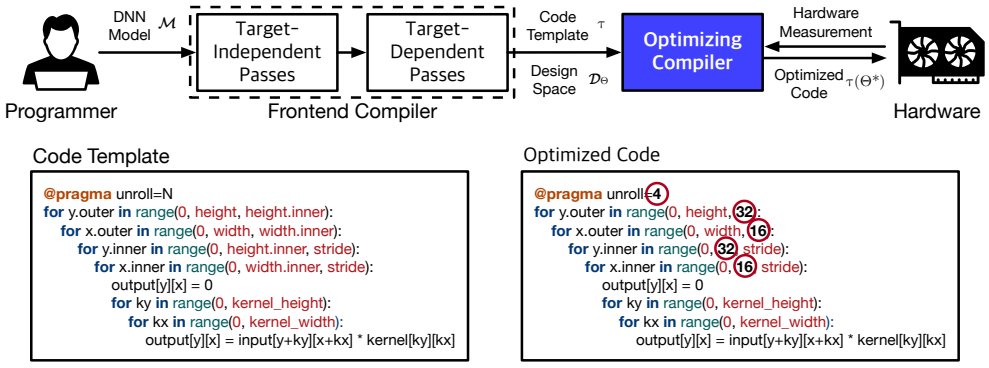


Figure 3: Overview optimizing compilation where the illustrated code template is fed into the optimizing compiler and is returned with the optimized configurations (N, height.inner, width.inner) inlined (circled red) to constitute the optimized code. [Ahn et al. (2020b)]

compiler then makes iterations of hardware measurements to find an optimal combination of the knobs in the design space $\mathcal{D}_\Theta$ from Θ:

$$\Theta^* = \underset{\Theta}{\operatorname{argmax}}\, f(\tau(\Theta)), \qquad \text{for } \Theta \in \mathcal{D}_\Theta. \tag{1}$$

TensorComprehensions (Vasilache et al., 2018) and TVM (Chen et al., 2018a) employ random search algorithms or genetic algorithms to explore the vast design space. However, as these methods may take exorbitant amount of time to converge, AutoTVM (Chen et al., 2018b) utilizes XGBoost (Chen & Guestrin, 2016) cost model to approximate the search space and use simulated annealing on top of the cost model to find high performant configuration quickly. One limitation of this direction is that it neglects the data layout transformations that has to happen between different operations in the graph. Therefore, recent work (Liu et al., 2019b) has implemented a graph tuner that utilizes dynamic programming and Partitioned Boolean Quadratic Programming (PBQP) to address this issue. Another limitation that is yet to be addressed has to do with system level scheduling, where the scheduling of the network should not be isolated but done in relation to other workloads that run simultaneously.

Recent and future directions. Optimizing compilation that finds optimal configuration for tiling, loop unrolling, thread management, etc. constitutes an enormous search space. Therefore, naive exhaustive search is impractical to enable timely deployment of the DNN models that may require optimization on heterogeneous devices. In order to mitigate this problem, AutoTVM (Chen et al., 2018b) utilizes XGBoost (Chen & Guestrin, 2016) cost model to approximate the search space and use simulated annealing on top of the cost model to find high performant configuration quickly. However, Ahn et al. (2020b) shows that this naive reliance on simulated annealing with rather weak guarantees and the naive sampling scheme implemented in these state-of-the-art optimizing compilers are not fast enough for practical use. As such, Chameleon (Ahn et al., 2020b) implements a reinforcement learning-based exploration that learns to navigate the search space and an adaptive solution using clustering to sample representative configurations from the exploration.

Another important direction of research is the development of generic Intermediate Representations (IR) to improve existing compilers, enable better compilation for heterogeneous hardware, reduce the engineering cost of building new compiler, and provide common low-level representation for different compilers. As LLVM (Lattner & Adve, 2004) gathered big community to develop various compiler techniques that were difficult to implement on GCC (Stallman & DeveloperCommunity, 2009), providing higher performance on wide range of hardware, such IR will empower deep learning compilers to achieve higher performance. Even TVM (Chen et al., 2018a), an open-source infrastructure integrated to deep learning pipelines in Amazon (Amazon), Xilinx, and Qualcomm[4], plans on implementing more graph optimization techniques and develop more graph optimization features on their current IR: Relay.

[4]According to personal communications (Lee), TVM is being used as the backbone of their in-house compilation infrastructures.

3 Optimization for Deep Neural Networks

Initial efforts regarding the Deep Neural Networks (DNNs) focused on achieving state-of-the-art performance on different tasks such as computer vision, speech recognition, natural language processing, and autonomous decision making. While these efforts took advantage of the increasing computational power which such levels are exclusively available in server-class computing resources, efficiency, connectivity, and privacy concerns are driving the migration of compute from cloud to edge. As such, various techniques to optimize DNN execution has been proposed. These approaches can be generally categorized into (1) efficient model design and (2) model compression.

3.1 Efficient Model Design

On the different end of the spectrum, there are methods that focus on designing efficient models in the first place. For example, this direction of studies focus either on devising and leveraging compute units that are computationally more efficient while providing similar if not better utility. Also, some works focus on developing efficient neural architectures building on (AutoML).

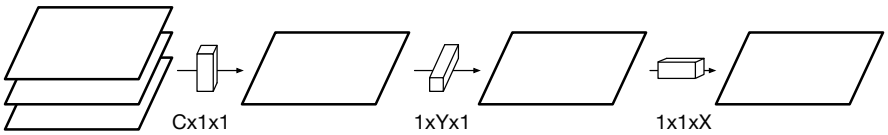


Figure 4: Extreme form of separable convolution proposed in Jin et al. (2014).

Operations for efficient computation. Some approaches have been proposed as building blocks to develop efficient networks. One of the most widely used approaches is the depthwise separable convolutions, which especially gained a lot of traction after being employed in the MobileNets (Howard et al., 2017). This method that was initially proposed in Sifre & Mallat (2014) aims to factorize the common convolution operation into depthwise convolution and pointwise convolution. By reducing redundancies among the weights, separable convolutions are able to provide similar performance with fraction of weights. Jin et al. (2014) illustrated in Figure 6 goes one step further and even decomposes the depthwise convolution into the row and column vectors. These operations are currently the building blocks for networks that show state-of-the-art performance with regard to efficiency (Tan & Le, 2019; Real et al., 2019; Xie et al., 2019; Howard et al., 2017).

However, recently, many research usually utilize the same operations that have been developed and focus on how to compose them in a more efficient way. For example, in the recurrent networks, one of the most widely used cell is Long Short-Term memory (LSTM) (Hochreiter & Schmidhuber, 1997). However, to achieve higher efficiency, Chung et al. (2014) proposed Gated Recurrent Units (GRU), which the main differences are the number of gates and its internal connections. Likewise, main advances in designing efficient operations seem to be focused on applying the previously learned insights to emerging domains Liu et al. (2019c)

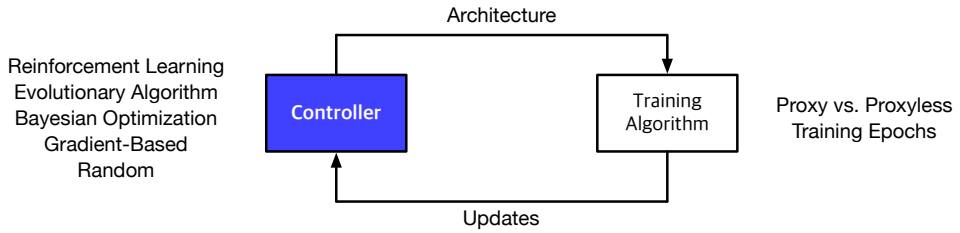
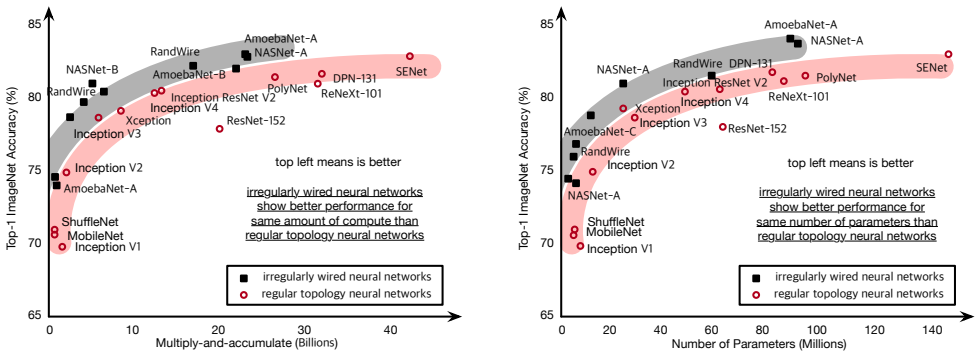


Figure 6: Illustration of neural architecture search workflow, and different options for the controller and the training algorithm.

Neural Architecture Search (NAS). Many of the works on Automating Machine Learning (AutoML) that leverages Neural Architecture Search (NAS) (Zoph & Le, 2017; Cortes et al., 2017; Zoph



(a) ImageNet accuracy vs. number of multiply-and-accumulate operations.

(b) ImageNet accuracy vs. number of parameters.

Figure 5: Networks from Neural Architecture Search (NAS) show higher efficiency in terms of accuracy per MAC operation or parameter. [Ahn et al. (2020a)]

et al., 2018; Liu et al., 2019a; Cai et al., 2019; Real et al., 2019) demonstrates significant improvement in classification performance. Furthermore, recent paper (Ahn et al., 2020a) shows that some of these networks emit models with numerous irregular wirings, and suggests that these networks can even adapt to some of the constraints of the hardware (e.g., memory capacity, bandwidth, number of functional units). Figure 5 plots the accuracy of different models given their computation and the number of parameters. The figure clearly shows that the Pareto frontier of the networks from NAS are better than the hand designed models, and this indicates that the efficiency in terms of accuracy given fixed resources are better with such networks.

Recent and future directions. Recent interest on AutoML using NAS to develop models with higher levels of efficiency has introduced an emerging class of neural architecture with numerous irregular wirings. These models with irregular topologies of the network, deviating from simple models that warrant streamlined execution of the nodes, introduces a new dimension of optimization where different schedule of the nodes show varying peak activation memory footprint. Chen et al. (2016) and Sharma et al. (2018) illustrates that memory footprint constitutes large fraction of the power consumption in deep learning accelerators. Therefore, reducing the peak memory footprint of emerging class of network, that yields state-of-the-art performance on resource constrained edge devices, provides new dimension of optimization for compilers. However, current solutions including TensorFlow Lite (Google, a), a widely used framework for edge devices, are oblivious to this imperative dimension of optimization, thus render themselves ineffective for the emerging class of networks. To address this imminent concern in the industry, Ahn et al. (2020a) introduced a memory-aware scheduling based on dynamic programming to find memory-optimal schedule considering the peak activation memory footprint, and devised a graph rewriting technique that can even reduce the memory footprint of the model without affecting the mathematical integrity of the model.

3.2 Model Compression

There have been large body works to compress deep learning models. For example, pruning that skips some of the computation within neural networks and quantization that reduces the precision of computation to enable more efficient execution on hardware. This subsection delves into these two different directions of optimization that are being used for model compression.

Pruning (Sparsification). Initial works on pruning (LeCun et al., 1990) pruned operations using diagonal Hessian approximation. However, as we transitioned to deeper and wider networks with tremendous amount of operations, these methods became impractical due to the computational complexity of calculating the inverse of the Hessian. Therefore, many of the recent works (Han et al., 2016b; See et al., 2016; Narang et al., 2017) rely on magnitude-based weight pruning, which are much more efficient and possible to scale. As pruning the operations may degrade the performance of the neural networks, many works go through fine-tuning steps to minimize the performance loss.

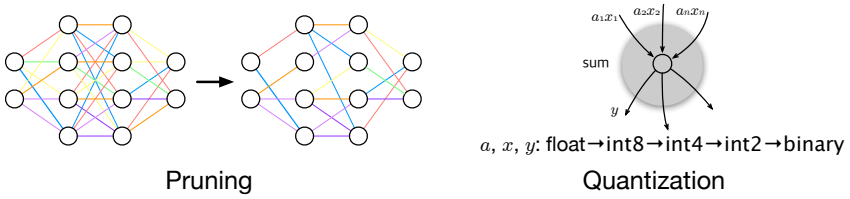


Figure 7: Illustration of pruning and quantization. Pruning removes many edges which is equivalent to setting weights to zero. Quantization reduces the precision of each operation (edge: multiply, node: sum) to sub-byte.

Some of the most recent works (Frankle & Carbin, 2019; Ramanujan et al., 2020) are diving deeper into the value of initialization and the connections itself. For example, Frankle & Carbin (2019) shows that it may even be possible to train sparse networks (subnetworks from a large model) with comparable performance from scratch. On the other hand, Ramanujan et al. (2020) demonstrates that with some intelligent algorithms, we can identify subnetworks within randomly weighted networks which achieve impressive performance with absolutely no training.

Quantization. Quantization refers to reducing the precision of operations in deep neural networks while maintaining the functionality of the computation. For example, making deep neural network models out of 8 bit integer operations instead of using 32 bit floating point operations may yield similar performance. There have been extensive works on compressing deep neural networks using quantization. For example, Han et al. (2016b) quantized the network weights with pruning and Huffman coding to achieve 35x reduction in memory requirement. On the other hand, Courbariaux et al. (2015) and Rastegari et al. (2016) binarized the network. Myriad of works () then focused on the middle ground that yields shoulder-to-shoulder performance to the full precision models.

In order to maximize the gains from these techniques, recent works combine them with Knowledge Distillation (Hinton et al., 2015; Mishra & Marr, 2018) or gradient-based approaches (Esser et al., 2020; Gong et al., 2019). Also, as mixed precision quantization have become a de facto standard for deployment to hardware, automated approaches that utilize reinforcement learning to learn these bit-widths (Wang et al., 2019; Elthakeb et al., 2018) have been developed. Importantly, however, most of the research in quantization focused on image related tasks, and there have been relatively less works done to quantize Automatic Speech Recognition and Robotics workloads.

Recent and future directions. Culmination of these optimization techniques are well demonstrated in various competitions such as Low-Power Image Recognition Challenge (Gauen et al., 2017) and Visual Wake Words Challenge (Chowdhery et al., 2019), which aim to accelerate the networks using both hardware and software optimizations. Both industry and academia compete for lowest possible number of operations, total number of bits for weights, and energy. To this end, these methods are being used to reduce the number of operations and enable more efficient deployment of these networks to embedded systems and domain-specific accelerators (Chen et al., 2016; 2014; Han et al., 2016a; Judd et al., 2016; Gao et al., 2017; Parashar et al., 2017; Sharma et al., 2018).

Overall, excitement in using pruning and quantization to optimize the pre-trained networks have now enabled us to set 90% of to zero and achieve similar performance with very few bits of precision. While each of these works have been explored much in depth and some works (Han et al., 2016b) have explored their combination in practice, we are far from knowing the exact relations of these two different dimensions of optimization. For example, theoretical analysis and large-scale experiments to discover the relationships of these different approaches may act as a good compass for both research and development and may unravel new directions of research.

4 Future Directions

Machine Learning and Systems. One interesting direction of research that have emerged over the years is the intersection of machine learning and systems (Ratner et al., 2019), which explore how one can be combined with another. For example, large body of works (Chen et al.,

2016; 2014; Han et al., 2016a; Judd et al., 2016; Gao et al., 2017; Parashar et al., 2017; Sharma et al., 2018; Abadi et al., 2016; Paszke et al., 2019; Chen et al., 2018a) that focus on developing a specialized architectures and compilation infrastructures to accelerate the execution of machine learning workloads would be classified as *systems for machine learning*. On the other hand, another side of the spectrum is about *machine learning for systems*, which explores application of various learning techniques to systems. For example, machine learning algorithms have now permeated into compiler infrastructures to provide better performance estimations (Chen et al., 2018b; Mendis et al., 2019) or better features (Haj-Ali et al., 2020). Likewise, developing a *AI-enabled compiler infrastructure for machine learning workloads* embedded inside, and finding the right *balance between the classical methods and machine learning methods* would be interesting problems to solve in the future.

Compilers for cross-domain applications. As illustrated in this report, there has been significant attention to building domain-specific programming stacks and optimization techniques to accelerate DNNs. However, the essential ingredients to enable end-to-end applications such as autonomous driving and drone delivery are not limited to just DNNs. For example, developing autonomous driving pipeline comprise of algorithms across multiple domains including digital signal processing, traditional computer vision, deep learning, and even control algorithms. Therefore, compilation stacks that have narrow focus on just deep learning workloads may render themselves rather myopic and impractical. Nevertheless, developing frameworks for deep learning has granted us with invaluable lessons on some key considerations to develop new infrastructures, and As such, building on the experiences of developing deep learning frameworks, compilers, and optimization techniques, an important direction to pursue would be developing *compilers for cross-domain applications*.

5 Conclusion

Immense computational intensity of machine learning workloads have brought about a major paradigm shift in computing. As general-purpose compute stacks come short in providing the required performance to enable machine learning, a domain-specific stack for machine learning has been introduced to bridge the performance gap. Concurrently, various optimization schemes have been introduced to maximize the efficiency of executing these compute intensive workloads on wide range of devices. This report explored the major advances in the *compilation* and *optimization* approaches for machine learning that has enabled the pervasive use of learning for various applications. Survey of the approaches also suggests various interesting future research directions including the *development of AI-assisted compilation stacks for cross-domain applications*.

Acknowledgement

I thank Chung-Kuan Cheng, Sorin Lerner, and Dean Tullsen for their insightful comments. I thank people in the Alternative Computing Technologies (ACT) Lab. for the fruitful discussions and feedbacks on this report.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, 2020a.

Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020b. URL `https://openreview.net/forum?id=rygG4AVFvH`.

Amazon. Amazon SageMaker: Machine learning for every developer and data scientist. URL `https://aws.amazon.com/sagemaker/`.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *JETC*, 2017.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. URL https://openreview.net/forum?id=HylVB3AqYm.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv*, 2015. URL https://arxiv.org/pdf/1512.01274.pdf.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018b.

Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC*, 2016.

Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset. *arXiv*, 2019. URL https://arxiv.org/pdf/1906.05721.pdf.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *NIPS Deep Learning and Representation Learning Workshop*, 2014.

Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. AdaNet: Adaptive structural learning of artificial neural networks. In *ICML*, 2017.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *arXiv*, 2018. URL https://arxiv.org/pdf/1801.08058.pdf.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2018.

Ahmed T Elthakeb, Prannoy Pilligundla, Amir Yazdanbakhsh, Sean Kinzer, and Hadi Esmaeilzadeh. ReLeQ: A reinforcement learning approach for deep quantization of neural networks. *arXiv*, 2018. URL https://arxiv.org/pdf/1811.01704.pdf.

Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. Learned step size quantization. In *ICLR*, 2020. URL https://openreview.net/forum?id=rkgO66VKDS.

Facebook. Caffe2. URL https://caffe2.ai.

Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *ICLR*, 2019. URL https://openreview.net/forum?id=rJl-b3RcF7.

Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.

Kent Gauen, Rohit Rangan, Anup Mohan, Yung-Hsiang Lu, Wei Liu, and Alexander C Berg. Low-power image recognition challenge. In *ASP-DAC*, 2017.

Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *ICCV*, 2019.

Google. TensorFlow Lite., a. URL https://www.tensorflow.org/mobile/tflite.

Google. XLA: Optimizing compiler for machine learning., b. URL https://www.tensorflow.org/xla.

Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *CGO*, 2020.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: efficient inference engine on compressed deep neural network. In *ISCA*, 2016a.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016b.

Horace He. PyTorch vs. TensorFlow, 2019. URL https://chillee.github.io/pytorch-vs-tensorflow/.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv*, 2015. URL https://arxiv.org/pdf/1503.02531.pdf.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv*, 2017. URL https://arxiv.org/pdf/1704.04861.pdf.

Intel. MKL. URL https://software.intel.com/en-us/mkl.

Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ISCA*, 2018.

Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *NSDI*, 2019.

Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. Flattened convolutional neural networks for feedforward acceleration. *arXiv*, 2014. URL https://arxiv.org/pdf/1412.5474.pdf.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *TOMS*, 1979.

Yann LeCun. Deep learning hardware: Past, present, and future. In *ISSCC*, 2019.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *NIPS*, 1990.

Jinwon Lee. personal communication.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The deep learning compiler: A comprehensive survey. *arXiv*, 2020. URL https://arxiv.org/pdf/2002.03794.pdf.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *ICLR*, 2019a. URL https://openreview.net/forum?id=S1eYHoC5FX.

Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *USENIX ATC*, 2019b.

Zhijian Liu, Haotian Tang, Yujun Lin, and Song Han. Point-voxel cnn for efficient 3d deep learning. In *NeurIPS*, 2019c.

Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *ICML*, 2019.

Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *ICLR*, 2018. URL https://openreview.net/forum?id=B1ae1lZRb.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks. *arXiv*, 2017. URL https://arxiv.org/pdf/1704.05119.pdf.

NVIDIA. cuDNN., a. URL https://developer.nvidia.com/cudnn.

NVIDIA. NVDLA., b. URL http://nvdla.org.

NVIDIA. TensorRT: Programmable inference accelerator., 2017. URL https://developer.nvidia.com/tensorrt.

ONNX. ONNX:open neural network exchange. URL https://onnx.ai.

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? *arXiv*, 2020. URL https://arxiv.org/pdf/1911.13299.pdf.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

Alexander Ratner, Dan Alistarh, Gustavo Alonso, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Eric Chung, Bill Dally, Jeff Dean, et al. MLSys: The new frontier of machine learning systems. *arXiv*, 2019. URL https://arxiv.org/pdf/1904.03257.pdf.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv*, 2018. URL https://arxiv.org/pdf/1805.00907.pdf.

Olli Saarikivi. personal communication.

Abigail See, Minh-Thang Luong, and Christopher D Manning. Compression of neural machine translation models via pruning. *ACL*, 2016.

Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *KDD*, 2016.

Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *ISCA*, 2018.

Laurent Sifre and Stéphane Mallat. Rigid-motion scattering for image classification. *Ph.D. dissertation*, 2014.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Richard M Stallman and GCC DeveloperCommunity. *Using the GNU compiler collection: a GNU manual for GCC version 4.3.3*. CreateSpace, 2009.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv*, 2018. URL `https://arxiv.org/pdf/1802.04730.pdf`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-aware automated quantization with mixed precision. In *CVPR*, 2019.

Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *ICCV*, 2019.

Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. In *ICLR Workshop*, 2018. URL `https://openreview.net/forum?id=S1lN69AT-`.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2017. URL `https://openreview.net/forum?id=r1Ue8Hcxg`.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.