Scalable and Intelligent Learning Systems

Tianqi Chen

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

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Carlos Guestrin, Chair

Luis Ceze

Arvind Krishnamurthy

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Scalable and Intelligent Learning Systems

Tianqi Chen

Chair of the Supervisory Committee:
Professor Carlos Guestrin
Computer Science and Engineering

Data, models, and computing are the three pillars that enable machine learning to solve real-world problems at scale. Making progress on these three domains requires not only disruptive algorithmic advances but also systems innovations that can continue to squeeze more efficiency out of modern hardware. Learning systems are in the center of every intelligent application nowadays. This thesis discusses aspects of learning systems under the context of three real-world systems – XGBoost, MXNet, and TVM.

The first half of the thesis focuses on scalable learning systems that learn parameters for complex models using large-scale data. We introduce XGBoost, a scalable tree boosting system that scales to billions of examples in distributed or memory-limited settings. We then bring a systematic approach under the context of MXNet to reduce the memory consumption of training to scale up real-world deep learning workloads using a minimal amount of resources.

The second half of the thesis brings intelligence to learning systems themselves. We introduce TVM, a system for deploying learning to diverse hardware platforms. TVM exposes graph-level and operator-level optimization knobs to provide performance portability to deep learning workloads across diverse hardware back-ends. We propose transfer learning methods to automate TVM and deliver performance competitive with state-of-the-art hand-tuned libraries for low-power CPU, mobile GPU, and server-class GPU.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Ph.D. is a long journey that I could not have completed without helps from many people. I want to thank all of them here.

First, I would like to thank my excellent advisor Carlos Guestrin. Carlos taught me to think across the boundaries of machine learning and systems. Sometimes he would straightly point out my problems. These honest feedbacks help me to face and overcome my weakness. Sometimes he would make cheerful jokes to encourage me to push beyond my best effort. He was always supportive of bold ideas and guided me to realize them concretely.

I also like to thank the rest of my thesis committee members. Arvind Krishnamurthy would open his door for me whenever I had new ideas in learning systems. He also encouraged me to design and teach a new course about systems for machine learning at UW. I want to thank Luis Ceze for introducing me to the world of hardware and architecture. We started a wonderful collaboration three years ago that eventually gave birth to the TVM project and the SAMPL Lab. I want to thank Marina Meila for thoughtful conversations on automatic program optimizations and model transfer.

Over the past six years, I was fortunate to interact with many researchers. I want to thank Emily Fox for great collaborations on stochastic gradient MCMC. I was fortunate to work with Ian Goodfellow and Jon Shlens for a wonderful summer at Google when we explored knowledge transfer for neural networks. I want to thank Alex Smola and Mu Li for collaborations on deep learning systems. I want to thank Zach Tatlock for fun conversations about compiler optimizations. I want to thank Dawn Song for enlightening discussions about machine learning and program synthesis.

I would also like to thank Ben Taskar. I was co-advised by him for three months before his unfortunate pass away. His wisdom and forward-thinking approach to research constantly inspired

me. Coincidentally, our last conversation was about a paper on hardware acceleration for bitcoin mininng [8] by Professor Micheal B. Taylor (who is now at UW). “If we can use ASICs to accelerate bitcoin mining, perhaps we could do the same thing for machine learning.” I did not realize how visionary these words are until TPU came out – he predicted the trend in 2013.

I also would like to thank my fellow students from SAMPL Lab and MODE Lab. I want to thank Marco Tulio Ribeiro, Tyler Johnson, Yi’an Ma, Thierry Moureau, Jared Roesch, Philip Cho, Haichen Shen, Qiao Zhang, Eddie Yan, Ziheng Jiang, Meghan Cowan, Luis Vega, Logan Weber, Pratyush Patel, Gus Smith, Steven Lyubomirsky, Liang Luo, Lianmin Zheng and many other fellow students at the UW Allen school.

I want to thank Google for giving me a fellowship that supported three years of my Ph.D. It gave me a great amount of freedom to explore new ideas.

I want to thank my collaborators in the XGBoost, Apache MXNet, and Apache TVM open source communities. They deserve the credits for making these systems continue to thrive and impact the world.

Finally, I would like to thank my family members for their supports. I want to thank my wife, Iris (Jianghong) Shi. Outside research, we explored the mountains, forests and islands together, and develop new food recipes. I want to thank her for giving me a wonderful six years’ journey.

DEDICATION

To my family.

Chapter 1

INTRODUCTION

Machine learning has become a part of our daily lives. Our emails are protected by smart spam classifiers, which learn from vast amounts of spam data and user feedback. As we shop online, a recommender system helps us find products that match our taste by learning from our shopping history. Automatic voice recognition systems give us a new way to communicate with smart devices. Image recognition systems help us automatically manage and organize our photos. Besides improving personal life, machine learning also plays a key role in helping companies make smart decisions and generate revenue: Advertising systems match the right ads with the right users at the right time. Demand forecasting systems predict product demand in advance, allowing sellers to be prepared. Fraud detection systems protect banks from malicious attackers. Great success has been witnessed in these applications of machine learning and in many others. Three factors collectively bring the current success of machine learning: modeling and algorithmic advances, data, and hardware accelerations.

Modeling and algorithmic innovations bring learning to a broad spectrum of application domains. Tree-based models [34, 35, 17, 52] and corresponding model interpretation methods [80, 65] are powerful tools to handle tabular data. Factorization models [79] and large-scale linear models [111, 61] power modern recommender systems and advertising pipelines. Techniques such as dropout [89], batch normalization [49] and residual connections [41] simplifies training of deep neural networks. Convolutional neural networks [58, 42] can now recognize images and videos. Recurrent neural networks [46, 93] and transformers [100, 30] bring great advance to speech recognition and natural language processing. Machines now outsmart humans in playing games [68, 85] thanks to the advances in large-scale reinforcement learning.

Data plays an equally important role. As researchers started to try more complicated models,

they need bigger datasets to fuel them. The Netflix challenge [11] brought a new era to the recommender system research. ImageNet [29] opened the door to the deep learning era of computer vision. Wikipedia provides a high-quality source for language modeling. Reinforcement learning's current success in games is partly due to an infinite amount of data from simulation environments and self-plays.

Big data and models create a massive demand for compute and put hardware on the spot. Machine learning researchers are among the early adopters of GPUPU [77]. GPUs now support majority part of the current deep learning pipelines in both academia and industry. Due to the end of Moore's Law and Dennard Scaling, people turn to hardware specialization for help. ASICs like TPU [51] are created to accelerate deep learning.

Model, data, and hardware are the three pillars of machine learning. However, we still need one crucial element to bring them together – learning systems. AlexNet's breakthrough [58] relied on specially written CUDA kernels and a framework built from scratch. It could take the original authors months of effort. Today, thanks to learning systems [3, 73, 19], researchers can write a few lines of python code to reproduce the result. Learning systems provide high-level abstractions for training or deployment and automatically handle details such as data preprocessing, workload distribution, and hardware acceleration. As a result, researchers and industry practitioners can quickly innovate on ideas and offload the engineering details to the systems.

1.1 Thesis Statement

This thesis focused on addressing the challenges of building efficient and scalable learning systems that perform learning to solve real-world problems while using learning to improve those very systems. In particular, this thesis makes contributions to the following aspects of learning systems:

- Scalable systems to bring data science and machine learning for everyone.
- Systems to deploy learning to diverse hardware platforms.
- Automating systems optimization with machine learning.

Scalable Learning Systems (Chapter 2 and 3) To learn the parameters for complex models in large-scale data, one needs to develop both large-scale learning algorithms and scalable training systems. We build XGBoost [17]: a scalable tree boosting system. XGboost contains an efficient columnar block layout for fast tree learning algorithms and a regularized objective to prevent overfitting. The system scales to multiple nodes with a fault-tolerant communication layer. Since its introduction, XGBoost has become one of the de-facto tools used by data scientists daily. It is adopted in production pipelines of the major companies such as Uber, Airbnb, Amazon and Google Cloud, and is used in many machine learning challenge wining solutions. Besides its impact on data science, it also helped scientific explorations. High energy physicists from CERN used XGBoost to analyze the data from the Large Hadron Collider, to find evidence of new physics. We introduce MXNet [19], a scalable deep learning system. We bring a systematic approach to reduce the memory consumption of training to scale up real-world deep learning workloads using a minimal amount of resources.

Deploy Learning to Diverse Hardware Platforms (Chapter 4) One of the key trends of recent learning systems is the introduction of diverse hardware platforms. To enable smart applications anywhere from the edge to the data center, we need to deploy ML workloads to new platforms – such as mobile phones, embedded devices, and accelerators (e.g., FPGAs, ASICs). Deploying ML on every new hardware platform in an ad-hoc fashion requires significant engineering effort. We build TVM [20] an automated optimizing compiler for deep learning. TVM exposes graph-level and operator-level optimization knobs to provide performance portability to deep learning workloads across diverse hardware back-ends. TVM delivers performance across hardware back-ends that are competitive with state-of-the-art, hand-tuned libraries for low-power CPU, mobile GPU, and server-class GPUs. TVM is now an umbrella open-source project that drives key innovations from both academic institutions and industry. The project pushes multiple fronts, innovating on hardware architectures, programming languages, and machine learning. It has more than 200 contributors and has been used in production in major companies.

Automating Systems with Machine Learning (Chapter 5) One of the key challenges in modern

systems design is the enormous amount of labor required to fine-tune performance and efficiency. A sustainable approach to this problem is to automate this process with machine learning. We develop AutoTVM [23], a learning-based framework that optimizes deep learning workloads on a diverse set of hardware backends. It applies domain invariant modeling to transfer statistical cost models between optimization tasks.

1.2 Thesis Contributions

This section summarizes the main contributions of the thesis. This thesis builds on top of materials from out past papers about XGBoost [17], MXNet [19, 22] and TVM [20, 23].

Chapter 2 introduces XGBoost, a scalable tree boosting system. We make the following contributions:

- We design and build a highly scalable end-to-end tree boosting system.
- We propose a theoretically justified weighted quantile sketch for efficient proposals.
- We introduce a novel sparsity-aware algorithm for parallel tree learning.
- We propose an effective cache-aware block structure for out-of-core tree learning.

Chapter 3 introduces MXNet, a scalable and flexible deep learning system. The main contributions are:

- We propose a mixed delcalrative and imperative programming model for deep learning.
- We introduce a sublinear memory planning algorithm to scale training to deep neural networks under limited GPU memory constraints.

Chapter 4 and Chapter 5 introduces TVM and AutoTVM. The contributions of these two chapters are as follows:

- We identify the major optimization challenges in providing performance portability to deep learning workloads across diverse hardware back-ends.

- We introduce novel schedule primitives that take advantage of cross-thread memory reuse, novel hardware intrinsics, and latency hiding.

- We formalize the new problem of learning to optimize tensor programs and summarize its key characteristics, and propose a machine learning framework to solve this problem.

- We further accelerate the optimization by $2\times$ to $10\times$ using transfer learning.

- We build an end-to-end compilation and optimization stack that allows the deployment of deep learning workloads specified in high-level frameworks (including TensorFlow, MXNet, PyTorch, Keras, CNTK) to diverse hardware back-ends (including CPUs, server GPUs, mobile GPUs, and FPGA-based accelerators). The open-sourced TVM is in production use inside several major companies. We evaluated TVM using real-world workloads on a server-class GPU, an embedded GPU, an embedded CPU, and a custom generic FPGA-based accelerator. Experimental results show that TVM offers portable performance across back-ends and achieves speedups ranging from $1.2\times$ to $3.8\times$ over existing frameworks backed by hand-optimized libraries.

Chapter 2

XGBOOST: A SCALABLE TREE BOOSTING SYSTEM

Machine learning and data-driven approaches are becoming very important in many areas. Smart spam classifiers protect our email by learning from massive amounts of spam data and user feedback; advertising systems learn to match the right ads with the right context; fraud detection systems protect banks from malicious attackers; anomaly event detection systems help experimental physicists to find events that lead to new physics. There are two important factors that drive these successful applications: usage of effective (statistical) models that capture the complex data dependencies and scalable learning systems that learn the model of interest from large datasets.

Among the machine learning methods used in practice, gradient tree boosting [34] is one technique that shines in many applications. Tree boosting has been shown to give state-of-the-art results on many standard classification benchmarks [62]. LambdaMART [13], a variant of tree boosting for ranking, achieves state-of-the-art result for ranking problems. Besides being used as a standalone predictor, it is also incorporated into real-world production pipelines for ad click through rate prediction [43]. Finally, it is the de-facto choice of ensemble method and is used in challenges such as the Netflix prize [11].

In this chapter, we describe XGBoost, a scalable machine learning system for tree boosting. The system is available as an open source package. The impact of the system has been widely recognized in a number of machine learning and data mining challenges. Take the challenges hosted by the machine learning competition site Kaggle for example. Among the 29 challenge winning solutions published at Kaggle's blog during 2015, 17 solutions used XGBoost. Among these solutions, eight solely used XGBoost to train the model, while most others combined XGBoost with neural nets in ensembles. For comparison, the second most popular method, deep neural nets, was used in 11 solutions. The success of the system was also witnessed in KDDCup 2015, where XG-

Boost was used by every winning team in the top-10. Moreover, the winning teams reported that ensemble methods outperform a well-configured XGBoost by only a small amount [9].

These results demonstrate that our system gives state-of-the-art results on a wide range of problems. Examples of the problems in these winning solutions include: store sales prediction; high energy physics event classification; web text classification; customer behavior prediction; motion detection; ad click through rate prediction; malware classification; product categorization; hazard risk prediction; massive online course dropout rate prediction. While domain dependent data analysis and feature engineering play an important role in these solutions, the fact that XGBoost is the consensus choice of learner shows the impact and importance of our system and tree boosting.

The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important systems and algorithmic optimizations. These innovations include: a novel tree learning algorithm is for handling *sparse data*; a theoretically justified weighted quantile sketch procedure enables handling instance weights in approximate tree learning. Parallel and distributed computing makes learning faster which enables quicker model exploration. More importantly, XGBoost exploits out-of-core computation and enables data scientists to process hundred millions of examples on a desktop. Finally, it is even more exciting to combine these techniques to make an end-to-end system that scales to even larger data with the least amount of cluster resources.

While there are some existing works on parallel tree boosting [96, 106, 72], the directions such as out-of-core computation, cache-aware and sparsity-aware learning have not been explored. More importantly, an end-to-end system that combines all of these aspects gives a novel solution for real-world use-cases. This enables data scientists as well as researchers to build powerful variants of tree boosting algorithms [18, 21]. Besides these major contributions, we also make additional improvements in proposing a regularized learning objective, which we will include for completeness.

The remainder of the chapter is organized as follows. We will first review tree boosting and

introduce a regularized objective in Sec. 2.1. We then describe the split finding methods in Sec. 2.2 as well as the system design in Sec. 2.3, including experimental results when relevant to provide quantitative support for each optimization we describe. Related work is discussed in Sec. 2.4. Detailed end-to-end evaluations are included in Sec. 2.5.

2.1 Tree Boosting in a NutShell

We review gradient tree boosting algorithms in this section. The derivation follows from the same idea in existing literatures in gradient boosting. Specicially the second order method is originated from Friedman et al. [36]. We make minor improvements in the reguralized objective, which were found helpful in practice.

2.1.1 Regularized Learning Objective

For a given data set with n examples and m features $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ ($|\mathcal{D}| = n, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R}$), a tree ensemble model (shown in Fig. 2.1) uses K additive functions to predict the output.

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F}, \tag{2.1}$$

where $\mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}(q : \mathbb{R}^m \to T, w \in \mathbb{R}^T)$ is the space of regression trees (also known as CART). Here q represents the structure of each tree that maps an example to the corresponding leaf index. T is the number of leaves in the tree. Each f_k corresponds to an independent tree structure q and leaf weights w. Unlike decision trees, each regression tree contains a continuous score on each of the leaf, we use w_i to represent score on i-th leaf. For a given example, we will use the decision rules in the trees (given by q) to classify it into the leaves and calculate the final prediction by summing up the score in the corresponding leaves (given by w). To learn the set of functions used in the model, we minimize the following *regularized* objective.

$$\begin{aligned} \mathcal{L}(\phi) &= \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \\ \text{where } \Omega(f) &= \gamma T + \frac{1}{2}\lambda \|w\|^2 \end{aligned} \tag{2.2}$$

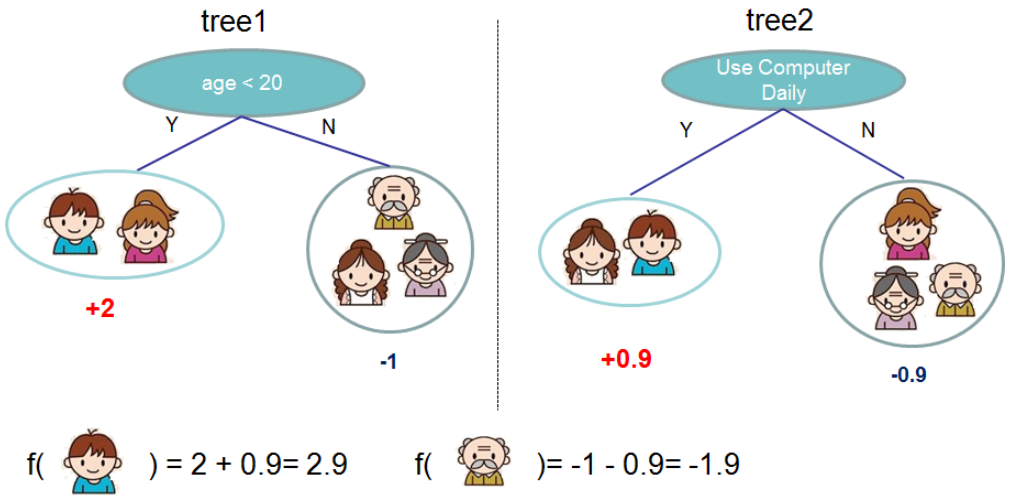


Figure 2.1: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

Here l is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target y_i. The second term Ω penalizes the complexity of the model (i.e., the regression tree functions). The additional regularization term helps to smooth the final learnt weights to avoid over-fitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions. A similar regularization technique has been used in Regularized greedy forest (RGF) [109] model. Our objective and the corresponding learning algorithm is simpler than RGF and easier to parallelize. When the regularization parameter is set to zero, the objective falls back to the traditional gradient tree boosting.

2.1.2 Gradient Tree Boosting

The tree ensemble model in Eq. (2.2) includes functions as parameters and cannot be optimized using traditional optimization methods in Euclidean space. Instead, the model is trained in an additive manner. Formally, let $\hat{y}_i^{(t)}$ be the prediction of the i-th instance at the t-th iteration, we will

need to add f_t to minimize the following objective.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

This means we greedily add the f_t that most improves our model according to Eq. (2.2). Second-order approximation can be used to quickly optimize the objective in the general setting [36].

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ are first and second order gradient statistics on the loss function. We can remove the constant terms to obtain the following simplified objective at step t.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t) \tag{2.3}$$

Define $I_j = \{i | q(\mathbf{x}_i) = j\}$ as the instance set of leaf j. We can rewrite Eq (2.3) by expanding Ω as follows

$$\begin{aligned} \tilde{\mathcal{L}}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned} \tag{2.4}$$

For a fixed structure $q(\mathbf{x})$, we can compute the optimal weight w_j^* of leaf j by

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \tag{2.5}$$

and calculate the corresponding optimal value by

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T. \tag{2.6}$$

Eq (2.6) can be used as a scoring function to measure the quality of a tree structure q. This score is like the impurity score for evaluating decision trees, except that it is derived for a wider range of objective functions. Fig. 2.2 illustrates how this score can be calculated.

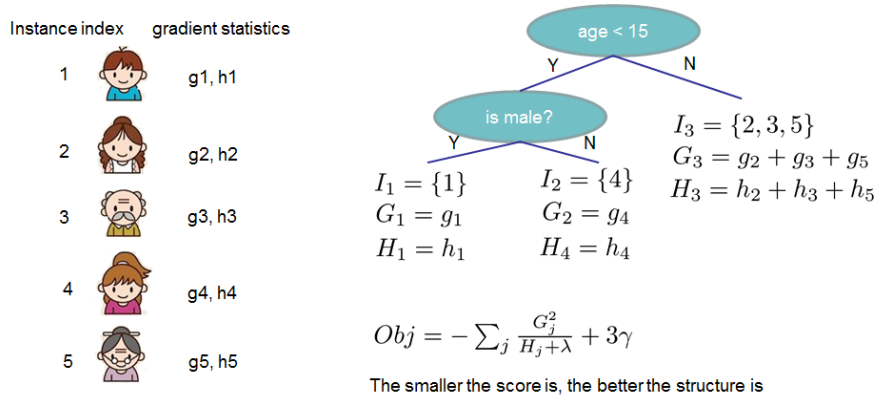


Figure 2.2: Structure Score Calculation. We only need to sum up the gradient and second order gradient statistics on each leaf, then apply the scoring formula to get the quality score.

Normally it is impossible to enumerate all the possible tree structures q. A greedy algorithm that starts from a single leaf and iteratively adds branches to the tree is used instead. Assume that I_L and I_R are the instance sets of left and right nodes after the split. Lettting $I = I_L \cup I_R$, then the loss reduction after the split is given by

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{2.7}$$

This formula is usually used in practice for evaluating the split candidates.

2.1.3 Shrinkage and Column Subsampling

Besides the regularized objective mentioned in Sec. 2.1.1, two additional techniques are used to further prevent over-fitting. The first technique is shrinkage introduced by Friedman [35]. Shrinkage scales newly added weights by a factor η after each step of tree boosting. Similar to a learning rate in tochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model. The second technique is column (feature) subsampling. This technique is used in RandomForest [12, 33], It is implemented in a commercial software TreeNet [1]

[1] https://www.salford-systems.com/products/treenet

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node

Input: d, feature dimension

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ ***to*** m **do**

  $G_L \leftarrow 0,\ H_L \leftarrow 0$

  for j *in sorted*(I, *by* $\mathbf{x}_{jk}$) **do**

    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$

    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$

    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

  end

end

Output: Split with max score

for gradient boosting, but is not implemented in existing opensource packages. According to user feedback, using column sub-sampling prevents over-fitting even more so than the traditional row sub-sampling (which is also supported). The usage of column sub-samples also speeds up computations of the parallel algorithm described later.

2.2 Split Finding Algorithms

2.2.1 Basic Exact Greedy Algorithm

One of the key problems in tree learning is to find the best split as indicated by Eq (2.7). In order to do so, a split finding algorithm enumerates over all the possible splits on all the features. We call this the *exact greedy algorithm*. Most existing single machine tree boosting implementations, such as scikit-learn [74], R's gbm [81] as well as the single machine version of XGBoost support the exact greedy algorithm. The exact greedy algorithm is shown in Alg. 1. It is computationally demanding to enumerate all the possible splits for continuous features. In order to do so efficiently,

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ ***to*** m **do**

Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.

Proposal can be done per tree (global), or per split(local).

end

for $k = 1$ ***to*** m **do**

$G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j \; H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$

end

Follow same step as in previous section to find max score only among proposed splits.

the algorithm must first sort the data according to feature values and visit the data in sorted order to accumulate the gradient statistics for the structure score in Eq (2.7).

2.2.2 Approximate Algorithm

The exact greedy algorithm is very powerful since it enumerates over all possible splitting points greedily. However, it is impossible to efficiently do so when the data does not fit entirely into memory. Same problem also arises in the distributed setting. To support effective gradient tree boosting in these two settings, an approximate algorithm is needed.

We summarize an approximate framework, which resembles the ideas proposed in past literatures [63, 10, 96], in Alg. 2. To summarize, the algorithm first proposes candidate splitting points according to percentiles of feature distribution (a specific criteria will be given in Sec. 2.2.3). The algorithm then maps the continuous features into buckets split by these candidate points, aggregates the statistics and finds the best solution among proposals based on the aggregated statistics.

There are two variants of the algorithm, depending on when the proposal is given. The global variant proposes all the candidate splits during the initial phase of tree construction, and uses the same proposals for split finding at all levels. The local variant re-proposes after each split. The global method requires less proposal steps than the local method. However, usually more candidate points are needed for the global proposal because candidates are not refined after each split. The

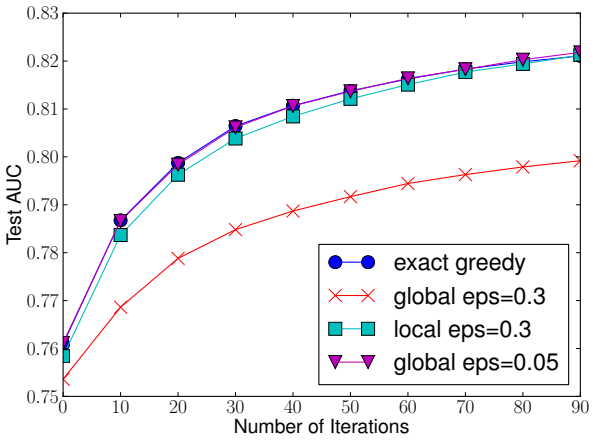


Figure 2.3: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

local proposal refines the candidates after splits, and can potentially be more appropriate for deeper trees. A comparison of different algorithms on a Higgs boson dataset is given by Fig. 2.3. We find that the local proposal indeed requires fewer candidates. The global proposal can be as accurate as the local one given enough candidates.

Most existing approximate algorithms for distributed tree learning also follow this framework. Notably, it is also possible to directly construct approximate histograms of gradient statistics [96]. It is also possible to use other variants of binning strategies instead of quantile [63]. Quantile strategy benefit from being distributable and recomputable, which we will detail in next subsection. From Fig. 2.3, we also find that the quantile strategy can get the same accuracy as exact greedy given reasonable approximation level.

Our system efficiently supports exact greedy for the single machine setting, as well as approximate algorithm with both local and global proposal methods for all settings. Users can freely choose between the methods according to their needs.

2.2.3 *Weighted Quantile Sketch*

One important step in the approximate algorithm is to propose candidate split points. Usually percentiles of a feature are used to make candidates distribute evenly on the data. Formally, let multi-set $\mathcal{D}_k = \{(x_{1k}, h_1), (x_{2k}, h_2) \cdots (x_{nk}, h_n)\}$ represent the k-th feature values and second order gradient statistics of each training instances. We can define a rank functions $r_k : \mathbb{R} \to [0, +\infty)$ as

$$r_k(z) = \frac{1}{\sum_{(x,h)\in\mathcal{D}_k} h} \sum_{(x,h)\in\mathcal{D}_k, x<z} h, \tag{2.8}$$

which represents the proportion of instances whose feature value k is smaller than z. The goal is to find candidate split points $\{s_{k1}, s_{k2}, \cdots s_{kl}\}$, such that

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon, \quad s_{k1} = \min_i \mathbf{x}_{ik}, s_{kl} = \max_i \mathbf{x}_{ik}. \tag{2.9}$$

Here ϵ is an approximation factor. Intuitively, this means that there is roughly $1/\epsilon$ candidate points. Here each data point is weighted by h_i. To see why h_i represents the weight, we can rewrite Eq (2.3) as

$$\sum_{i=1}^{n} \frac{1}{2} h_i (f_t(\mathbf{x}_i) - g_i/h_i)^2 + \Omega(f_t) + constant,$$

which is exactly weighted squared loss with labels g_i/h_i and weights h_i. For large datasets, it is non-trivial to find candidate splits that satisfy the criteria. When every instance has equal weights, an existing algorithm called quantile sketch [39, 108] solves the problem. However, there is no existing quantile sketch for the weighted datasets. Therefore, most existing approximate algorithms either resorted to sorting on a random subset of data which have a chance of failure or heuristics that do not have theoretical guarantee.

To solve this problem, we introduced a novel distributed weighted quantile sketch algorithm that can handle weighted data with a *provable theoretical guarantee*. The general idea is to propose a data structure that supports *merge* and *prune* operations, with each operation proven to maintain a certain accuracy level. A detailed description of the algorithm as well as proofs are given in Sec. 2.6.

Algorithm 3: Sparsity-aware Split Finding

Input: I, instance set of current node

Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$

Input: d, feature dimension

Also applies to the approximate setting, only collect statistics of non-missing entries into buckets

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I}, g_i$,$H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ ***to*** m **do**

// enumerate missing value goto right

$G_L \leftarrow 0,\ H_L \leftarrow 0$

for j *in sorted(*I_k*, ascent order by* $\mathbf{x}_{jk}$*)* **do**

$G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

end

// enumerate missing value goto left

$G_R \leftarrow 0,\ H_R \leftarrow 0$

for j *in sorted(*I_k*, descent order by* $\mathbf{x}_{jk}$*)* **do**

$G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$

$G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$

$score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

end

end

Output: Split and default directions with max gain

2.2.4 Sparsity-aware Split Finding

In many real-world problems, it is quite common for the input $\mathbf{x}$ to be sparse. There are multiple possible causes for sparsity: 1) presence of missing values in the data; 2) frequent zero entries in the statistics; and, 3) artifacts of feature engineering such as one-hot encoding. It is important

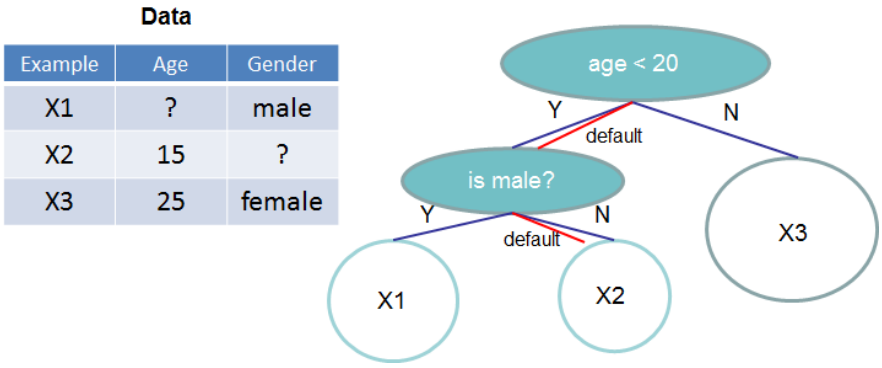


Figure 2.4: Tree structure with default directions. An example will be classified into the default direction when the feature needed for the split is missing.

to make the algorithm aware of the sparsity pattern in the data. In order to do so, we propose to add a default direction in each tree node, which is shown in Fig. 2.4. When a value is missing in the sparse matrix $\mathbf{x}$, the instance is classified into the default direction. There are two choices of default direction in each branch. The optimal default directions are learnt from the data. The algorithm is shown in Alg. 3. The key improvement is to only visit the non-missing entries I_k. The presented algorithm treats the non-presence as a missing value and learns the best direction to handle missing values. The same algorithm can also be applied when the non-presence corresponds to a user specified value by limiting the enumeration only to consistent solutions.

To the best of our knowledge, most existing tree learning algorithms are either only optimized for dense data, or need specific procedures to handle limited cases such as categorical encoding. XGBoost handles all sparsity patterns in a unified way. More importantly, our method exploits the sparsity to make computation complexity linear to number of non-missing entries in the input. Fig. 2.5 shows the comparison of sparsity aware and a naive implementation on an Allstate-10K dataset (description of dataset given in Sec. 2.5). We find that the sparsity aware algorithm runs 50 times faster than the naive version. This confirms the importance of the sparsity aware algorithm.

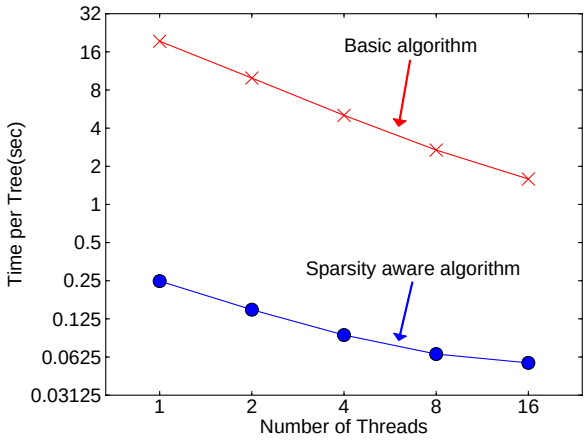


Figure 2.5: Impact of the sparsity aware algorithm on Allstate-10K. The dataset is sparse mainly due to one-hot encoding. The sparsity aware algorithm is more than 50 times faster than the naive version that does not take sparsity into consideration.

2.3 System Design

2.3.1 Column Block for Parallel Learning

The most time consuming part of tree learning is to get the data into sorted order. In order to reduce the cost of sorting, we propose to store the data in in-memory units, which we called *block*. Data in each block is stored in the compressed column (CSC) format, with each column sorted by the corresponding feature value. This input data layout only needs to be computed once before training, and can be reused in later iterations.

In the exact greedy algorithm, we store the entire dataset in a single block and run the split search algorithm by linearly scanning over the pre-sorted entries. We do the split finding of all leaves collectively, so one scan over the block will collect the statistics of the split candidates in all leaf branches. Fig. 2.6 shows how we transform a dataset into the format and find the optimal split using the block structure.

The block structure also helps when using the approximate algorithms. Multiple blocks can

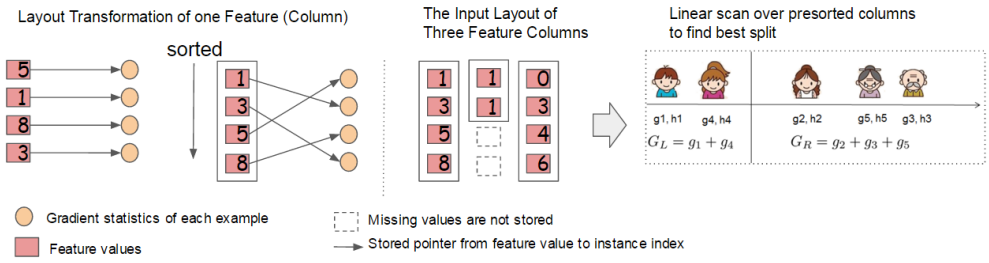
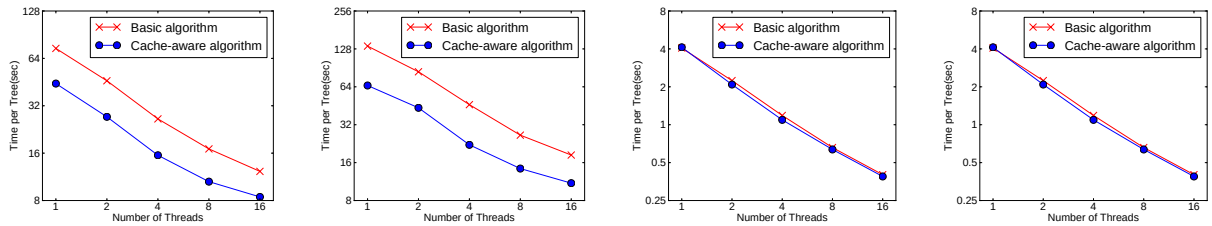


Figure 2.6: Block structure for parallel learning. Each column in a block is sorted by the corresponding feature value. A linear scan over one column in the block is sufficient to enumerate all the split points.



(a) Allstate 10M (b) Higgs 10M (c) Allstate 1M (d) Higgs 1M

Figure 2.7: Impact of cache-aware prefetching in exact greedy algorithm. We find that the cache-miss effect impacts the performance on the large datasets (10 million instances). Using cache aware prefetching improves the performance by factor of two when the dataset is large.

be used in this case, with each block corresponding to subset of rows in the dataset. Different blocks can be distributed across machines, or stored on disk in the out-of-core setting. Using the sorted structure, the quantile finding step becomes a *linear scan* over the sorted columns. This is especially valuable for local proposal algorithms, where candidates are generated frequently at each branch. The binary search in histogram aggregation also becomes a linear time merge style algorithm.

Collecting statistics for each column can be *parallelized*, giving us a parallel algorithm for split finding. Importantly, the column block structure also supports column subsampling, as it is easy to

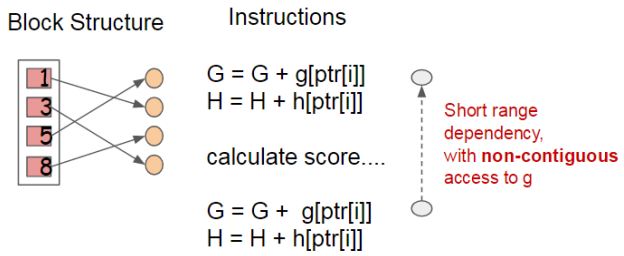


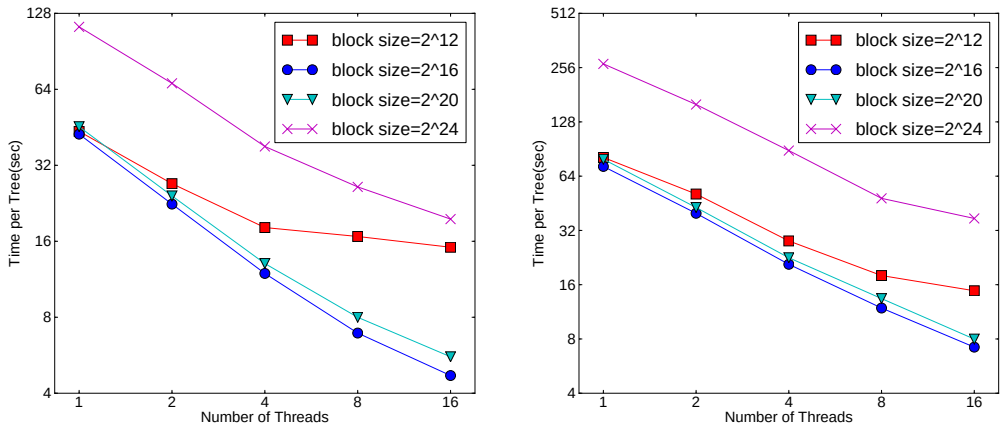
Figure 2.8: Short range data dependency pattern that can cause stall due to cache miss.

select a subset of columns in a block.

Time Complexity Analysis Let d be the maximum depth of the tree and K be total number of trees. For the exact greedy algorithm, the time complexity of original spase aware algorithm is $O(Kd\|\mathbf{x}\|_0 \log n)$. Here we use $\|\mathbf{x}\|_0$ to denote number of non-missing entries in the training data. On the other hand, tree boosting on the block structure only cost $O(Kd\|\mathbf{x}\|_0 + \|\mathbf{x}\|_0 \log n)$. Here $O(\|\mathbf{x}\|_0 \log n)$ is the one time preprocessing cost that can be amortized. This analysis shows that the block structure helps to save an additional $\log n$ factor, which is significant when n is large. For the approximate algorithm, the time complexity of original algorithm with binary search is $O(Kd\|\mathbf{x}\|_0 \log q)$. Here q is the number of proposal candidates in the dataset. While q is usually between 32 and 100, the log factor still introduces overhead. Using the block structure, we can reduce the time to $O(Kd\|\mathbf{x}\|_0 + \|\mathbf{x}\|_0 \log B)$, where B is the maximum number of rows in each block. Again we can save the additional $\log q$ factor in computation.

2.3.2 Cache-aware Access

While the proposed block structure helps optimize the computation complexity of split finding, the new algorithm requires indirect fetches of gradient statistics by row index, since these values are accessed in order of feature. This is a non-continuous memory access. A naive implementation of split enumeration introduces immediate read/write dependency between the accumulation and the non-continuous memory fetch operation (see Fig. 2.8). This slows down split finding when the gradient statistics do not fit into CPU cache and cache miss occur.



(a) Allstate 10M (b) Higgs 10M

Figure 2.9: The impact of block size in the approximate algorithm. We find that overly small blocks results in inefficient parallelization, while overly large blocks also slows down training due to cache misses.

For the exact greedy algorithm, we can alleviate the problem by a cache-aware prefetching algorithm. Specifically, we allocate an internal buffer in each thread, fetch the gradient statistics into it, and then perform accumulation in a mini-batch manner. This prefetching changes the direct read/write dependency to a longer dependency and helps to reduce the runtime overhead when number of rows in the is large. Figure 2.7 gives the comparison of cache-aware vs. non cache-aware algorithm on the the Higgs and the Allstate dataset. We find that cache-aware implementation of the exact greedy algorithm runs twice as fast as the naive version when the dataset is large.

For approximate algorithms, we solve the problem by choosing a correct block size. We define the block size to be maximum number of examples in contained in a block, as this reflects the cache storage cost of gradient statistics. Choosing an overly small block size results in small workload for each thread and leads to inefficient parallelization. On the other hand, overly large blocks result in cache misses, as the gradient statistics do not fit into the CPU cache. A good choice of block size balances these two factors. We compared various choices of block size on two data sets. The results are given in Fig. 2.9. This result validates our discussion and shows that choosing 2^{16}

Table 2.1: Comparison of major tree boosting systems.

System	exact greedy	approximate global	approximate local	out-of-core	sparsity aware	parallel
XGBoost	yes	yes	yes	yes	yes	yes
pGBRT	no	no	yes	no	no	yes
Spark MLLib	no	yes	no	no	partially	yes
H2O	no	yes	no	no	partially	yes
scikit-learn	yes	no	no	no	no	no
R GBM	yes	no	no	no	partially	no

examples per block balances the cache property and parallelization.

2.3.3 Blocks for Out-of-core Computation

One goal of our system is to fully utilize a machine's resources to achieve scalable learning. Besides processors and memory, it is important to utilize disk space to handle data that does not fit into main memory. To enable out-of-core computation, we divide the data into multiple blocks and store each block on disk. During computation, it is important to use an independent thread to pre-fetch the block into a main memory buffer, so computation can happen in concurrence with disk reading. However, this does not entirely solve the problem since the disk reading takes most of the computation time. It is important to reduce the overhead and increase the throughput of disk IO. We mainly use two techniques to improve the out-of-core computation.

Block Compression The first technique we use is block compression. The block is compressed by columns, and decompressed on the fly by an independent thread when loading into main memory. This helps to trade some of the computation in decompression with the disk reading cost. We use a general purpose compression algorithm for compressing the features values. For the row index, we substract the row index by the begining index of the block and use a 16bit integer to store each offset. This requires 2^{16} examples per block, which is confirmed to be a good setting. In most of

the dataset we tested, we achieve roughly a 26% to 29% compression ratio.

Block Sharding The second technique is to shard the data onto multiple disks in an alternative manner. A pre-fetcher thread is assigned to each disk and fetches the data into an in-memory buffer. The training thread then alternatively reads the data from each buffer. This helps to increase the throughput of disk reading when multiple disks are available.

2.4 Related Works

Our system implements gradient boosting [34], which performs additive optimization in functional space. Gradient tree boosting has been successfully used in classification [36], learning to rank [13], structured prediction [21] as well as other fields. XGBoost incorporates a regularized model to prevent overfitting. This this resembles previous work on regularized greedy forest [109], but simplifies the objective and algorithm for parallelization. Column sampling is a simple but effective technique borrowed from RandomForest [12]. While sparsity-aware learning is essential in other types of models such as linear models [32], few works on tree learning have considered this topic in a principled way. The algorithm proposed in this chapter is the first unified approach to handle all kinds of sparsity patterns.

There are several existing works on parallelizing tree learning [96, 72]. Most of these algorithms fall into the approximate framework described in this chapter. Notably, it is also possible to partition data by columns [106] and apply the exact greedy algorithm. This is also supported in our framework, and the techniques such as cache-aware pre-fecthing can be used to benefit this type of algorithm. While most existing works focus on the algorithmic aspect of parallelization, our work improves in two unexplored system directions: out-of-core computation and cache-aware learning. This gives us insights on how the system and the algorithm can be jointly optimized and provides an end-to-end system that can handle large scale problems with very limited computing resources. We also summarize the comparison between our system and existing opensource implementations in Table 2.1.

Quantile summary (without weights) is a classical problem in the database community [39, 108]. However, the approximate tree boosting algorithm reveals a more general problem – finding

Table 2.2: Dataset used in the Experiments.

Dataset	n	m	Task
Allstate	10 M	4227	Insurance claim classification
Higgs Boson	10 M	28	Event classification
Yahoo LTRC	473K	700	Learning to Rank
Criteo	1.7 B	67	Click through rate prediction

quantiles on weighted data. To the best of our knowledge, the weighted quantile sketch proposed in this chapter is the first method to solve this problem. The weighted quantile summary is also not specific to the tree learning and can benefit other applications in data science and machine learning in the future.

2.5 Evaluations

2.5.1 System Implementation

We implemented XGBoost as an open source package. The package is portable and reusable. It supports various weighted classification and rank objective functions, as well as user defined objective function. It is available in popular languages such as python, R, Julia and integrates naturally with language native data science pipelines such as scikit-learn. The distributed version is built on top of the rabit library[2] for allreduce. The portability of XGBoost makes it available in many ecosystems, instead of only being tied to a specific platform. The distributed XGBoost runs natively on Hadoop, MPI Sun Grid engine. Recently, we also enable distributed XGBoost on jvm bigdata stacks such as Flink and Spark. The distributed version has also been integrated into cloud platform Tianchi of Alibaba. We believe that there will be more integrations in the future.

2.5.2 Dataset and Setup

We used four datasets in our experiments. A summary of these datasets is given in Table 2.2. In some of the experiments, we use a randomly selected subset of the data either due to slow baselines or to demonstrate the performance of the algorithm with varying dataset size. We use a suffix to denote the size in these cases. For example Allstate-10K means a subset of the Allstate dataset with 10K instances.

The first dataset we use is the Allstate insurance claim dataset[3]. The task is to predict the likelihood and cost of an insurance claim given different risk factors. In the experiment, we simplified the task to only predict the likelihood of an insurance claim. This dataset is used to evaluate the impact of sparsity-aware algorithm in Sec. 2.2.4. Most of the sparse features in this data come from one-hot encoding. We randomly select 10M instances as training set and use the rest as evaluation set.

The second dataset is the Higgs boson dataset[4] from high energy physics. The data was produced using Monte Carlo simulations of physics events. It contains 21 kinematic properties measured by the particle detectors in the accelerator. It also contains seven additional derived physics quantities of the particles. The task is to classify whether an event corresponds to the Higgs boson. We randomly select 10M instances as training set and use the rest as evaluation set.

The third dataset is the Yahoo! learning to rank challenge dataset [15], which is one of the most commonly used benchmarks in learning to rank algorithms. The dataset contains 20K web search queries, with each query corresponding to a list of around 22 documents. The task is to rank the documents according to relevance of the query. We use the official train test split in our experiment.

The last dataset is the criteo terabyte click log dataset[5]. We use this dataset to evaluate the scaling property of the system in the out-of-core and the distributed settings. The data contains

[2]https://github.com/dmlc/rabit

[3]https://www.kaggle.com/c/ClaimPredictionChallenge

[4]https://archive.ics.uci.edu/ml/datasets/HIGGS

[5]http://labs.criteo.com/downloads/download-terabyte-click-logs/

Table 2.3: Comparison of Exact Greedy Methods with 500 trees on Higgs-1M data.

Method	Time per Tree (sec)	Test AUC
XGBoost	0.6841	0.8304
XGBoost (colsample=0.5)	0.6401	0.8245
scikit-learn	28.51	0.8302
R.gbm	1.032	0.6224

13 integer features and 26 ID features of user, item and advertiser information. Since a tree based model is better at handling continuous features, we preprocess the data by calculating the statistics of average CTR and count of ID features on the first ten days, replacing the ID features by the corresponding count statistics during the next ten days for training. The training set after preprocessing contains 1.7 billion instances with 67 features (13 integer, 26 average CTR statistics and 26 counts). The entire dataset is more than one terabyte in LibSVM format.

We use the first three datasets for the single machine parallel setting, and the last dataset for the distributed and out-of-core settings. All the single machine experiments are conducted on a Dell PowerEdge R420 with two eight-core Intel Xeon (E5-2470) (2.3GHz) and 64GB of memory. If not specified, all the experiments are run using all the available cores in the machine. The machine settings of the distributed and the out-of-core experiments will be described in the corresponding section. In all the experiments, we boost trees with a common setting of maximum depth equals 8, shrinkage equals 0.1 and no column subsampling unless explicitly specified. We can find similar results when we use other settings of maximum depth.

2.5.3 Classification

In this section, we evaluate the performance of XGBoost on a single machine using the exact greedy algorithm on Higgs-1M data, by comparing it against two other commonly used exact greedy tree boosting implementations. Since scikit-learn only handles non-sparse input, we choose the dense Higgs dataset for a fair comparison. We use the 1M subset to make scikit-learn finish running in

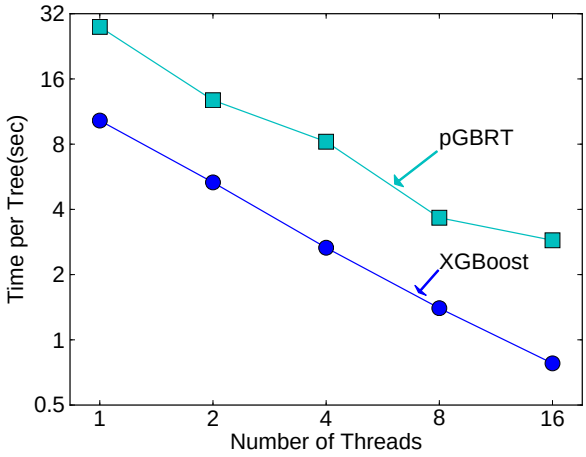


Figure 2.10: Comparison between XGBoost and pGBRT on Yahoo LTRC dataset.

Table 2.4: Comparison of Learning to Rank with 500 trees on Yahoo! LTRC Dataset

Method	Time per Tree (sec)	NDCG@10
XGBoost	0.826	0.7892
XGBoost (colsample=0.5)	0.506	0.7913
pGBRT [96]	2.576	0.7915

reasonable time. Among the methods in comparison, R's GBM uses a greedy approach that only expands one branch of a tree, which makes it faster but can result in lower accuracy, while both scikit-learn and XGBoost learn a full tree. The results are shown in Table 2.3. Both XGBoost and scikit-learn give better performance than R's GBM, while XGBoost runs more than 10x faster than scikit-learn. In this experiment, we also find column subsamples gives slightly worse performance than using all the features. This could due to the fact that there are few important features in this dataset and we can benefit from greedily select from all the features.

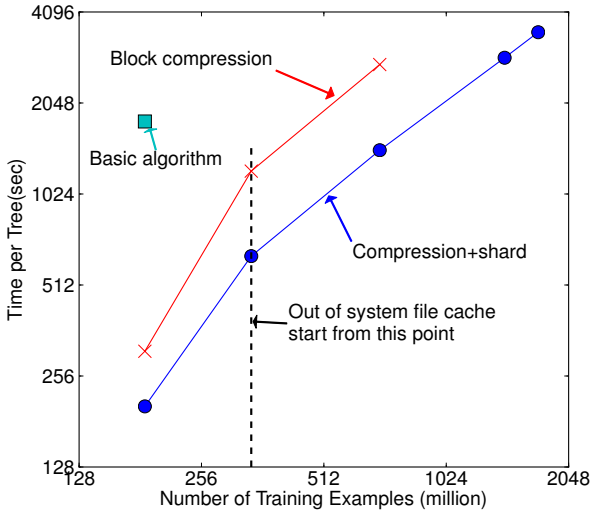
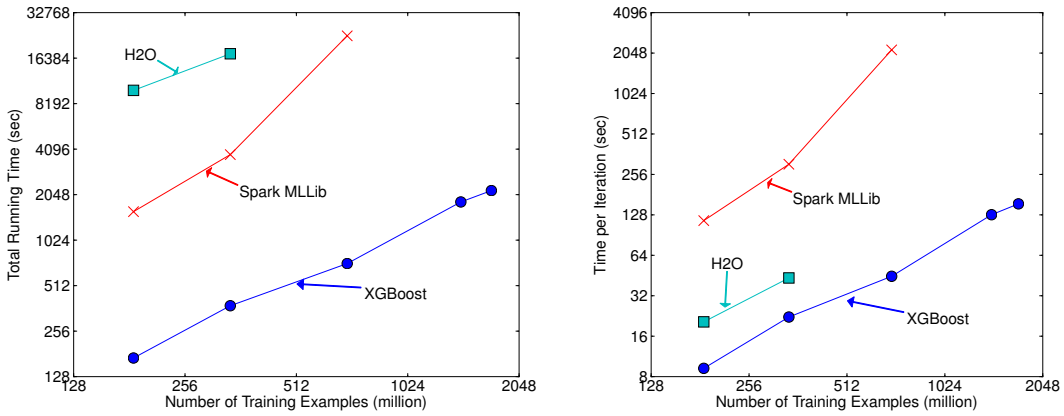


Figure 2.11: Comparison of out-of-core methods on different subsets of criteo data. The missing data points are due to out of disk space. We can find that basic algorithm can only handle 200M examples. Adding compression gives 3x speedup, and sharding into two disks gives another 2x speedup. The system runs out of file cache start from 400M examples. The algorithm really has to rely on disk after this point. The compression+shard method has a less dramatic slowdown when running out of file cache, and exhibits a linear trend afterwards.

2.5.4 Learning to Rank

We next evaluate the performance of XGBoost on the learning to rank problem. We compare against pGBRT [96], the best previously pubished system on this task. XGBoost runs exact greedy algorithm, while pGBRT only support an approximate algorithm. The results are shown in Table 2.4 and Fig. 2.10. We find that XGBoost runs faster. Interestingly, subsampling columns not only reduces running time, and but also gives a bit higher performance for this problem. This could due to the fact that the subsampling helps prevent overfitting, which is observed by many of the users.



(a) End-to-end time cost include data loading (b) Per iteration cost exclude data loading

Figure 2.12: Comparison of different distributed systems on 32 EC2 nodes for 10 iterations on different subset of criteo data. XGBoost runs more 10x than spark per iteration and 2.2x as H2O's optimized version (However, H2O is slow in loading the data, getting worse end-to-end time). Note that spark suffers from drastic slow down when running out of memory. XGBoost runs faster and scales smoothly to the full 1.7 billion examples with given resources by utilizing out-of-core computation.

2.5.5 Out-of-core Experiment

We also evaluate our system in the out-of-core setting on the criteo data. We conducted the experiment on one AWS c3.8xlarge machine (32 vcores, two 320 GB SSD, 60 GB RAM). The results are shown in Figure 2.11. We can find that compression helps to speed up computation by factor of three, and sharding into two disks further gives 2x speedup. For this type of experiment, it is important to use a very large dataset to drain the system file cache for a real out-of-core setting. This is indeed our setup. We can observe a transition point when the system runs out of file cache. Note that the transition in the final method is less dramatic. This is due to larger disk throughput and better utilization of computation resources. Our final method is able to process 1.7 billion examples on a single machine.

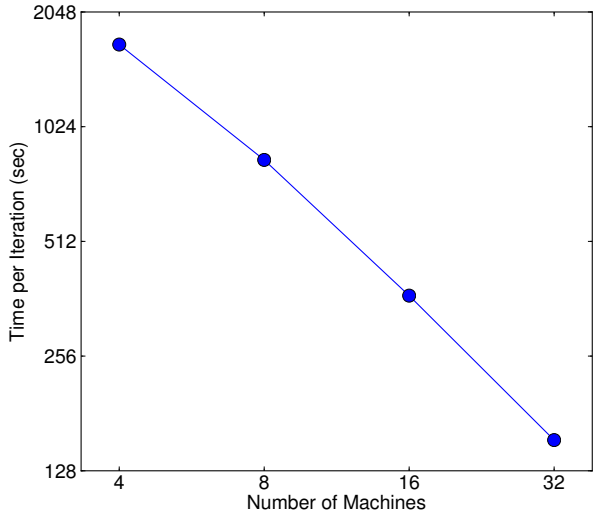


Figure 2.13: Scaling of XGBoost with different number of machines on criteo full 1.7 billion dataset. Using more machines results in more file cache and makes the system run faster, causing the trend to be slightly super linear. XGBoost can process the entire dataset using as little as four machines, and scales smoothly by utilizing more available resources.

2.5.6 Distributed Experiment

Finally, we evaluate the system in the distributed setting. We set up a YARN cluster on EC2 with m3.2xlarge machines, which is a very common choice for clusters. Each machine contains 8 virtual cores, 30GB of RAM and two 80GB SSD local disks. The dataset is stored on AWS S3 instead of HDFS to avoid purchasing persistent storage.

We first compare our system against two production-level distributed systems: Spark MLLib [66] and H2O [6]. We use 32 m3.2xlarge machines and test the performance of the systems with various input size. Both of the baseline systems are in-memory analytics frameworks that need to store the data in RAM, while XGBoost can switch to out-of-core setting when it runs out of memory. The results are shown in Fig. 2.12. We can find that XGBoost runs faster than the baseline

[6] www.h2o.ai

systems. More importantly, it is able to take advantage of out-of-core computing and smoothly scale to all 1.7 billion examples with the given limited computing resources. The baseline systems are only able to handle subset of the data with the given resources. This experiment shows the advantage to bring all the system improvement together and solve a real-world scale problem. We also evaluate the scaling property of XGBoost by varying the number of machines. The results are shown in Fig. 2.13. We can find XGBoost's performance scales linearly as we add more machines. Importantly, XGBoost is able to handle the entire 1.7 billion data with only four machines. This shows the system's potential to handle even larger data.

2.6 Details about Weighted Quantile Sketch

In this section, we discuss the details about the weighted quantile sketch algorithm. Approximate answer of quantile queries is for many real-world applications. One classical approach to this problem is GK algorithm [39] and extensions based on the GK framework [108]. The main component of these algorithms is a data structure called quantile summary, that is able to answer quantile queries with relative accuracy of ϵ. Two operations are defined for a quantile summary:

- A merge operation that combines two summaries with approximation error ϵ_1 and ϵ_2 together and create a merged summary with approximation error $\max(\epsilon_1, \epsilon_2)$.

- A prune operation that reduces the number of elements in the summary to $b+1$ and changes approximation error from ϵ to $\epsilon + \frac{1}{b}$.

A quantile summary with merge and prune operations forms basic building blocks of the distributed and streaming quantile computing algorithms [108].

In order to use quantile computation for approximate tree boosting, we need to find quantiles on weighted data. This more general problem is not supported by any of the existing algorithm. In this section, we describe a non-trivial weighted quantile summary structure to solve this problem. Importantly, the new algorithm contains merge and prune operations with *the same guarantee* as GK summary. This allows our summary to be plugged into all the frameworks used GK summary as building block and answer quantile queries over weighted data efficiently.

2.6.1 Formalization and Definitions

Given an input multi-set $\mathcal{D} = \{(x_1, w_1), (x_2, w_2) \cdots (x_n, w_n)\}$ such that $w_i \in [0, +\infty), x_i \in \mathcal{X}$. Each x_i corresponds to a position of the point and w_i is the weight of the point. Assume we have a total order $<$ defined on $\mathcal{X}$. Let us define two rank functions $r^-_{\mathcal{D}}, r^+_{\mathcal{D}} : \mathcal{X} \to [0, +\infty)$

$$r^-_{\mathcal{D}}(y) = \sum_{(x,w)\in\mathcal{D}, x<y} w \tag{2.10}$$

$$r^+_{\mathcal{D}}(y) = \sum_{(x,w)\in\mathcal{D}, x\leq y} w \tag{2.11}$$

We should note that since $\mathcal{D}$ is defined to be a *multiset* of the points. It can contain multiple record with exactly same position x and weight w. We also define another weight function $\omega_{\mathcal{D}} : \mathcal{X} \to [0, +\infty)$ as

$$\omega_{\mathcal{D}}(y) = r^+_{\mathcal{D}}(y) - r^-_{\mathcal{D}}(y) = \sum_{(x,w)\in\mathcal{D}, x=y} w. \tag{2.12}$$

Finally, we also define the weight of multi-set $\mathcal{D}$ to be the sum of weights of all the points in the set

$$\omega(\mathcal{D}) = \sum_{(x,w)\in\mathcal{D}} w \tag{2.13}$$

Our task is given a series of input $\mathcal{D}$, to estimate $r^+(y)$ and $r^-(y)$ for $y \in \mathcal{X}$ as well as finding points with specific rank. Given these notations, we define quantile summary of weighted examples as follows:

Definition 2.6.1. *Quantile Summary of Weighted Data*
A quantile summary for $\mathcal{D}$ is defined to be tuple $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$, where $S = \{x_1, x_2, \cdots, x_k\}$ is selected from the points in $\mathcal{D}$ (i.e. $x_i \in \{x|(x, w) \in \mathcal{D}\}$) with the following properties:

1) $x_i < x_{i+1}$ for all i, and x_1 and x_k are minimum and maximum point in $\mathcal{D}$:

$$x_1 = \min_{(x,w)\in\mathcal{D}} x, \quad x_k = \max_{(x,w)\in\mathcal{D}} x$$

2) $\tilde{r}^+_{\mathcal{D}}$, $\tilde{r}^-_{\mathcal{D}}$ and $\tilde{\omega}_{\mathcal{D}}$ are functions in $S \to [0, +\infty)$, that satisfies

$$\tilde{r}^-_{\mathcal{D}}(x_i) \leq r^-_{\mathcal{D}}(x_i), \quad \tilde{r}^+_{\mathcal{D}}(x_i) \geq r^+_{\mathcal{D}}(x_i), \quad \tilde{\omega}_{\mathcal{D}}(x_i) \leq \omega_{\mathcal{D}}(x_i), \tag{2.14}$$

the equality sign holds for maximum and minimum point ($\tilde{r}^-_{\mathcal{D}}(x_i) = r^-_{\mathcal{D}}(x_i)$, $\tilde{r}^+_{\mathcal{D}}(x_i) = r^+_{\mathcal{D}}(x_i)$ and $\tilde{\omega}_{\mathcal{D}}(x_i) = \omega_{\mathcal{D}}(x_i)$ for $i \in \{1, k\}$).

Finally, the function value must also satisfy the following constraints

$$\tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i) \leq \tilde{r}^-_{\mathcal{D}}(x_{i+1}), \;\; \tilde{r}^+_{\mathcal{D}}(x_i) \leq \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) \tag{2.15}$$

Since these functions are only defined on S, it is suffice to use $4k$ record to store the summary. Specifically, we need to remember each x_i and the corresponding function values of each x_i.

Definition 2.6.2. *Extension of Function Domains*

Given a quantile summary $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$ defined in Definition 2.6.1, the domain of $\tilde{r}^+_{\mathcal{D}}$, $\tilde{r}^-_{\mathcal{D}}$ and $\tilde{\omega}_{\mathcal{D}}$ were defined only in S. We extend the definition of these functions to $\mathcal{X} \to [0, +\infty)$ as follows

When $y < x_1$:

$$\tilde{r}^-_{\mathcal{D}}(y) = 0, \; \tilde{r}^+_{\mathcal{D}}(y) = 0, \; \tilde{\omega}_{\mathcal{D}}(y) = 0 \tag{2.16}$$

When $y > x_k$:

$$\tilde{r}^-_{\mathcal{D}}(y) = \tilde{r}^+_{\mathcal{D}}(x_k), \; \tilde{r}^+_{\mathcal{D}}(y) = \tilde{r}^+_{\mathcal{D}}(x_k), \; \tilde{\omega}_{\mathcal{D}}(y) = 0 \tag{2.17}$$

When $y \in (x_i, x_{i+1})$ for some i:

$$\begin{aligned} \tilde{r}^-_{\mathcal{D}}(y) &= \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i), \\ \tilde{r}^+_{\mathcal{D}}(y) &= \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}), \\ \tilde{\omega}_{\mathcal{D}}(y) &= 0 \end{aligned} \tag{2.18}$$

Lemma 2.6.1. *Extended Constraint*

The extended definition of $\tilde{r}^-_{\mathcal{D}}$, $\tilde{r}^+_{\mathcal{D}}$, $\tilde{\omega}_{\mathcal{D}}$ satisfies the following constraints

$$\tilde{r}^-_{\mathcal{D}}(y) \leq r^-_{\mathcal{D}}(y), \;\; \tilde{r}^+_{\mathcal{D}}(y) \geq r^+_{\mathcal{D}}(y), \;\; \tilde{\omega}_{\mathcal{D}}(y) \leq \omega_{\mathcal{D}}(y) \tag{2.19}$$

$$\tilde{r}^-_{\mathcal{D}}(y) + \tilde{\omega}_{\mathcal{D}}(y) \leq \tilde{r}^-_{\mathcal{D}}(x), \;\; \tilde{r}^+_{\mathcal{D}}(y) \leq \tilde{r}^+_{\mathcal{D}}(x) - \tilde{\omega}_{\mathcal{D}}(x), \text{ for all } y < x \tag{2.20}$$

Proof. The only non-trivial part is to prove the case when $y \in (x_i, x_{i+1})$:

$$\tilde{r}^-_{\mathcal{D}}(y) = \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i) \leq r^-_{\mathcal{D}}(x_i) + \omega_{\mathcal{D}}(x_i) \leq r^-_{\mathcal{D}}(y)$$

$$\tilde{r}^+_{\mathcal{D}}(y) = \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) \geq r^+_{\mathcal{D}}(x_{i+1}) - \omega_{\mathcal{D}}(x_{i+1}) \geq r^+_{\mathcal{D}}(y)$$

This proves Eq. (2.19). Furthermore, we can verify that

$$\tilde{r}^+_{\mathcal{D}}(x_i) \leq \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) = \tilde{r}^+_{\mathcal{D}}(y) - \tilde{\omega}_{\mathcal{D}}(y)$$

$$\tilde{r}^-_{\mathcal{D}}(y) + \tilde{\omega}_{\mathcal{D}}(y) = \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i) + 0 \leq \tilde{r}^-_{\mathcal{D}}(x_{i+1})$$

$$\tilde{r}^+_{\mathcal{D}}(y) = \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1})$$

Using these facts and transitivity of $<$ relation, we can prove Eq. (2.20) □

We should note that the extension is based on the ground case defined in S, and we do not require extra space to store the summary in order to use the extended definition. We are now ready to introduce the definition of ϵ-approximate quantile summary.

Definition 2.6.3. *ϵ-Approximate Quantile Summary*
Given a quantile summary $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$, we call it is ϵ-approximate summary if for any $y \in \mathcal{X}$

$$\tilde{r}^+_{\mathcal{D}}(y) - \tilde{r}^-_{\mathcal{D}}(y) - \tilde{\omega}_{\mathcal{D}}(y) \leq \epsilon\omega(\mathcal{D}) \tag{2.21}$$

We use this definition since we know that $r^-(y) \in [\tilde{r}^-_{\mathcal{D}}(y), \tilde{r}^+_{\mathcal{D}}(y) - \tilde{\omega}_{\mathcal{D}}(y)]$ and $r^+(y) \in [\tilde{r}^-_{\mathcal{D}}(y) + \tilde{\omega}_{\mathcal{D}}(y), \tilde{r}^+_{\mathcal{D}}(y)]$. Eq. (2.21) *means the we can get estimation of $r^+(y)$ and $r^-(y)$ by error of at most $\epsilon\omega(\mathcal{D})$.*

Lemma 2.6.2. *Quantile summary $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$ is an ϵ-approximate summary if and only if the following two condition holds*

$$\tilde{r}^+_{\mathcal{D}}(x_i) - \tilde{r}^-_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_i) \leq \epsilon\omega(\mathcal{D}) \tag{2.22}$$

$$\tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{r}^-_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_i) \leq \epsilon\omega(\mathcal{D}) \tag{2.23}$$

Proof. The key is again consider $y \in (x_i, x_{i+1})$

$$\tilde{r}^+_{\mathcal{D}}(y) - \tilde{r}^-_{\mathcal{D}}(y) - \tilde{\omega}_{\mathcal{D}}(y) = [\tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1})] - [\tilde{r}^+_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i)] - 0$$

This means the condition in Eq. (2.23) plus Eq.(2.22) can give us Eq. (2.21) □

Property of Extended Function In this section, we have introduced the extension of function $\tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}}$ to $\mathcal{X} \to [0, +\infty)$. The key theme discussed in this section is the relation of *constraints on the original function and constraints on the extended function*. Lemma 2.6.1 and 2.6.2 show that the constraints on the original function can lead to in more general constraints on the extended function. This is a very useful property which will be used in the proofs in later sections.

2.6.2 Construction of Initial Summary

Given a small multi-set $\mathcal{D} = \{(x_1, w_1), (x_2, w_2), \cdots, (x_n, w_n)\}$, we can construct initial summary $Q(\mathcal{D}) = \{S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}}\}$, with S to the set of all values in $\mathcal{D}$ ($S = \{x|(x, w) \in \mathcal{D}\}$), and $\tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}}$ defined to be

$$\tilde{r}^+_{\mathcal{D}}(x) = r^+_{\mathcal{D}}(x), \;\; \tilde{r}^-_{\mathcal{D}}(x) = r^-_{\mathcal{D}}(x), \;\; \tilde{\omega}_{\mathcal{D}}(x) = \omega_{\mathcal{D}}(x) \text{ for } x \in S \tag{2.24}$$

The constructed summary is 0-approximate summary, since it can answer all the queries accurately. The constructed summary can be feed into future operations described in the latter sections.

2.6.3 Merge Operation

In this section, we define how we can merge the two summaries together. Assume we have $Q(\mathcal{D}_1) = (S_1, \tilde{r}^+_{\mathcal{D}_1}, \tilde{r}^-_{\mathcal{D}_1}, \tilde{\omega}_{\mathcal{D}_1})$ and $Q(\mathcal{D}_2) = (S_2, \tilde{r}^+_{\mathcal{D}_1}, \tilde{r}^-_{\mathcal{D}_2}, \tilde{\omega}_{\mathcal{D}_2})$ quantile summary of two dataset $\mathcal{D}_1$ and $\mathcal{D}_2$. Let $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$, and define the merged summary $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$ as follows.

$$S = \{x_1, x_2 \cdots, x_k\}, x_i \in S_1 \text{ or } x_i \in S_2 \tag{2.25}$$

The points in S are combination of points in S_1 and S_2. And the function $\tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}}$ are defined to be

$$\tilde{r}^-_{\mathcal{D}}(x_i) = \tilde{r}^-_{\mathcal{D}_1}(x_i) + \tilde{r}^-_{\mathcal{D}_2}(x_i) \tag{2.26}$$

$$\tilde{r}^+_{\mathcal{D}}(x_i) = \tilde{r}^+_{\mathcal{D}_1}(x_i) + \tilde{r}^+_{\mathcal{D}_2}(x_i) \tag{2.27}$$

$$\tilde{\omega}_{\mathcal{D}}(x_i) = \tilde{\omega}_{\mathcal{D}_1}(x_i) + \tilde{\omega}_{\mathcal{D}_2}(x_i) \tag{2.28}$$

Here we use functions defined on $S \to [0, +\infty)$ on the left sides of equalities and use the extended function definitions on the right sides.

Due to additive nature of r^+, r^- and ω, which can be formally written as

$$\begin{aligned} r^-_{\mathcal{D}}(y) =& r^-_{\mathcal{D}_1}(y) + r^-_{\mathcal{D}_2}(y), \\ r^+_{\mathcal{D}}(y) =& r^+_{\mathcal{D}_1}(y) + r^+_{\mathcal{D}_2}(y), \\ \omega_{\mathcal{D}}(y) =& \omega_{\mathcal{D}_1}(y) + \omega_{\mathcal{D}_2}(y), \end{aligned} \tag{2.29}$$

and the extended constraint property in Lemma 2.6.1, we can verify that $Q(\mathcal{D})$ satisfies all the constraints in Definition 2.6.1. Therefore it is a valid quantile summary.

Lemma 2.6.3. *The combined quantile summary satisfies*

$$\tilde{r}^-_{\mathcal{D}}(y) = \tilde{r}^-_{\mathcal{D}_1}(y) + \tilde{r}^-_{\mathcal{D}_2}(y) \tag{2.30}$$

$$\tilde{r}^+_{\mathcal{D}}(y) = \tilde{r}^+_{\mathcal{D}_1}(y) + \tilde{r}^+_{\mathcal{D}_2}(y) \tag{2.31}$$

$$\tilde{\omega}_{\mathcal{D}}(y) = \tilde{\omega}_{\mathcal{D}_1}(y) + \tilde{\omega}_{\mathcal{D}_2}(y) \tag{2.32}$$

for all $y \in \mathcal{X}$

This can be obtained by straight-forward application of Definition 2.6.2.

Theorem 2.6.1. *If* $Q(\mathcal{D}_1)$ *is* ϵ_1*-approximate summary, and* $Q(\mathcal{D}_2)$ *is* ϵ_2*-approximate summary. Then the merged summary* $Q(\mathcal{D})$ *is* $\max(\epsilon_1, \epsilon_2)$*-approximate summary.*

Proof. For any $y \in \mathcal{X}$, we have

$$\begin{aligned} & \tilde{r}^+_{\mathcal{D}}(y) - \tilde{r}^-_{\mathcal{D}}(y) - \tilde{\omega}_{\mathcal{D}}(y) \\ =& [\tilde{r}^+_{\mathcal{D}_1}(y) + \tilde{r}^+_{\mathcal{D}_2}(y)] - [\tilde{r}^-_{\mathcal{D}_1}(y) + \tilde{r}^-_{\mathcal{D}_2}(y)] - [\tilde{\omega}_{\mathcal{D}_1}(y) + \tilde{\omega}_{\mathcal{D}_2}(y)] \\ \leq& \epsilon_1 \omega(\mathcal{D}_1) + \epsilon_2 \omega(\mathcal{D}_2) \leq \max(\epsilon_1, \epsilon_2) \omega(\mathcal{D}_1 \cup \mathcal{D}_2) \end{aligned}$$

Here the first inequality is due to Lemma 2.6.3. □

Algorithm 4: Query Function $g(Q, d)$

Input: d: $0 \leq d \leq \omega(\mathcal{D})$

Input: $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$ where $S = x_1, x_2, \cdots, x_k$

if $d < \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_1) + \tilde{r}^+_{\mathcal{D}}(x_1)]$ **then return** x_1 ;

if $d \geq \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_k) + \tilde{r}^+_{\mathcal{D}}(x_k)]$ **then return** x_k ;

Find i such that $\frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{r}^+_{\mathcal{D}}(x_i)] \leq d < \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_{i+1}) + \tilde{r}^+_{\mathcal{D}}(x_{i+1})]$

if $2d < \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i) + \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1})$ **then**
  return x_i

else
  return x_{i+1}

end

2.6.4 Prune Operation

Before we start discussing the prune operation, we first introduce a query function $g(Q, d)$. The definition of function is shown in Algorithm 4. For a given rank d, the function returns a x whose rank is close to d. This property is formally described in the following Lemma.

Lemma 2.6.4. *For a given ϵ-approximate summary $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$, $x^* = g(Q, d)$ satisfies the following property*

$$\begin{aligned} d &\geq \tilde{r}^+_{\mathcal{D}}(x^*) - \tilde{\omega}_{\mathcal{D}}(x^*) - \frac{\epsilon}{2}\omega(\mathcal{D}) \\ d &\leq \tilde{r}^-_{\mathcal{D}}(x^*) + \tilde{\omega}_{\mathcal{D}}(x^*) + \frac{\epsilon}{2}\omega(\mathcal{D}) \end{aligned} \tag{2.33}$$

Proof. We need to discuss four possible cases

- $d < \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_1) + \tilde{r}^+_{\mathcal{D}}(x_1)]$ and $x^* = x_1$. Note that the rank information for x_1 is accurate

($\tilde{\omega}_{\mathcal{D}}(x_1) = \tilde{r}^+_{\mathcal{D}}(x_1) = \omega(x_1)$, $\tilde{r}^-_{\mathcal{D}}(x_1) = 0$), we have

$$\begin{aligned} d &\geq 0 - \frac{\epsilon}{2}\omega(\mathcal{D}) = \tilde{r}^+_{\mathcal{D}}(x_1) - \tilde{\omega}_{\mathcal{D}}(x_1) - \frac{\epsilon}{2}\omega(\mathcal{D}) \\ d &< \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_1) + \tilde{r}^+_{\mathcal{D}}(x_1)] \\ &\leq \tilde{r}^-_{\mathcal{D}}(x_1) + \tilde{r}^+_{\mathcal{D}}(x_1) \\ &= \tilde{r}^-_{\mathcal{D}}(x_1) + \tilde{\omega}^+_{\mathcal{D}}(x_1) \end{aligned}$$

- $d \geq \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_k) + \tilde{r}^+_{\mathcal{D}}(x_k)]$ and $x^* = x_k$, then

$$\begin{aligned} d &\geq \frac{1}{2}[\tilde{r}^-_{\mathcal{D}}(x_k) + \tilde{r}^+_{\mathcal{D}}(x_k)] \\ &= \tilde{r}^+_{\mathcal{D}}(x_k) - \frac{1}{2}[\tilde{r}^+_{\mathcal{D}}(x_k) - \tilde{r}^-_{\mathcal{D}}(x_k)] \\ &= \tilde{r}^+_{\mathcal{D}}(x_k) - \frac{1}{2}\tilde{\omega}_{\mathcal{D}}(x_k) \\ d &< \omega(\mathcal{D}) + \frac{\epsilon}{2}\omega(\mathcal{D}) = \tilde{r}^-_{\mathcal{D}}(x_k) + \tilde{\omega}_{\mathcal{D}}(x_k) + \frac{\epsilon}{2}\omega(\mathcal{D}) \end{aligned}$$

- $x^* = x_i$ in the general case, then

$$\begin{aligned} 2d &< \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i) + \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) \\ &= 2[\tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i)] + [\tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) - \tilde{r}^-_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_i)] \\ &\leq 2[\tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i)] + \epsilon\omega(\mathcal{D}) \\ 2d &\geq \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{r}^+_{\mathcal{D}}(x_i) \\ &= 2[\tilde{r}^+_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_i)] - [\tilde{r}^+_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_i) - \tilde{r}^-_{\mathcal{D}}(x_i)] + \tilde{\omega}_{\mathcal{D}}(x_i) \\ &\geq 2[\tilde{r}^+_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_i)] - \epsilon\omega(\mathcal{D}) + 0 \end{aligned}$$

- $x^* = x_{i+1}$ in the general case

$$\begin{aligned}
2d &\geq \tilde{r}^-_{\mathcal{D}}(x_i) + \tilde{\omega}_{\mathcal{D}}(x_i) + \tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) \\
&= 2[\tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1})] \\
&\quad - [\tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) - \tilde{r}^-_{\mathcal{D}}(x_i) - \tilde{\omega}_{\mathcal{D}}(x_i)] \\
&\geq 2[\tilde{r}^+_{\mathcal{D}}(x_{i+1}) + \tilde{\omega}_{\mathcal{D}}(x_{i+1})] - \epsilon\omega(\mathcal{D}) \\
2d &\leq \tilde{r}^-_{\mathcal{D}}(x_{i+1}) + \tilde{r}^+_{\mathcal{D}}(x_{i+1}) \\
&= 2[\tilde{r}^-_{\mathcal{D}}(x_{i+1}) + \tilde{\omega}_{\mathcal{D}}(x_{i+1})] \\
&\quad + [\tilde{r}^+_{\mathcal{D}}(x_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) - \tilde{r}^-_{\mathcal{D}}(x_{i+1})] - \tilde{\omega}_{\mathcal{D}}(x_{i+1}) \\
&\leq 2[\tilde{r}^-_{\mathcal{D}}(x_{i+1}) + \tilde{\omega}_{\mathcal{D}}(x_{i+1})] + \epsilon\omega(\mathcal{D}) - 0
\end{aligned}$$

□

Now we are ready to introduce the prune operation. Given a quantile summary $Q(\mathcal{D}) = (S, \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$ with $S = \{x_1, x_2, \cdots, x_k\}$ elements, and a memory budget b. The prune operation creates another summary $Q'(\mathcal{D}) = (S', \tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}})$ with $S' = \{x'_1, x'_2, \cdots, x'_{b+1}\}$, where x'_i are selected by query the original summary such that

$$x'_i = g\left(Q, \frac{i-1}{b}\omega(\mathcal{D})\right).$$

The definition of $\tilde{r}^+_{\mathcal{D}}, \tilde{r}^-_{\mathcal{D}}, \tilde{\omega}_{\mathcal{D}}$ in Q' is copied from original summary Q, by restricting input domain from S to S'. There could be duplicated entries in the S'. These duplicated entries can be safely removed to further reduce the memory cost. Since all the elements in Q' comes from Q, we can verify that Q' satisfies all the constraints in Definition 2.6.1 and is a valid quantile summary.

Theorem 2.6.2. *Let $Q'(\mathcal{D})$ be the summary pruned from an ϵ-approximate quantile summary $Q(\mathcal{D})$ with b memory budget. Then $Q'(\mathcal{D})$ is a $(\epsilon + \frac{1}{b})$-approximate summary.*

Proof. We only need to prove the property in Eq. (2.23) for Q'. Using Lemma 2.6.4, we have

$$\begin{aligned}
\frac{i-1}{b}\omega(\mathcal{D}) + \frac{\epsilon}{2}\omega(\mathcal{D}) &\geq \tilde{r}^+_{\mathcal{D}}(x'_i) - \tilde{\omega}_{\mathcal{D}}(x'_i) \\
\frac{i-1}{b}\omega(\mathcal{D}) - \frac{\epsilon}{2}\omega(\mathcal{D}) &\leq \tilde{r}^-_{\mathcal{D}}(x'_i) + \tilde{\omega}_{\mathcal{D}}(x'_i)
\end{aligned}$$

Combining these inequalities gives

$$\begin{aligned}&\tilde{r}^{+}_{\mathcal{D}}(x'_{i+1}) - \tilde{\omega}_{\mathcal{D}}(x'_{i+1}) - \tilde{r}^{-}_{\mathcal{D}}(x'_i) - \tilde{\omega}_{\mathcal{D}}(x'_i)\\ \leq&[\frac{i}{b}\omega(\mathcal{D}) + \frac{\epsilon}{2}\omega(\mathcal{D})] - [\frac{i-1}{b}\omega(\mathcal{D}) - \frac{\epsilon}{2}\omega(\mathcal{D})] = (\frac{1}{b} + \epsilon)\omega(\mathcal{D})\end{aligned}$$

□

Chapter 3

MXNET: A SCALABLE AND FLEXIBLE DEEP LEARNING SYSTEM

The scale and complexity of machine learning algorithms are becoming increasingly large. winners employ neural networks with very deep layers, requiring billions of floating-point operations to process one single sample. The rise of structural and computational complexity poses interesting challenges to learning system design and implementation. Most learning systems embed a domain-specific language (DSL) into a host language (e.g. Python, Lua, C++). Possible programming paradigms range from *imperative*, where the user specifies exactly "how" computation needs to be performed, and *declarative*, where the user specification focuses on "what" to be done. Examples of imperative programming include numpy and Matlab.

In this chapter, we describe MXNet, flexible and efficient deep learning system. MXNet intends to blend blend advantages of imperative and declarative programs. Declarative programming offers clear boundary on the global computation graph, discovering more optimization opportunity, whereas imperative programs offers more flexibility. We will start the chapter by discussing the programming model. Then we will discuss memory optimization methods to scale training under limited device memory contraints.

3.1 Programming Interface

3.1.1 `Symbol`*: Declarative Symbolic Expressions*

MXNet uses multi-output symbolic expressions, `Symbol`, declare the computation graph. Symbols are composited by operators, such as simple matrix operations (e.g. "+"), or a complex neural network layer (e.g. convolution layer). An operator can take several input variables, produce more than one output variables, and have internal state variables. A variable can be either free, which we can bind with value later, or an output of another symbol. Figure 3.1 shows the construction of

```
using MXNet
mlp = @mx.chain mx.Variable(:data) =>
  mx.FullyConnected(num_hidden=64) =>
  mx.Activation(act_type=:relu)    =>
  mx.FullyConnected(num_hidden=10) =>
  mx.Softmax()
```

Figure 3.1: Symbol expression construction in Julia.

```
>>> import mxnet as mx
>>> a = mx.nd.ones((2, 3),
... mx.gpu())
>>> print (a * 2).asnumpy()
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
```

Figure 3.2: NDArray interface in Python

a multi-layer perception symbol by chaining a variable , which presents the input data, and several layer operators.

To evaluate a symbol we need to bind the free variables with data and declare the required outputs. Beside evaluation ("forward"), a symbol supports auto symbolic differentiation ("backward"). Other functions, such as load, save, memory estimation, and visualization, are also provided for symbols.

3.1.2 `NDArray`: Imperative Tensor Computation

MXNet also offers `NDArray` with imperative tensor computation to fill the gap between the declarative symbolic expression and the host language. Figure 3.2 shows an example which does matrix-constant multiplication on GPU and then prints the results by `numpy.ndarray`.

`NDArray` abstraction works seamlessly with the executions declared by `Symbol`, we can mix the imperative tensor computation of the former with the latter. For example, given a symbolic neural network and the weight updating function, e.g. $w = w - \eta g$. Then we can implement the gradient descent by

```
while(1) { net.foward_backward(); net.w -= eta * net.g };
```

The above is as efficient as the implementation using a single but often much more complex symbolic expression. The reason is that MXNet uses lazy evaluation of `NDArray` and the backend engine can correctly resolve the data dependency between the two.

3.1.3 Dependency Engine

In MXNet, each source units, including `NDArray`, random number generator and temporal space, is registered to the engine with a unique tag. Any operations, such as a matrix operation or data communication, is then pushed into the engine with specifying the required resource tags. The engine continuously schedules the pushed operations for execution if dependencies are resolved. Since there usually exists multiple computation resources such as CPUs, GPUs, and the memory/PCIe buses, the engine uses multiple threads to scheduling the operations for better resource utilization and parallelization.

Different to most dataflow engines, our engine tracks mutation operations as an existing resource unit. That is, ours supports the specification of the tags that a operation will *write* in addition to *read*. This enables scheduling of array mutations as in numpy and other tensor libraries. It also enables easier memory reuse of parameters, by representing parameter updates as mutating the parameter arrays. It also makes scheduling of some special operations easier. For example, when generating two random numbers with the same random seed, we can inform the engine they will write the seed so that they should not be executed in parallel. This helps reproducibility.

3.2 Memory Optimization with Computation Graph

One common trend in deep learning is to use deeper architectures [90, 58, 49, 41] to capture the complex patterns in a large amount of training data. Since the cost of storing feature maps and their gradients scales linearly with the depth of network, our capability of exploring deeper models is limited by the device (usually a GPU) memory. For example, we already run out of memories in one of the current state-of-art models as described in [42]. In the long run, an ideal machine learning system should be able to continuously learn from an increasing amount of training data. Since the optimal model size and complexity often grows with more training data, it is very important to have memory-efficient training algorithms. In the next few sections of this chapter, we discuss how to optimize memory usage of deep learning models.

The trade-off between memory and computation has been a long standing topic in systems

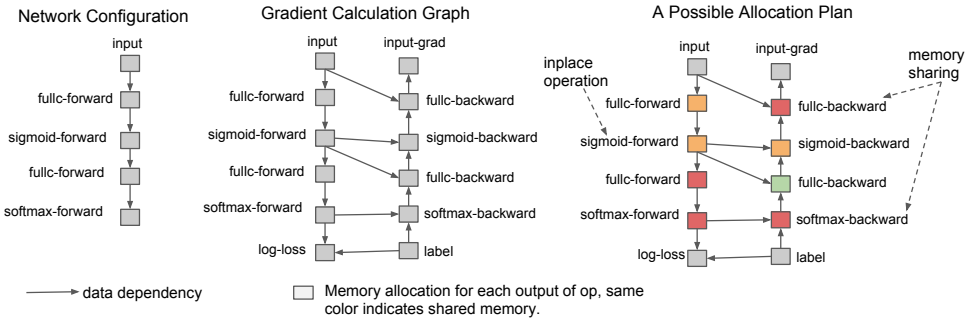


Figure 3.3: Computation graph and possible memory allocation plan of a two layer fully connected neural network training procedure. Each node represents an operation and each edge represents a dependency between the operations. The nodes with the same color share the memory to store output or back-propagated gradient in each operator. To make the graph more clearly, we omit the weights and their output gradient nodes from the graph and assume that the gradient of weights are also calculated during backward operations. We also annotate two places where the in-place and sharing strategies are used.

research. Although not widely known, the idea of dropping intermediate results is also known as gradient checkpointing technique in automatic differentiation literature [40]. We bring this idea to neural network gradient graph construction for general deep neural networks. Through the discussion with our colleagues [110], we know that the idea of dropping computation has been applied in some limited specific use-cases.

We start by reviewing the concept of computation graph and the memory optimization techniques. Some of these techniques are already used by existing frameworks. A computation graph consists of operational nodes and edges that represent the dependencies between the operations. Fig. 3.3 gives an example of the computation graph of a two-layer fully connected neural network. Here we use coarse grained forward and backward operations to make the graph simpler. We further simplify the graph by hiding the weight nodes and gradients of the weights. A computation graph used in practice can be more complicated and contains mixture of fine/coarse grained operations. The analysis presented in this chapter can be directly used in those more general cases.

Once the network configuration (forward graph) is given, we can construct the corresponding backward pathway for gradient calculation. A backward pathway can be constructed by traversing the configuration in reverse topological order, and apply the backward operators as in normal back-propagation algorithm. The backward pathway in Fig. 3.3 represents the gradient calculation steps *explicitly*, so that the gradient calculation step in training is simplified to just a forward pass on the entire computation graph (including the gradient calculation pathway). Explicit gradient path also offers some other benefits (e.g. being able to calculate higher order gradients), which is beyond our scope and will not be covered in this chapter.

When training a deep convolutional/recurrent network, a great proportion of the memory is usually used to store the intermediate outputs and gradients. Each of these intermediate results corresponds to a node in the graph. A smart allocation algorithm is able to assign the least amount of memory to these nodes by sharing memory when possible. Fig. 3.3 shows a possible allocation plan of the example two-layer neural network. Two types of memory optimizations can be used

- *Inplace operation*: Directly store the output values to memory of a input value.

- *Memory sharing*: Memory used by intermediate results that are no longer needed can be recycled and used in another node.

Allocation plan in Fig. 3.3 contains examples of both cases. The first sigmoid transformation is carried out using inplace operation to save memory, which is then reused by its backward operation. The storage of the softmax gradient is shared with the gradient by the first fully connected layer. Ad hoc application of these optimizations can leads to errors. For example, if the input of an operation is still needed by another operation, applying inplace operation on the input will lead to a wrong result.

We can only share memory between the nodes whose lifetime do not overlap. There are multiple ways to solve this problem. One option is to construct the conflicting graph of with each variable as node and edges between variables with overlapping lifespan and then run a graph-coloring algorithm. This will cost $O(n^2)$ computation time. We adopt a simpler heuristic with

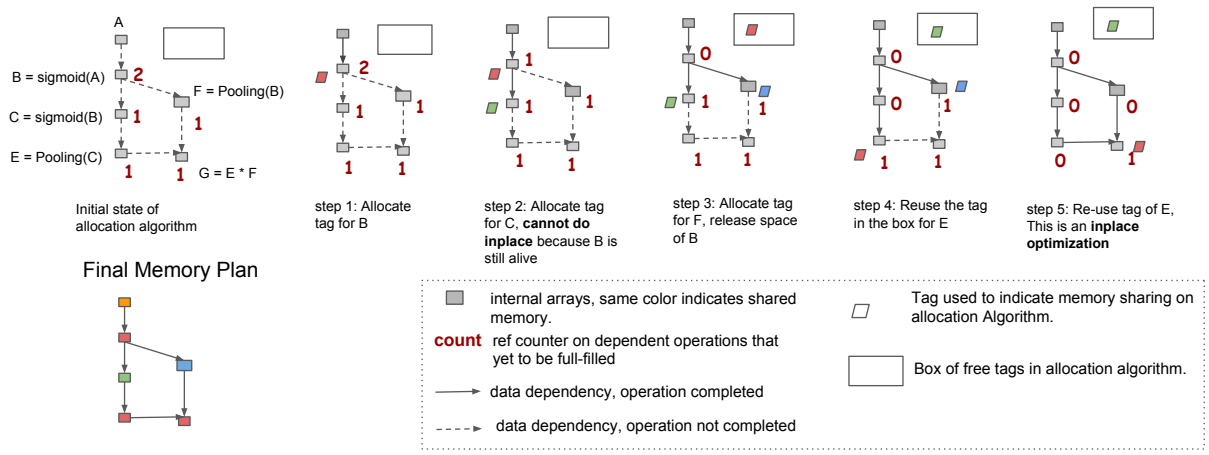


Figure 3.4: Memory allocation algorithm on computation graph. Each node associated with a liveness counter to count on operations to be full-filled. A temporal tag is used to indicate memory sharing. Inplace operation can be carried out when the current operations is the only one left (input of counter equals 1). The tag of a node can be recycled when the node's counter goes to zero.

only $O(n)$ time. The algorithm is demonstrated in Fig. 3.4. It traverses the graph in topological order, and uses a counter to indicate the liveness of each record. An inplace operation can happen when there is no other pending operations that depend on its input. Memory sharing happens when a recycled tag is used by another node. This can also serve as a dynamic runtime algorithm that traverses the graph, and use a garbage collector to recycle the outdated memory. We use this as a static memory allocation algorithm, to allocate the memory to each node before the execution starts, in order to avoid the overhead of garbage collection during runtime.

Guidelines for Deep Learning Frameworks As we can see from the algorithm demonstration graph in Fig. 3.4. The data dependency causes longer lifespan of each output and increases the memory consumption of big network. It is important for deep learning frameworks to

- Declare the dependency requirements of gradient operators in minimum manner.

- Apply liveness analysis on the dependency information and enable memory sharing.

It is important to declare minimum dependencies. For example, the allocation plan in Fig. 3.3 won't be possible if `sigmoid-backward` also depend on the output of the first `fullc-forward`. The dependency analysis can usually reduce the memory footprint of deep network prediction of a n layer network from $O(n)$ to nearly $O(1)$ because sharing can be done between each intermediate results. The technique also helps to reduce the memory footprint of training, although only up to a constant factor.

3.3 Training Deep Neural Networks with Sublinear Memory Cost

3.3.1 General Methodology

The techniques introduced in Sec. 3.2 can reduce the memory footprint for both training and prediction of deep neural networks. However, due to the fact that most gradient operators will depend on the intermediate results of the forward pass, we still need $O(n)$ memory for intermediate results to train a n layer convolutional network or a recurrent neural networks with a sequence of length n. In order to further reduce the memory, we propose to *drop some of the intermediate results*, and recover them from an extra forward computation when needed.

More specifically, during the backpropagation phase, we can re-compute the dropped intermediate results by running forward from the closest recorded results. To present the idea more clearly, we show a simplified algorithm for a linear chain feed-forward neural network in Alg. 5. Specifically, the neural network is divided into several segments. The algorithm only remembers the output of each segment and drops all the intermediate results within each segment. The dropped results are recomputed at the segment level during back-propagation. As a result, we only need to pay the memory cost to store the outputs of each segment plus the maximum memory cost to do backpropagation on each segment.

Alg. 5 can also be generalized to common computation graphs as long as we can divide the graph into segments. However, there are two drawbacks on directly applying Alg. 5: 1) users have to manually divide the graph and write customized training loop; 2) we cannot benefit from other memory optimizations presented in Sec 3.2. We solve this problem by introducing a general

Algorithm 5: Backpropagation with Data Dropping in a Linear Chain Network

```
v ← input
for k = 1 to length(segments) do
    temp[k] ← v
    for i = segments[k].begin to segments[k].end − 1 do
        v ← layer[i].forward(v)
    end
end
g ← gradient(v, label)
for k = length(segments) to 1 do
    v ← temp[k]
    localtemp ← empty hashtable
    for i = segments[k].begin to segments[k].end − 1 do
        localtemp[i] ← v
        v ← layer[i].forward(v)
    end
    for i = segments[k].end − 1 to segments[k].begin do
        g ← layer[i].backward(g, localtemp[i])
    end
end
```

gradient graph construction algorithm that uses essentially the same idea. The algorithm is given in Alg. 6. In this algorithm, the user specify a function $m : \mathcal{V} \to \mathbb{N}$ on the nodes of a computation graph to indicate how many times a result can be recomputed. We call m the mirror count function as the re-computation is essentially duplicating (mirroring) the nodes. When all the mirror counts are set to 0, the algorithm degenerates to normal gradient graph. To specify re-computation pattern in Alg. 6, the user only needs to set the $m(v) = 1$ for nodes within each segment and $m(v) = 0$ for the output node of each segment. The mirror count can also be larger than 1, which leads to a recursive generalization to be discussed in Sec 3.3.4. Fig. 3.5 shows an example of memory optimized gradient graph. Importantly, Alg. 6 also outputs a traversal order for the computation,

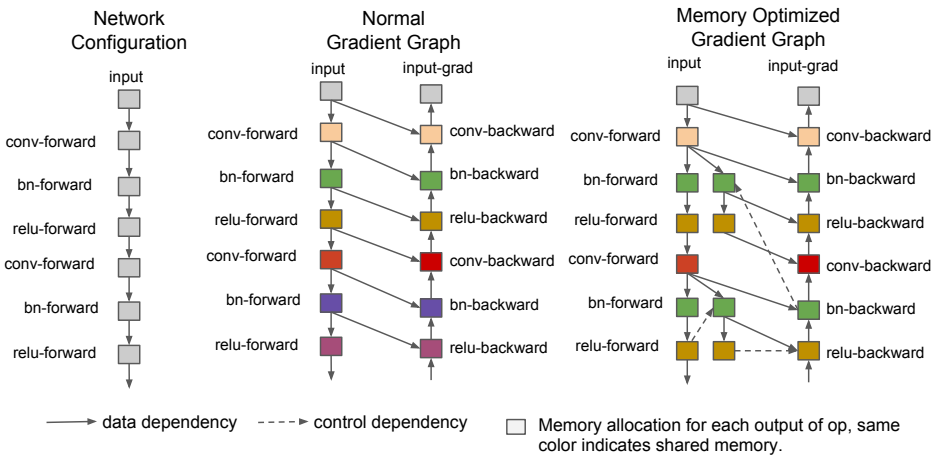


Figure 3.5: Memory optimized gradient graph generation example. The forward path is *mirrored* to represent the re-computation happened at gradient calculation. User specifies the mirror factor to control whether a result should be dropped or kept.

so the memory usage can be optimized. Moreover, this traversal order can help introduce control flow dependencies for frameworks that depend on runtime allocation.

3.3.2 Drop the Results of Low Cost Operations

One quick application of the general methodology is to drop the results of low cost operations and keep the results that are time consuming to compute. This is usually useful in a `Conv-BN-Act` pipeline in convolutional neural networks. We can always keep the result of convolution, but drop the result of the batch normalization, activation function and pooling. In practice this will translate to a memory saving with little computation overhead, as the computation for both batch normalization and activation functions are cheap.

3.3.3 An $O(\sqrt{n})$ Memory Cost Algorithm

Alg. 6 provides a general way to trade computation for memory. It remains to ask which intermediate result we should keep and which ones to re-compute. Assume we divide the n network into

Algorithm 6: Memory Optimized Gradient Graph Construction

Input: $G = (V, \text{pred})$, input computation graph, the pred$[v]$ gives the predecessors array of node v.

Input: gradient(succ_grads, output, inputs), symbolic gradient function that creates a gradient node given successor gradients and output and inputs

Input: $m : V \to \mathbb{N}^+$, $m(v)$ gives how many time node v should be duplicated, $m(v) = 0$ means do no drop output of node v.

$a[v] \leftarrow v$ for $v \in V$

for $k = 1$ ***to*** $\max_{v \in V} m(v)$ **do**

  for v *in topological-order*(V) **do**

    if $k \leq m(v)$ **then**

      $a[v] \leftarrow$ new node, same operator as v

      pred$[a[v]] \leftarrow \cup_{u \in \text{pred}[v]} \{a[u]\}$

    end

  end

end

$V' \leftarrow$ topological-order(V)

for v *in reverse-topological-order*(V) **do**

  $g[v] \leftarrow gradient([g[v] \text{ for v in } successor(v)], a[v], [a[v] \text{ for v in } pred[v]])$

  $V' \leftarrow append(V', \text{topological-order}(acenstors(g[v])) - V')$

end

Output: $G' = (V', \text{pred})$ the new graph, the order in V' gives the logical execution order.

k segments the memory cost to train this network is given as follows.

$$\text{cost-total} = \max_{i=1,\ldots,k} \text{cost-of-segment}(i) + O(k) = O\left(\frac{n}{k}\right) + O(k) \tag{3.1}$$

The first part of the equation is the memory cost to run back-propagation on each of the segment. Given that the segment is equally divided, this translates into $O(n/k)$ cost. The second part of equation is the cost to store the intermediate outputs between segments. Setting $k = \sqrt{n}$, we get the cost of $O(2\sqrt{n})$. This algorithm *only requires an additional forward pass* during training, but reduces the memory cost to be *sub-linear*. Since the backward operation is nearly twice as time

Algorithm 7: Memory Planning with Budget

Input: $G = (V, \text{pred})$, input computation graph.

Input: $C \subset V$, candidate stage splitting points, we will search splitting points over $v \subset C$

Input: B, approximate memory budget. We can search over B to optimize the memory allocation.

$temp \leftarrow 0, x \leftarrow 0, y \leftarrow 0$

for v *in topological-order*(V) **do**

  $temp \leftarrow temp +$ size-of-output(v)

  if $v \in C$ *and* $temp > B$ **then**

    $x \leftarrow x +$ size-of-output(v), $y \leftarrow max(y, temp)$

    $m(v) = 0$, $temp \leftarrow 0$

  else

    $m(v) = 1$

  end

end

Output: x approximate cost to store inter-stage feature maps

Output: y approximate memory cost for each sub stage

Output: m the mirror plan to feed to Alg. 6

consuming as the forward one, it only slows down the computation by a small amount.

In the most general case, the memory cost of each layer is not the same, so we cannot simply set $k = \sqrt{n}$. However, the trade-off between the intermediate outputs and the cost of each stage still holds. In this case, we use Alg. 7 to do a greedy allocation with a given budget for the memory cost within each segment as a single parameter B. Varying B gives us various allocation plans that either assign more memory to the intermediate outputs, or to computation within each stage. When we do static memory allocation, we can get the *exact memory cost* given each allocation plan. We can use this information to do a heuristic search over B to find optimal memory plan that balances the cost of the two. The details of the searching step is presented in the supplementary material. We find this approach works well in practice. We can also generalize this algorithm by considering the cost to run each operation to try to keep time consuming operations when possible.

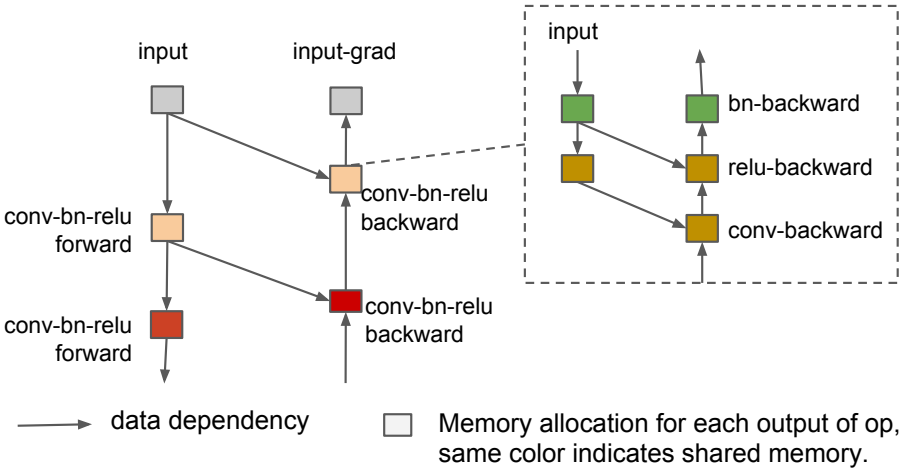


Figure 3.6: Recursion view of the memory optimized allocations. The segment can be viewed as a single operator that combines all the operators within the segment. Inside each operator, a sub-graph as executed to calculate the gradient.

3.3.4 More General View: Recursion and Subroutine

In this section, we provide an alternative view of the memory optimization scheme described above. Specifically, we can view each segment as a bulk operator that combines all the operations inside the segment together. The idea is illustrated in Fig. 3.6. The combined operator calculates the gradient by executing over the sub-graph that describes its internal computation. This view allows us to treat a series of operations as subroutines. The optimization within the sub-graph does not affect the external world. As a result, we can recursively apply our memory optimization scheme to each sub-graph.

Pay Even Less Memory with Recursion Let $g(n)$ to be the memory cost to do forward and backward pass on a n layer neural network. Assume that we store k intermediate results in the graph and apply the same strategy recursively when doing forward and backward pass on the sub-path. We have the following recursion formula.

$$g(n) = k + g\left(n/(k+1)\right) \tag{3.2}$$

Solving this recursion formula gives us

$$g(n) = k \log_{k+1}(n) \tag{3.3}$$

As a special case, if we set $k = 1$, we get $g(n) = \log_2 n$. This is interesting conclusion as all the existing implementations takes $O(n)$ memory in feature map to train a n layer neural network. This will require $O(\log_2 n)$ cost forward pass cost, so may not be used commonly. But it demonstrates how we can trade memory even further by using recursion.

3.3.5 Guideline for Deep Learning Frameworks

In this section, we have shown that it is possible to trade computation for memory and combine it with the system optimizations proposed in Sec 3.2. It is helpful for deep learning frameworks to

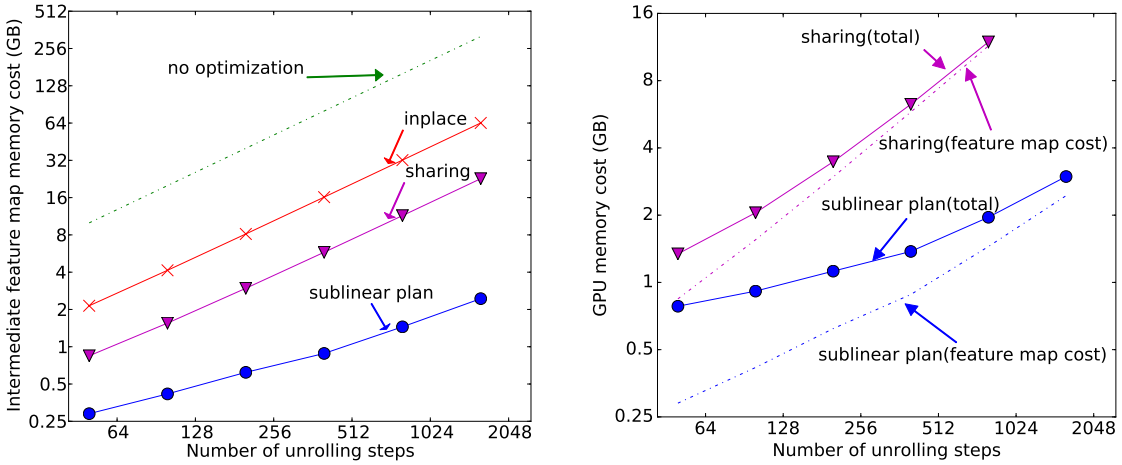
- Enable option to drop result of low cost operations.
- Provide planning algorithms to give efficient memory plan.
- Enable user to set the mirror attribute in the computation graph for memory optimization.

While the last option is not strictly necessary, providing such interface enables user to hack their own memory optimizers and encourages future researches on the related directions.

3.4 Evaluations

3.4.1 Experiment Setup

We evaluate the memory cost of storing intermediate feature maps using the methods described in this chapter. Note that the memory cost of parameters and temporal memory (e.g. required by convolution) are not part of the memory cost report. We also record the runtime total memory cost by running training steps on a Titan X GPU. Note that all the memory optimizations proposed in this chapter gives equivalent weight gradient for training and can always be safely applied. We compare the following memory allocation algorithms

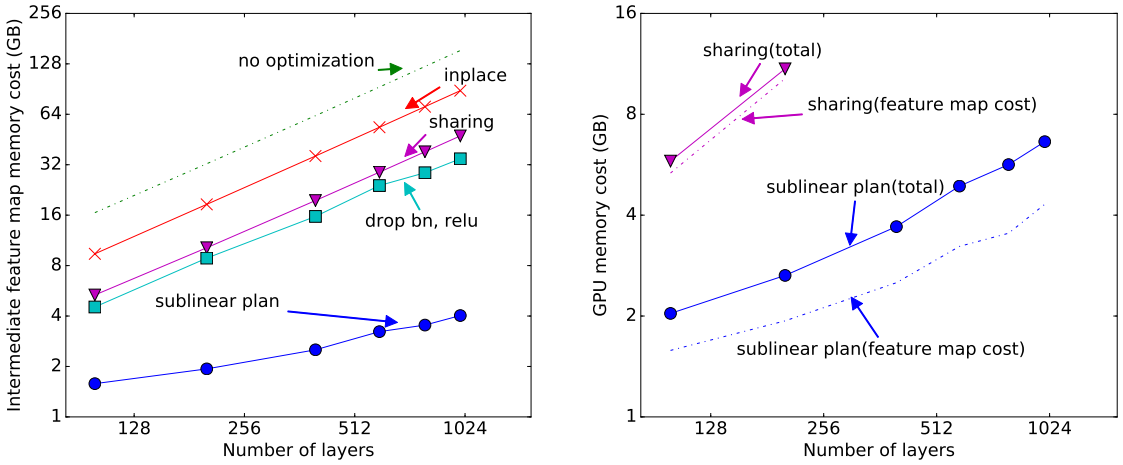


(a) Feature map memory cost estimation

(b) Runtime total memory cost

Figure 3.7: The memory cost of different memory allocation strategies on LSTM configurations. System optimization gives a lot of memory saving on the LSTM graph, which contains a lot of fine grained operations. The sub-linear plan can give more than 4x reduction over the optimized plan that do not trade computation with memory.

- *no optimization*, directly allocate memory to each node without any optimization.
- *inplace*, enable inplace optimization when possible.
- *sharing*, enable inplace optimization as well as sharing. This represents all the system optimizations presented at Sec. 3.2.
- *drop bn-relu*, apply all system optimizations, drop result of batch norm and relu, this is only shown in convolutional net benchmark.
- *sublinear plan*, apply all system optimizations, use plan search with Alg 7 to trade computation with memory.



(a) Feature map memory cost estimation

(b) Runtime total memory cost

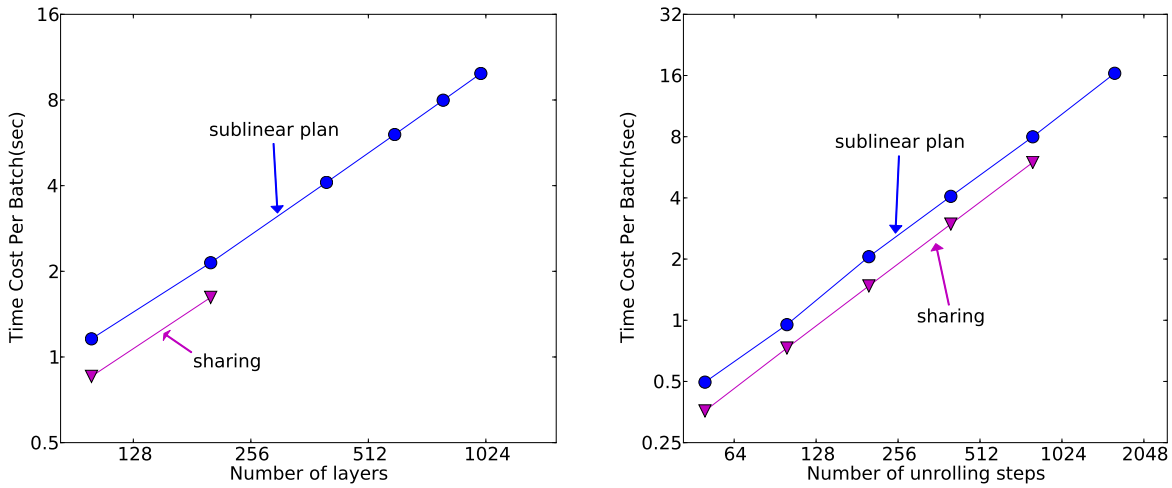
Figure 3.8: The memory cost of different allocation strategies on deep residual net configurations. The feature map memory cost is generated from static memory allocation plan. We also use nvidia-smi to measure the total memory cost during runtime (the missing points are due to out of memory). The figures are in log-scale, so $y = \alpha x^{\beta}$ will translate to $\log(y) = \beta \log(x) + \log \alpha$. We can find that the graph based allocation strategy indeed help to reduce the memory cost by a factor of two to three. More importantly, the sub-linear planning algorithm indeed gives sub-linear memory trend with respect to the workload. The real runtime result also confirms that we can use our method to greatly reduce memory cost deep net training.

3.4.2 Deep Convolutional Network

We first evaluate the proposed method on convolutional neural network for image classification. We use deep residual network architecture [42] (ResNet), which gives the state of art result on this task. Specifically, we use 32 batch size and set input image shape as $(3, 224, 224)$. We generate different depth configuration of ResNet [1] by increasing the depth of each residual stage.

We show the results in Fig. 3.8. We can find that the system optimizations introduced in Sec. 3.2

[1] We count a conv-bn-relu as one layer



(a) ResNet

(b) LSTM

Figure 3.9: The runtime speed of different allocation strategy on the two settings. The speed is measured by a running 20 batches on a Titan X GPU. We can see that using sub-linear memory plan incurs roughly 30% of additional runtime cost compared to linear memory allocation. The general trend of speed vs workload remains linear for both strategies.

can help to reduce the memory cost by factor of two to three. However, the memory cost after optimization still exhibits a linear trend with respect to number of layers. Even with all the system optimizations, it is only possible to train a 200 layer ResNet with the best GPU we can get. On the other hand, the proposed algorithm gives a sub-linear trend in terms of number of layers. By trade computation with memory, we can train a 1000 layer ResNet using less than 7GB of GPU memory.

3.4.3 LSTM for Long Sequences

We also evaluate the algorithms on a LSTM under a long sequence unrolling setting. We unrolled a four layer LSTM with 1024 hidden states equals 64 over time. The batch size is set to 64. The input of each timestamp is a continuous 50 dimension vector and the output is softmax over 5000 class. This is a typical setting for speech recognition[82], but our result can also be generalized to

other recurrent networks. Using a long unrolling step can potentially help recurrent model to learn long term dependencies over time. We show the results in Fig. 3.7. We can find that inplace helps a lot here. This is because inplace optimization in our experiment enables direct addition of weight gradient to a single memory cell, preventing allocate space for gradient at each timestamp. The sub-linear plan gives more than 4x reduction over the optimized memory plan.

3.4.4 Impact on Training Speed

We also measure the runtime cost of each strategy. The speed is benchmarked on a single Titan X GPU. The results are shown in Fig. 3.9. Because of the double forward cost in gradient calculation, the sublinear allocation strategy costs 30% additional runtime compared to the normal strategy. By paying the small price, we are now able to train a much wider range of deep learning models.

Chapter 4

TVM: END-TO-END OPTIMIZING COMPILER FOR DEEP LEARNING

Deep learning models can now recognize images, process natural language, and defeat humans in challenging strategy games. There is a growing demand to deploy smart applications to a wide spectrum of devices, ranging from cloud servers to self-driving cars and embedded devices. Mapping deep learning workloads to these devices is complicated by the diversity of hardware characteristics, including embedded CPUs, GPUs, FPGAs, and ASICs (e.g., the TPU [51]). These hardware targets diverge in terms of memory organization, compute functional units, etc., as shown in Figure 4.1.

Current deep learning frameworks, such as TensorFlow, MXNet, Caffe, and PyTorch, rely on a computational graph intermediate representation to implement optimizations, e.g., auto differentiation and dynamic memory management [3, 19, 4]. Graph-level optimizations, however, are often too high-level to handle hardware back-end-specific operator-level transformations. Most of these frameworks focus on a narrow class of server-class GPU devices and delegate target-specific optimizations to highly engineered and vendor-specific operator libraries. These operator-level libraries require significant manual tuning and hence are too specialized and opaque to be easily ported across hardware devices. Providing support in various deep learning frameworks for diverse hardware back-ends presently requires significant engineering effort. Even for supported back-ends, frameworks must make the difficult choice between: (1) avoiding graph optimizations that yield new operators not in the predefined operator library, and (2) using unoptimized implementations of these new operators.

To enable both graph- and operator-level optimizations for diverse hardware back-ends, we take a fundamentally different, end-to-end approach. In this chapter, we introduce TVM, *a compiler that takes a high-level specification of a deep learning program from existing frameworks and generates*

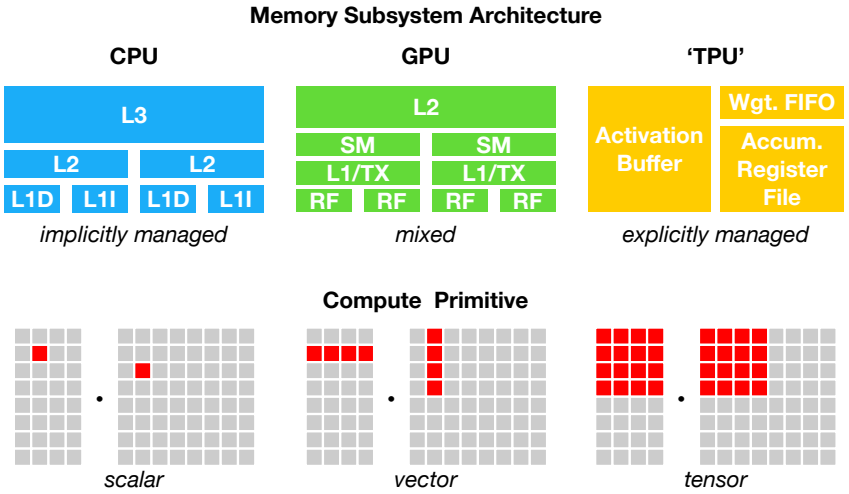


Figure 4.1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

low-level optimized code for a diverse set of hardware back-ends. To be attractive to users, TVM needs to offer performance competitive with the multitude of manually optimized operator libraries across diverse hardware back-ends. This goal requires addressing the key challenges described below.

Leveraging Specific Hardware Features and Abstractions. DL accelerators introduce optimized tensor compute primitives [51, 1, 25], while GPUs and CPUs continuously improve their processing elements. This poses a significant challenge in generating optimized code for a given operator description. The inputs to hardware instructions are multi-dimensional, with fixed or variable lengths; they dictate different data layouts; and they have special requirements for memory hierarchy. The system must effectively exploit these complex primitives to benefit from acceleration. Further, accelerator designs also commonly favor leaner control [51] and offload most scheduling complexity to the compiler stack. For specialized accelerators, the system now needs to generate code that explicitly controls pipeline dependencies to hide memory access latency – a job that hardware performs for CPUs and GPUs.

Large Search Space for Optimization Another challenge is producing efficient code without manually tuning operators. The combinatorial choices of memory access, threading pattern, and novel hardware primitives creates a huge configuration space for generated code (e.g., loop tiles and ordering, caching, unrolling) that would incur a large search cost if we implement black box auto-tuning. One could adopt a predefined cost model to guide the search, but building an accurate cost model is difficult due to the increasing complexity of modern hardware. Furthermore, such an approach would require us to build separate cost models for each hardware type.

TVM addresses these challenges with three key modules. (**1**) We introduce a *tensor expression language* to build operators and provide program transformation primitives that generate different versions of the program with various optimizations. This layer extends Halide [76]'s compute/schedule separation concept by also separating target hardware intrinsics from transformation primitives, which enables support for novel accelerators and their corresponding new intrinsics. Moreover, we introduce new transformation primitives to address GPU-related challenges and enable deployment to specialized accelerators. We can then apply different sequences of program transformations to form a rich space of valid programs for a given operator declaration. (**2**) We introduce an *automated program optimization framework* to find optimized tensor operators. The optimizer is guided by an ML-based cost model that adapts and improves as we collect more data from a hardware back-end. (**3**) On top of the automatic code generator, we introduce a *graph rewriter* that takes full advantage of high- and operator-level optimizations.

By combining these three modules, TVM can take model descriptions from existing deep learning frameworks, perform joint high- and low-level optimizations, and generate hardware-specific optimized code for back-ends, e.g., CPUs, GPUs, and FPGA-based specialized accelerators.

4.1 System Overview

This section describes TVM by using an example to walk through its components. Figure 4.2 summarizes the components TVM stack. The system first takes as input a model from an existing framework and transforms it into a high-level computational graph IR. It then performs high-level dataflow rewriting to generate an optimized graph. Operators are specified in a declarative tensor

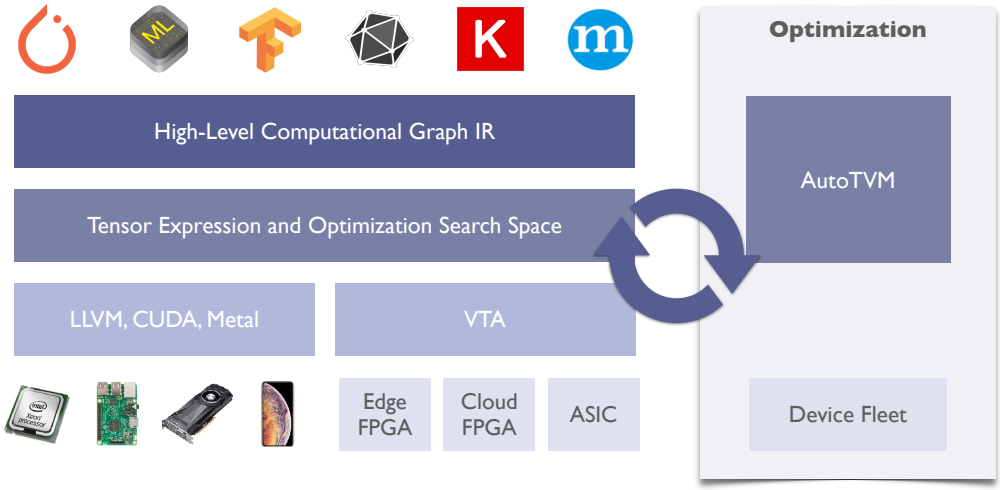


Figure 4.2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

expression language; execution details are unspecified. TVM identifies a collection of possible code optimizations for a given hardware target's operators. Then we search over a space of possible optimizations to generate efficient code for each fused operator in this graph. AutoTVM is our ML-based program optimizer which we will discuss in the next chapter. Finally, the system packs the generated code into a deployable module.

End-User Example. In a few lines of code, a user can take a model from existing deep learning frameworks and call the TVM API to get a deployable module:

```
import tvm as t
# Use keras framework as example, import model
graph, params = t.frontend.from_keras(keras_model)
target = t.target.cuda()
graph, lib, params = t.compiler.build(graph, target, params)
```

This compiled runtime module contains three components: the final optimized computational graph (`graph`), generated operators (`lib`), and module parameters (`params`). These components can then be used to deploy the model to the target back-end:

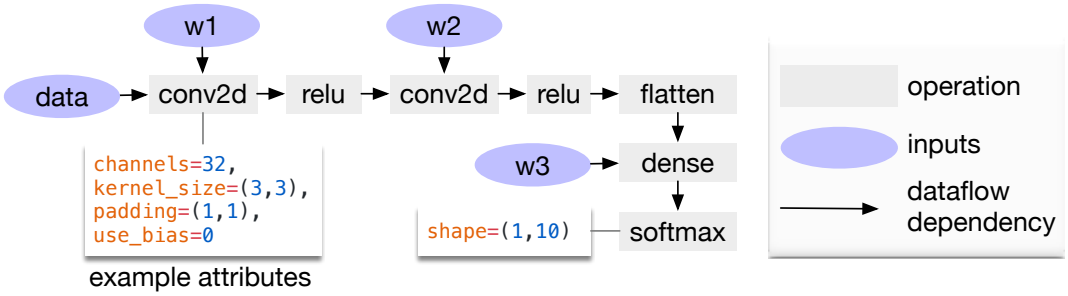


Figure 4.3: Example computational graph of a two-layer convolutional neural network. Each node in the graph represents an operation that consumes one or more tensors and produces one or more tensors. Tensor operations can be parameterized by attributes to configure their behavior (e.g., padding or strides).

```
import tvm.runtime as t
module = runtime.create(graph, lib, t.cuda(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=t.cuda(0))
module.get_output(0, output)
```

TVM supports multiple deployment back-ends in languages such as C++, Java and Python. The rest of this chapter describes TVM's architecture and how a system programmer can extend it to support new back-ends.

4.2 Optimizing Computational Graphs

Computational graphs are a common way to represent programs in DL frameworks [3, 7, 19, 4]. Figure 4.3 shows an example computational graph representation of a two-layer convolutional neural network. The main difference between this high-level representation and a low-level compiler intermediate representation (IR), such as LLVM, is that the intermediate data items are large, multi-dimensional tensors. Computational graphs provide a global view of operators, but they avoid specifying how each operator must be implemented. Like LLVM IRs, a computational graph can be transformed into functionally equivalent graphs to apply optimizations. We also take advantage of shape specificity in common DL workloads to optimize for a fixed set of input shapes.

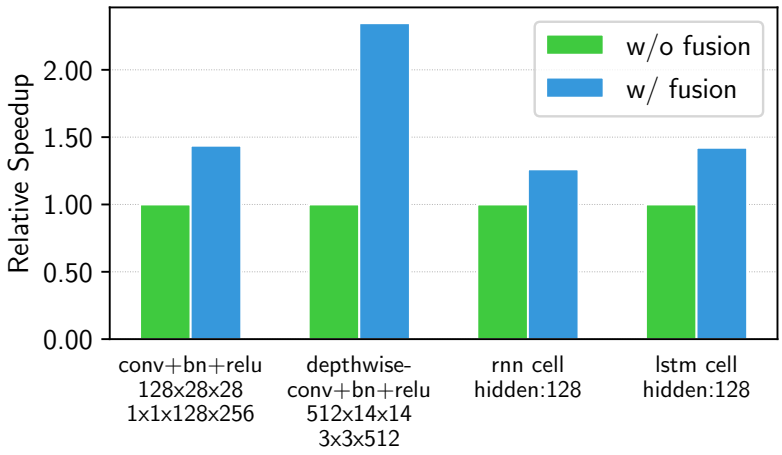


Figure 4.4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.

TVM exploits a computational graph representation to apply high-level optimizations: a node represents an operation on tensors or program inputs, and edges represent data dependencies between operations. It implements many graph-level optimizations, including: *operator fusion*, which fuses multiple small operations together; *constant-folding*, which pre-computes graph parts that can be determined statically, saving execution costs; a *static memory planning pass*, which pre-allocates memory to hold each intermediate tensor; and *data layout transformations*, which transform internal data layouts into back-end-friendly forms. We now discuss operator fusion and the data layout transformation.

Operator Fusion. Operator fusion combines multiple operators into a single kernel without saving the intermediate results in memory. This optimization can greatly reduce execution time, particularly in GPUs and specialized accelerators. Specifically, we recognize four categories of graph operators: (1) injective (one-to-one map, e.g., add), (2) reduction (e.g., sum), (3) complex-out-fusable (can fuse element-wise map to output, e.g., conv2d), and (4) opaque (cannot be fused, e.g., sort). We provide generic rules to fuse these operators, as follows. Multiple injective operators can

be fused into another injective operator. A reduction operator can be fused with input injective operators (e.g., fuse scale and sum). Operators such as conv2d are complex-out-fusable, and we can fuse element-wise operators to its output. We can apply these rules to transform the computational graph into a fused version. Figure 4.4 demonstrates the impact of this optimization on different workloads. We find that fused operators generate up to a $1.2\times$ to $2\times$ speedup by reducing memory accesses.

Data Layout Transformation. There are multiple ways to store a given tensor in the computational graph. The most common data layout choices are column major and row major. In practice, we may prefer to use even more complicated data layouts. For instance, a DL accelerator might exploit 4×4 matrix operations, requiring data to be tiled into 4×4 chunks to optimize for access locality.

Data layout optimization converts a computational graph into one that can use better internal data layouts for execution on the target hardware. It starts by specifying the preferred data layout for each operator given the constraints dictated by memory hierarchies. We then perform the proper layout transformation between a producer and a consumer if their preferred data layouts do not match.

While high-level graph optimizations can greatly improve the efficiency of DL workloads, they are only as effective as what the operator library provides. Currently, the few DL frameworks that support operator fusion require the operator library to provide an implementation of the fused patterns. With more network operators introduced on a regular basis, the number of possible fused kernels can grow dramatically. This approach is no longer sustainable when targeting an increasing number of hardware back-ends since the required number of fused pattern implementations grows combinatorially with the number of data layouts, data types, and accelerator intrinsics that must be supported. It is not feasible to handcraft operator kernels for the various operations desired by a program and for each back-end. To this end, we next propose a code generation approach that can generate various possible implementations for a given model's operators.

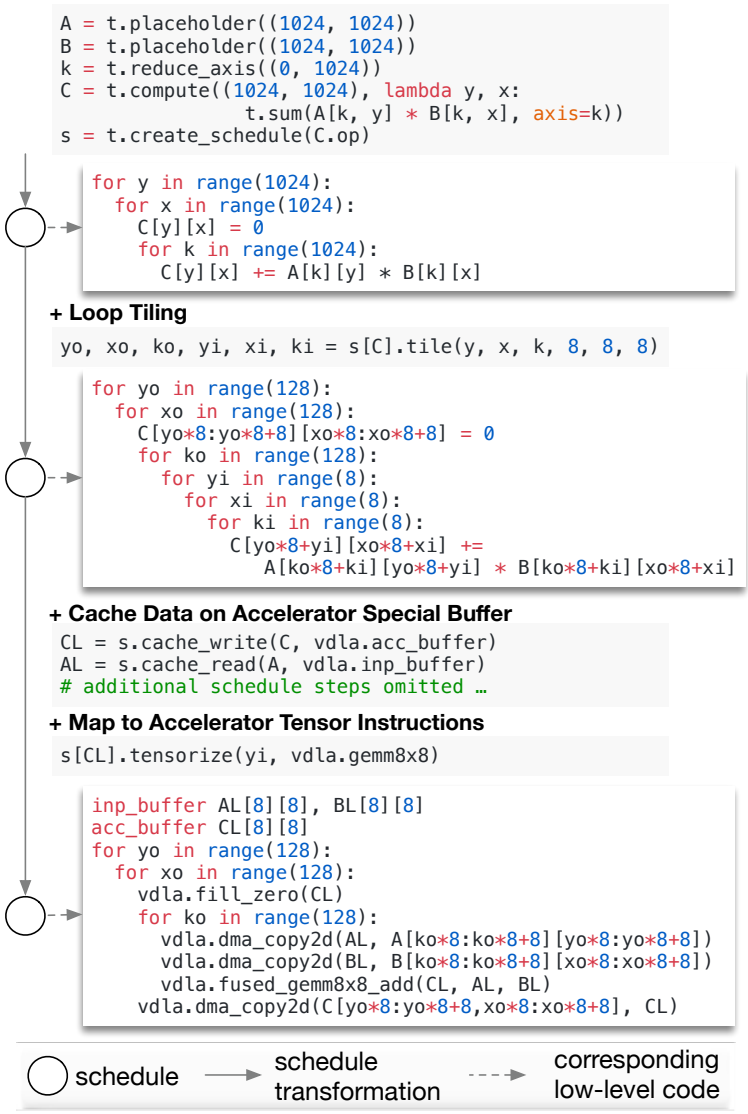
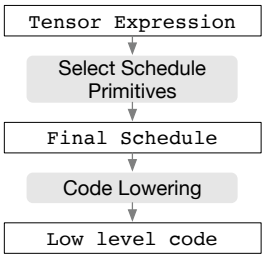


Figure 4.5: Example schedule transformations that optimize a matrix multiplication on a specialized accelerator.

4.3 Generating Tensor Operations

TVM produces efficient code for each operator by generating many valid implementations on each hardware back-end and choosing an optimized implementation. This process builds on Halide's idea of decoupling descriptions from computation rules (or *schedule optimizations*) [76] and ex-



Schedule primitives used in various hardware backends	CPU Schedule	GPU Schedule	Accel. Schedule
[Halide] Loop Transformations	✓	✓	✓
[Halide] Thread Binding	✓	✓	✓
[Halide] Compute Locality	✓	✓	✓
[TVM] Special Memory Scope		✓	✓
[TVM] Tensorization	✓	✓	✓
[TVM] Latency Hiding			✓

Figure 4.6: TVM schedule lowering and code generation process. The table lists existing Halide and novel TVM scheduling primitives being used to optimize schedules for CPUs, GPUs and accelerator back-ends. Tensorization is essential for accelerators, but it can also be used for CPUs and GPUs. Special memory-scope enables memory reuse in GPUs and explicit management of on-chip memory in accelerators. Latency hiding is specific to TPU-like accelerators.

tends it to support new optimizations (nested parallelism, tensorization, and latency hiding) and a wide array of hardware back-ends. We now highlight TVM-specific features.

4.3.1 Tensor Expression and Schedule Space

We introduce a tensor expression language to support automatic code generation. Unlike high-level computation graph representations, where the implementation of tensor operations is opaque, each operation is described in an index formula expression language. The following code shows an example tensor expression to compute transposed matrix multiplication:

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
              t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

Each compute operation specifies both the shape of the output tensor and an expression describing how to compute each element of it. Our tensor expression language supports common arithmetic and math operations and covers common DL operator patterns. The language does not specify the loop structure and many other execution details, and it provides flexibility for adding

hardware-aware optimizations for various back-ends. Adopting the decoupled compute/schedule principle from Halide [76], we use a schedule to denote a specific mapping from a tensor expression to low-level code. Many possible schedules can perform this function.

We build a schedule by incrementally applying basic transformations (schedule primitives) that preserve the program's logical equivalence. Figure 4.5 shows an example of scheduling matrix multiplication on a specialized accelerator. Internally, TVM uses a data structure to keep track of the loop structure and other information as we apply schedule transformations. This information can then help generate low-level code for a given final schedule.

Our tensor expression takes cues from Halide [76], Darkroom [44], and TACO [54]. Its primary enhancements include support for the new schedule optimizations discussed below. To achieve high performance on many back-ends, we must support enough schedule primitives to cover a diverse set of optimizations on different hardware back-ends. Figure 4.6 summarizes the operation code generation process and schedule primitives that TVM supports. We reuse helpful primitives and the low-level loop program AST from Halide, and we introduce new primitives to optimize GPU and accelerator performance. The new primitives are necessary to achieve optimal GPU performance and essential for accelerators. CPU, GPU, TPU-like accelerators are three important types of hardware for deep learning. This section describes new optimization primitives for CPUs, GPUs and TPU-like accelerators.

4.3.2 Nested Parallelism with Cooperation

Parallelism is key to improving the efficiency of compute-intensive kernels in DL workloads. Modern GPUs offer massive parallelism, requiring us to bake parallel patterns into schedule transformations. Most existing solutions adopt a model called *nested parallelism*, a form of fork–join. This model requires a parallel schedule primitive to parallelize a data parallel task; each task can be further recursively subdivided into subtasks to exploit the target architecture's multi-level thread hierarchy (e.g., thread groups in GPU). We call this model *shared-nothing nested parallelism* because one working thread cannot look at the data of its sibling within the same parallel stage.

An alternative to the shared-nothing approach is to fetch data cooperatively. Specifically,

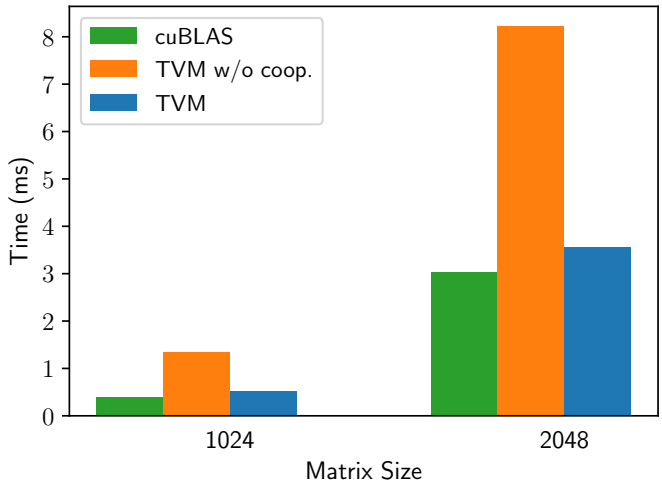


Figure 4.7: Performance comparison between TVM with and without cooperative shared memory fetching on matrix multiplication workloads. Tested on an NVIDIA Titan X. By cooperative shared memory fetching. TVM can get close to 80% of the peak GPU utilization.

groups of threads can cooperatively fetch the data they all need and place it into a shared memory space. [1] This optimization can take advantage of the GPU memory hierarchy and enable data reuse across threads through shared memory regions. TVM supports this well-known GPU optimization using a schedule primitive to achieve optimal performance. The following GPU code example optimizes matrix multiplication.

```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_barrier_among_threads()
      for yi in range(8):
        for xi in range(8):
          CL[yi][xi] += AS[yi] * BS[xi]
      for yi in range(8):
        for xi in range(8):
          C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

All threads cooperatively load AS and BS in different parallel patterns

Barrier inserted automatically by compiler

[1] Halide recently added shared memory support but without general memory scope for accelerators.

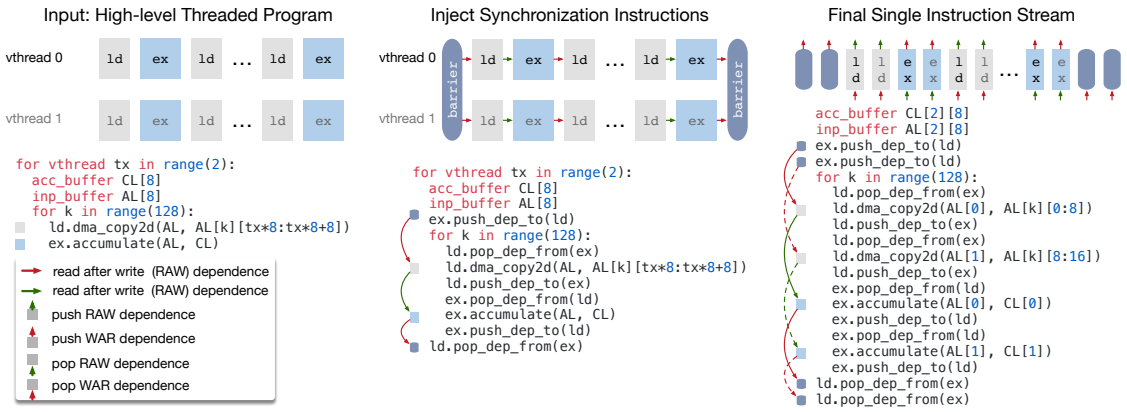


Figure 4.8: TVM virtual thread lowering transforms a virtual thread-parallel program to a single instruction stream; the stream contains explicit low-level synchronizations that the hardware can interpret to recover the pipeline parallelism required to hide memory access latency.

Figure 4.7 demonstrates the impact of this optimization. We introduce the concept of *memory scopes* to the schedule space so that a compute stage (AS and BS in the code) can be marked as shared. Without explicit memory scopes, automatic scope inference will mark compute stages as thread-local. The shared task must compute the dependencies of all working threads in the group. Additionally, memory synchronization barriers must be properly inserted to guarantee that shared loaded data is visible to consumers. Finally, in addition to being useful to GPUs, memory scopes let us tag special memory buffers and create special lowering rules when targeting specialized DL accelerators.

4.3.3 Tensorization

DL workloads have high arithmetic intensity, which can typically be decomposed into tensor operators like matrix-matrix multiplication or 1D convolution. These natural decompositions have led to the recent trend of adding tensor compute primitives [51, 1, 25]. These new primitives create both opportunities and challenges for schedule-based compilation; while using them can improve performance, the compilation framework must seamlessly integrate them. We dub this *tensorization*:

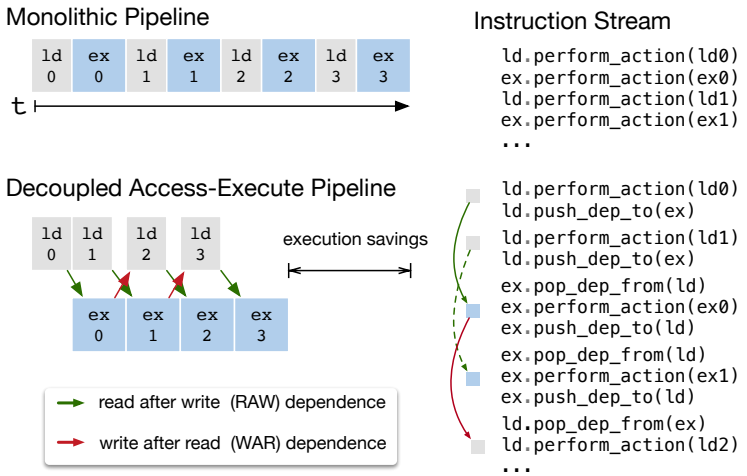


Figure 4.9: Decoupled Access-Execute in hardware hides most memory access latency by allowing memory and computation to overlap. Execution correctness is enforced by low-level synchronization in the form of dependence token enqueueing/dequeuing actions, which the compiler stack must insert in the instruction stream.

it is analogous to vectorization for SIMD architectures but has significant differences. Instruction inputs are multi-dimensional, with fixed or variable lengths, and each has different data layouts. More importantly, we cannot support a fixed set of primitives since new accelerators are emerging with their own variations of tensor instructions. We therefore need an *extensible* solution.

We make tensorization extensible by separating the target hardware intrinsic from the schedule with a mechanism for tensor-intrinsic declaration. We use the same tensor expression language to declare both the behavior of each new hardware intrinsic and the lowering rule associated with it. The following code shows how to declare an 8×8 tensor hardware intrinsic.

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
   ww_ptr = inputs[0].access_ptr("r")
   xx_ptr = inputs[1].access_ptr("r")
   zz_ptr = outputs[0].access_ptr("w")
   compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
   reset = t.hardware_intrin("fill_zero", zz_ptr)
   update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
   return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

Additionally, we introduce a *tensorize* schedule primitive to replace a unit of computation with the corresponding intrinsics. The compiler matches the computation pattern with a hardware declaration and lowers it to the corresponding hardware intrinsic.

Tensorization decouples the schedule from specific hardware primitives, making it easy to extend TVM to support new hardware architectures. The generated code of tensorized schedules aligns with practices in high-performance computing: break complex operations into a sequence of micro-kernel calls. We can also use the *tensorize* primitive to take advantage of handcrafted micro-kernels, which can be beneficial in some platforms. For example, we implement ultra low precision operators for mobile CPUs that operate on data types that are one- or two-bits wide by leveraging a bit-serial matrix vector multiplication micro-kernel. This micro-kernel accumulates results into progressively larger data types to minimize the memory footprint. Presenting the micro-kernel as a tensor intrinsic to TVM yields up to a 1.5× speedup over the non-tensorized version.

4.3.4 Explicit Memory Latency Hiding

Latency hiding refers to the process of overlapping memory operations with computation to maximize utilization of memory and compute resources. It requires different strategies depending on the target hardware back-end. On CPUs, memory latency hiding is achieved implicitly with simultaneous multithreading [31] or hardware prefetching [50, 24]. GPUs rely on rapid context switching of many warps of threads [102]. In contrast, specialized DL accelerators such as the TPU [51] usually favor leaner control with a *decoupled access-execute* (DAE) architecture [86] and offload the problem of fine-grained synchronization to software.

Figure 4.9 shows a DAE hardware pipeline that reduces runtime latency. Compared to a monolithic hardware design, the pipeline can hide most memory access overheads and almost fully utilize compute resources. To achieve higher utilization, the instruction stream must be augmented with fine-grained synchronization operations. Without them, dependencies cannot be enforced, leading to erroneous execution. Consequently, DAE hardware pipelines require fine-grained dependence enqueuing/dequeuing operations between the pipeline stages to guarantee correct execu-

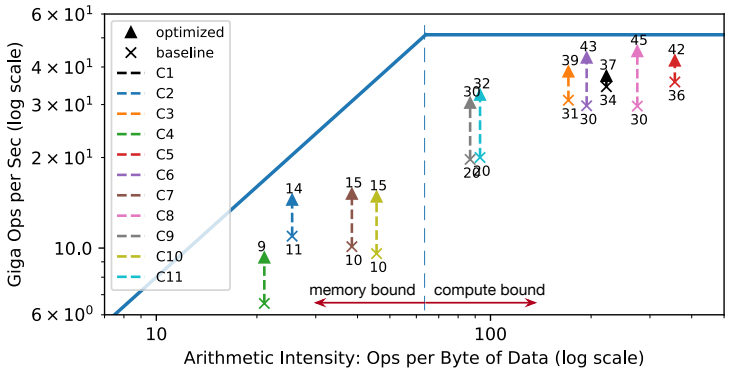


Figure 4.10: Roofline [105] of an FPGA-based DL accelerator running ResNet inference. With latency hiding enabled by TVM, performance of the benchmarks is brought closer to the roofline, demonstrating higher compute and memory bandwidth efficiency.

tion, as shown in Figure 4.9's instruction stream.

Programming DAE accelerators that require explicit low-level synchronization is difficult. To reduce the programming burden, we introduce a virtual threading scheduling primitive that lets programmers specify a high-level data parallel program as they would a hardware back-end with support for multithreading. TVM then automatically lowers the program to a single instruction stream with low-level explicit synchronization, as shown in Figure 4.8. The algorithm starts with a high-level multi-threaded program schedule and then inserts the necessary low-level synchronization operations to guarantee correct execution within each thread. Next, it interleaves operations of all virtual threads into a single instruction stream. Finally, the hardware recovers the available pipeline parallelism dictated by the low-level synchronizations in the instruction stream.

Hardware Evaluation of Latency Hiding. We now demonstrate the effectiveness of latency hiding on a custom FPGA-based accelerator design, which we describe in depth in subsection 4.6.4. We ran each layer of ResNet on the accelerator and used TVM to generate two schedules: one with latency hiding, and one without. The schedule with latency hiding parallelized the program with virtuals threads to expose pipeline parallelism and therefore hide memory access latency.

Results are shown in Figure 4.10 as a roofline diagram [105]; roofline performance diagrams provide insight into how well a given system uses computation and memory resources for different benchmarks. Overall, latency hiding improved performance on all ResNet layers. Peak compute utilization increased from 70% with no latency hiding to 88% with latency hiding.

4.4 Automating Optimization

Given the rich set of schedule primitives, our remaining problem is to find optimal operator implementations for each layer of a DL model. Here, TVM creates a specialized operator for the specific input shape and layout associated with each layer. Such specialization offers significant performance benefits (in contrast to handcrafted code that would target a smaller diversity of shapes and layouts), but it also raises automation challenges. The system needs to choose the schedule optimizations – such as modifying the loop order or optimizing for the memory hierarchy – as well as schedule-specific parameters, such as the tiling size and the loop unrolling factor. Such combinatorial choices create a large search space of operator implementations for each hardware back-end. To address this challenge, we built an *automated schedule optimizer* with two main components: a schedule explorer that *proposes* promising new configurations, and a machine learning cost model that *predicts* the performance of a given configuration. This section describes these components and TVM's automated optimization flow (Figure 4.11).

4.4.1 Schedule Space Specification

We built a *schedule template specification API* to let a developer declare knobs in the schedule space. The template specification allows incorporation of a developer's domain-specific knowledge, as necessary, when specifying possible schedules. We also created a *generic master template for each hardware back-end* that automatically extracts possible knobs based on the computation description expressed using the tensor expression language. At a high level, we would like to consider as many configurations as possible and let the optimizer manage the selection burden. Consequently, the optimizer must search over *billions* of possible configurations for the real world

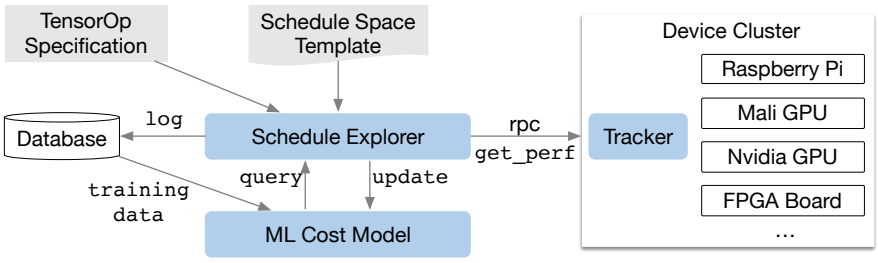


Figure 4.11: Overview of automated optimization framework. A schedule explorer examines the schedule space using an ML-based cost model and chooses experiments to run on a distributed device cluster via RPC. To improve its predictive power, the ML model is updated periodically using collected data recorded in a database.

DL workloads used in our experiments.

4.4.2 ML-Based Cost Model

One way to find the best schedule from a large configuration space is through blackbox optimization, i.e., auto-tuning. This method is used to tune high performance computing libraries[104, 37]. However, auto-tuning requires many experiments to identify a good configuration.

An alternate approach is to build a predefined cost model to guide the search for a particular hardware back-end instead of running all possibilities and measuring their performance. Ideally, a perfect cost model considers all factors affecting performance: memory access patterns, data reuse, pipeline dependencies, and threading patterns, among others. This approach, unfortunately, is burdensome due to the increasing complexity of modern hardware. Furthermore, every new hardware target requires a new (predefined) cost model.

We instead take a statistical approach to solve the cost modeling problem. In this approach, a schedule explorer proposes configurations that may improve an operator's performance. For each schedule configuration, we use an ML model that takes the lowered loop program as input and predicts its running time on a given hardware back-end. The model, trained using runtime mea-

Method Category	Data Cost	Model Bias	Need Hardware Info	Learn from History
Blackbox auto-tuning	high	none	no	no
Predefined cost model	none	high	yes	no
ML based cost model	**low**	**low**	**no**	**yes**

Table 4.1: Comparison of automation methods. Model bias refers to inaccuracy due to modeling.

surement data collected during exploration, does not require the user to input detailed hardware information. We update the model periodically as we explore more configurations during optimization, which improves accuracy for other related workloads, as well. In this way, the quality of the ML model improves with more experimental trials. Table 4.1 summarizes the key differences between automation methods. ML-based cost models strike a balance between auto-tuning and predefined cost modeling and can benefit from the historical performance data of related workloads. We will describe the design details of the ML-based program optimizer in Chapter. 5.

4.4.3 Distributed Device Pool and RPC

A *distributed device pool* scales up the running of on-hardware trials and enables fine-grained resource sharing among multiple optimization jobs. TVM implements a customized, RPC-based distributed device pool that enables clients to run programs on a specific type of device. We can use this interface to compile a program on the host compiler, request a remote device, run the function remotely, and access results in the same script on the host. TVM's RPC supports dynamic upload and runs cross-compiled modules and functions that use its runtime convention. As a result, the same infrastructure can perform a single workload optimization and end-to-end graph inference. Our approach automates the compile, run, and profile steps across multiple devices. This infrastructure is especially critical for embedded devices, which traditionally require tedious manual effort for cross-compilation, code deployment, and measurement.

4.5 Related Works

Deep learning frameworks [3, 7, 19, 4] provide convenient interfaces for users to express DL workloads and deploy them easily on different hardware back-ends. While existing frameworks currently depend on vendor-specific tensor operator libraries to execute their workloads, they can leverage TVM's stack to generate optimized code for a larger number of hardware devices.

High-level computation graph DSLs are a typical way to represent and perform high-level optimizations. Tensorflow's XLA [3] and the recently introduced DLVM [103] fall into this category. The representations of computation graphs in these works are similar, and a high-level computation graph DSL is also used in this chapter. While graph-level representations are a good fit for high-level optimizations, they are too high level to optimize tensor operators under a diverse set of hardware back-ends. Prior work relies on specific lowering rules to directly generate low-level LLVM or resorts to vendor-crafted libraries. These approaches require significant engineering effort for each hardware back-end and operator-variant combination.

Halide [76] introduced the idea of separating computing and scheduling. We adopt Halide's insights and reuse its existing useful scheduling primitives in our compiler. Our tensor operator scheduling is also related to other work on DSL for GPUs [91, 45, 92, 55] and polyhedral-based loop transformation [6, 101]. TACO [54] introduces a generic way to generate sparse tensor operators on CPU. Weld [71] is a DSL for data processing tasks. We specifically focus on solving the new scheduling challenges of DL workloads for GPUs and specialized accelerators. Our new primitives can potentially be adopted by the optimization pipelines in these works.

High-performance libraries such as ATLAS [104] and FFTW [37] use auto-tuning to get the best performance. Tensor comprehension [99] applied black-box auto-tuning together with polyhedral optimizations to optimize CUDA kernels. OpenTuner [5] and existing hyper parameter-tuning algorithms [60] apply domain-agnostic search. A predefined cost model is used to automatically schedule image processing pipelines in Halide [69]. TVM's ML model uses effective domain-aware cost modeling that considers program structure. The based distributed schedule optimizer scales to a larger search space and can find state-of-the-art kernels on a large range of supported

Name	Operator	H, W	IC, OC	K, S
C1	conv2d	224, 224	3,64	7, 2
C2	conv2d	56, 56	64,64	3, 1
C3	conv2d	56, 56	64,64	1, 1
C4	conv2d	56, 56	64,128	3, 2
C5	conv2d	56, 56	64,128	1, 2
C6	conv2d	28, 28	128,128	3, 1
C7	conv2d	28, 28	128,256	3, 2
C8	conv2d	28, 28	128,256	1, 2
C9	conv2d	14, 14	256,256	3, 1
C10	conv2d	14, 14	256,512	3, 2
C11	conv2d	14, 14	256,512	1, 2
C12	conv2d	7, 7	512,512	3, 1

Name	Operator	H, W	IC	K, S
D1	depthwise conv2d	112, 112	32	3, 1
D2	depthwise conv2d	112, 112	64	3, 2
D3	depthwise conv2d	56, 56	128	3, 1
D4	depthwise conv2d	56, 56	128	3, 2
D5	depthwise conv2d	28, 28	256	3, 1
D6	depthwise conv2d	28, 28	256	3, 2
D7	depthwise conv2d	14, 14	512	3, 1
D8	depthwise conv2d	14, 14	512	3, 2
D9	depthwise conv2d	7, 7	1024	3, 1

Table 4.2: Configurations of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet used in the single kernel experiments. H/W denotes height and width, IC input channels, OC output channels, K kernel size, and S stride size. All ops use "SAME" padding. All depthwise conv2d operations have channel multipliers of 1.

back-ends. More importantly, we provide an end-to-end stack that can take descriptions directly from DL frameworks and jointly optimize together with the graph-level stack.

Despite the emerging popularity of accelerators for deep learning [51, 26], it remains unclear how a compilation stack can be built to effectively target these devices. The VTA design used in our evaluation provides a generic way to summarize the properties of TPU-like accelerators and enables a concrete case study on how to compile code for accelerators. Our approach could potentially benefit existing systems that compile deep learning to FPGA [84, 97], as well. this chapter provides a generic solution to effectively target accelerators via tensorization and compiler-driven latency hiding.

4.6 Evaluations

TVM's core is implemented in C++ (∼50k LoC). We provide language bindings to Python and Java. Earlier sections of this chapter evaluated the impact of several individual optimizations and components of TVM, namely, *operator fusion* in Figure 4.4, *latency hiding* in Figure 4.10.

We now focus on an end-to-end evaluation that aims to answer the following questions:

- Can TVM optimize DL workloads over multiple platforms?
- How does TVM compare to existing DL frameworks (which rely on heavily optimized libraries) on each back-end?
- Can TVM support new, emerging DL workloads (e.g., depthwise convolution, low precision operations)?
- Can TVM support and optimize for new specialized accelerators?

To answer these questions, we evaluated TVM on four types of platforms: (1) a server-class GPU, (2) an embedded GPU, (3) an embedded CPU, and (4) a DL accelerator implemented on a low-power FPGA SoC. The benchmarks are based on real world DL inference workloads, including ResNet [42], MobileNet [47], the LSTM Language Model [107], the Deep Q Network (DQN) [68] and Deep Convolutional Generative Adversarial Networks (DCGAN) [75]. We compare our approach to existing DL frameworks, including MxNet [19] and TensorFlow [2], that rely on highly engineered, vendor-specific libraries. TVM performs end-to-end automatic optimization and code generation *without the need for an external operator library*.

4.6.1 Server-Class GPU Evaluation

We first compared the end-to-end performance of deep neural networks TVM, MXNet (v1.1), Tensorflow (v1.7), and Tensorflow XLA on an Nvidia Titan X. MXNet and Tensorflow both use cuDNN v7 for convolution operators; they implement their own versions of depthwise convolution

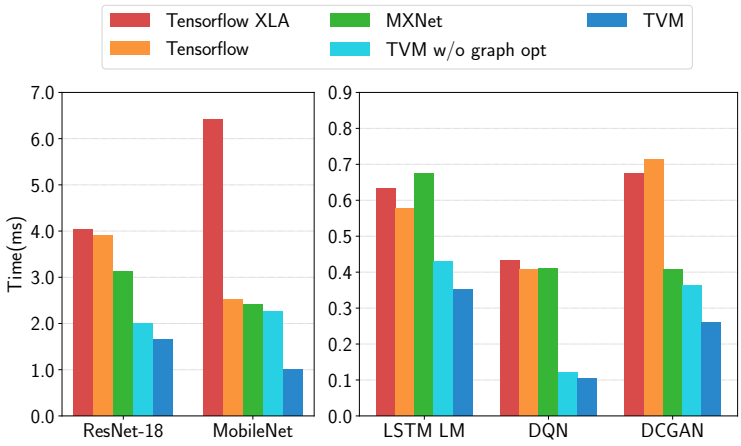


Figure 4.12: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

since it is relatively new and not yet supported by the latest libraries. They also use cuBLAS v8 for matrix multiplications. On the other hand, Tensorflow XLA uses JIT compilation.

Figure 4.12 shows that TVM outperforms the baselines, with speedups ranging from 1.6× to 3.8× due to both joint graph optimization and the automatic optimizer, which generates high-performance fused operators. DQN's 3.8 x speedup results from its use of unconventional operators (4×4 conv2d, strides=2) that are not well optimized by cuDNN; the ResNet workloads are more conventional. TVM automatically finds optimized operators in both cases.

To evaluate the effectiveness of operator level optimization, we also perform a breakdown comparison for each tensor operator in ResNet and MobileNet, shown in Figure 4.13. We include TensorComprehension (TC, commit: ef644ba) [99], a recently introduced auto-tuning framework, as an additional baseline. [2] TC results include the best kernels it found in 10 generations × 100 population × 2 random seeds for each operator (i.e., 2000 trials per operator). 2D convolution, one of the most important DL operators, is heavily optimized by cuDNN. However, TVM can still generate better GPU kernels for most layers. Depthwise convolution is a newly introduced operator

[2] According to personal communication [98], TC is not yet meant to be used for compute-bound problems. However, it is still a good reference baseline to include in the comparison.

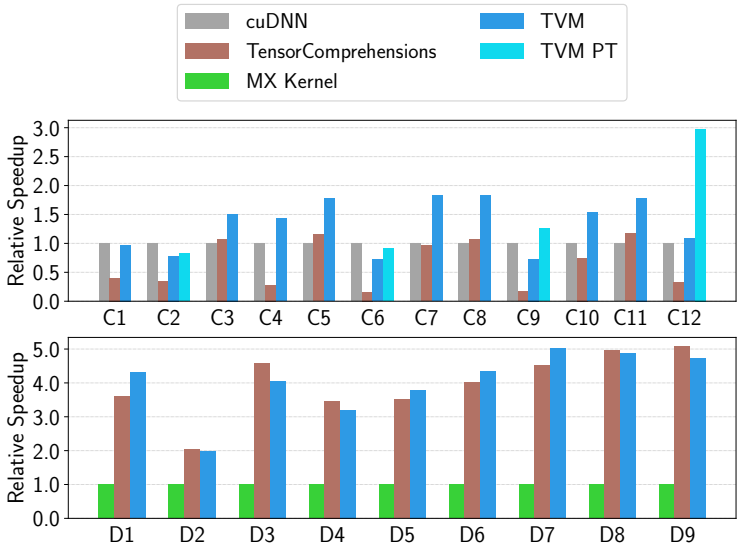


Figure 4.13: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet. Tested on a TITAN X. See Table 5.1 for operator configurations. We also include a weight pre-transformed Winograd [59] for 3x3 conv2d (TVM PT).

with a simpler structure [47]. In this case, both TVM and TC can find fast kernels compared to MXNet's handcrafted kernels. TVM's improvements are mainly due to its exploration of a large schedule space and an effective ML-based search algorithm.

4.6.2 Embedded CPU Evaluation

We evaluated the performance of TVM on an ARM Cortex A53 (Quad Core 1.2GHz). We used Tensorflow Lite (TFLite, commit: 7558b085) as our baseline system.

Figure 4.15 compares TVM operators to hand-optimized ones for ResNet and MobileNet. We observe that TVM generates operators that outperform the hand-optimized TFLite versions for both neural network workloads. This result also demonstrates TVM's ability to quickly optimize emerging tensor operators, such as depthwise convolution operators. Finally, Figure 4.14 shows

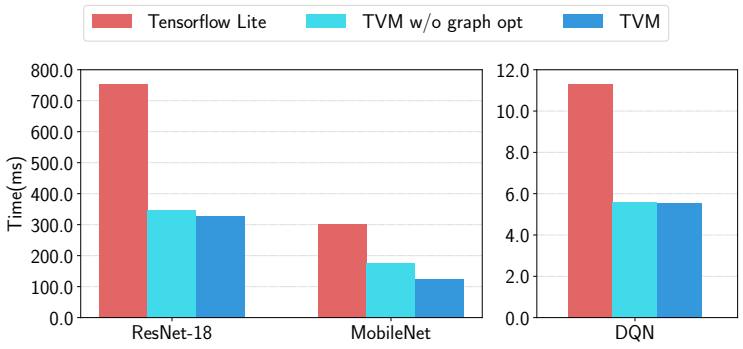


Figure 4.14: ARM A53 end-to-end evaluation of TVM and TFLite.

an end-to-end comparison of three workloads, where TVM outperforms the TFLite baseline.[3]

Ultra Low-Precision Operators We demonstrate TVM's ability to support ultra low-precision inference [28, 78] by generating highly optimized operators for fixed-point data types of less than 8-bits. Low-precision networks replace expensive multiplication with vectorized bit-serial multiplication that is composed of bitwise *and* popcount reductions [95]. Achieving efficient low-precision inference requires packing quantized data types into wider standard data types, such as `int8` or `int32`. Our system generates code that outperforms hand-optimized libraries from Caffe2 (commit: 39e07f7)[95]. We implemented an ARM-specific *tensorization* intrinsic that leverages ARM instructions to build an efficient, low-precision matrix-vector microkernel.We then used TVM's automated optimizer to explore the scheduling space.

Figure 4.16 compares TVM to the Caffe2 ultra low-precision library on ResNet for 2-bit activations, 1-bit weights inference. Since the baseline is single threaded, we also compare it to a single-threaded TVM version. Single-threaded TVM outperforms the baseline, particularly for `C5`, `C8`, and `C11` layers; these are convolution layers of kernel size 1×1 and stride of 2 for which the ultra low-precision baseline library is not optimized. Furthermore, we take advantage of additional TVM capabilities to produce a parallel library implementation that shows improvement over the

[3]DCGAN and LSTM results are not presented because they are not yet supported by the baseline.

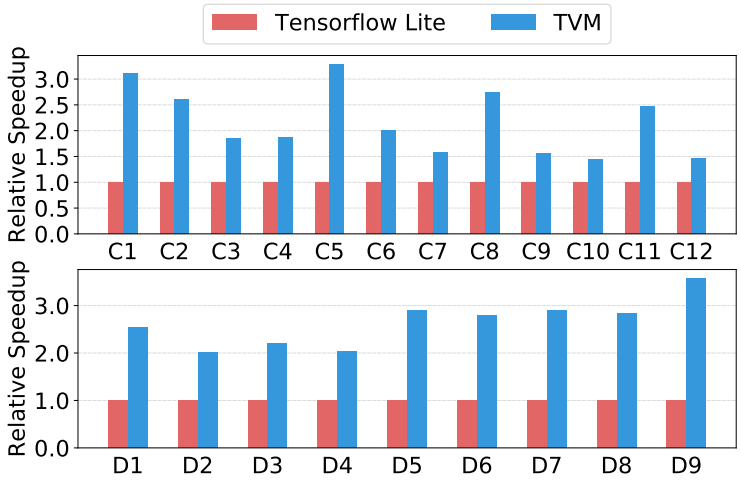


Figure 4.15: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 5.1 for the configurations of these operators.

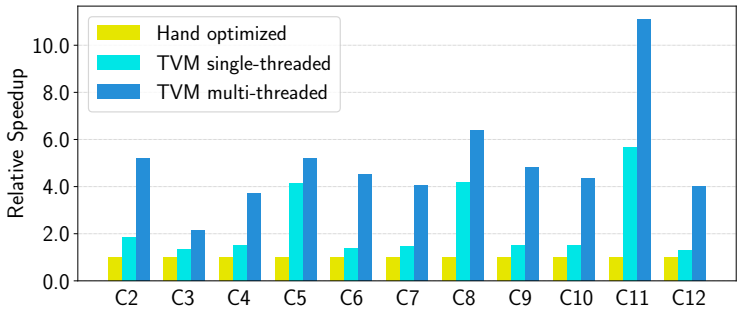


Figure 4.16: Relative speedup of single- and multi-threaded low-precision conv2d operators in ResNet. Baseline was a single-threaded, hand-optimized implementation from Caffe2 (commit: 39e07f7). C5, C3 are 1x1 convolutions that have less compute intensity, resulting in less speedup by multi-threading.

baseline. In addition to the 2-bit+1-bit configuration, TVM can generate and optimize for other precision configurations that are unsupported by the baseline library, offering improved flexibility.

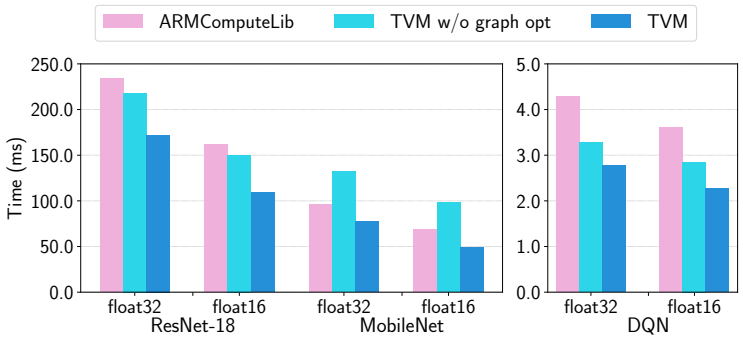


Figure 4.17: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

4.6.3 Embedded GPU Evaluation

For our mobile GPU experiments, we ran our end-to-end pipeline on a Firefly-RK3399 board equipped with an ARM Mali-T860MP4 GPU. The baseline was a vendor-provided library, the ARM Compute Library (v18.03). As shown in Figure 4.17, we outperformed the baseline on three available models for both `float16` and `float32` (DCGAN and LSTM are not yet supported by the baseline). The speedup ranged from 1.2× to 1.6×.

4.6.4 FPGA Accelerator Evaluation

Versatile Tensor Accelerator We now relate how TVM tackled accelerator-specific code generation on a generic inference accelerator design we prototyped on an FPGA. We used in this evaluation the Versatile Tensor Accelerator (VTA) – which distills characteristics from previous accelerator proposals [64, 25, 51] into a minimalist hardware architecture – to demonstrate TVM's ability to generate highly efficient schedules that can target specialized accelerators. Figure 4.18 shows the high-level hardware organization of the VTA architecture. VTA is programmed as a tensor processor to efficiently execute operations with high compute intensity (e.g, matrix multiplication, high dimensional convolution). It can perform load/store operations to bring blocked 3-dimensional tensors from DRAM into a contiguous region of SRAM. It also provides special-

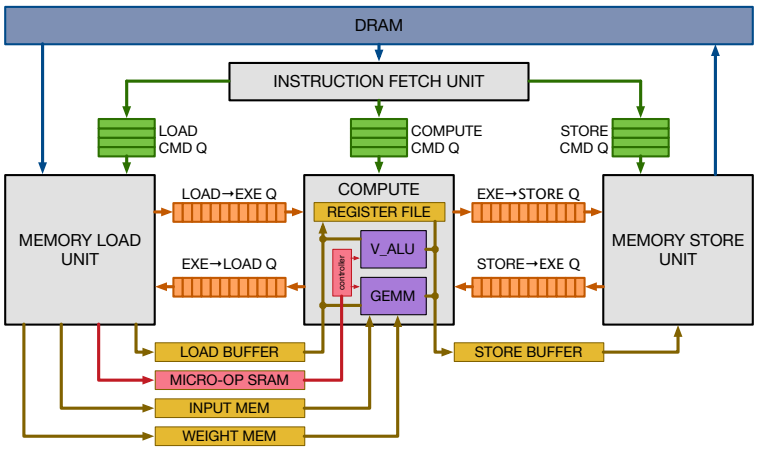


Figure 4.18: VTA Hardware design overview.

ized on-chip memories for network parameters, layer inputs (narrow data type), and layer outputs (wide data type). Finally, VTA provides explicit synchronization control over successive loads, computes, and stores to maximize the overlap between memory and compute operations.

Methodology. We implemented the VTA design on a low-power PYNQ board that incorporates an ARM Cortex A9 dual core CPU clocked at 667MHz and an Artix-7 based FPGA fabric. On these modest FPGA resources, we implemented a 16×16 matrix-vector unit clocked at 200MHz that performs products of 8-bit values and accumulates them into a 32-bit register every cycle. The theoretical peak throughput of this VTA design is about 102.4GOPS/s. We allocated 32kB of resources for activation storage, 32kB for parameter storage, 32kB for microcode buffers, and 128kB for the register file. These on-chip buffers are by no means large enough to provide sufficient on-chip storage for a single layer of ResNet and therefore enable a case study on effective memory reuse and latency hiding.

We built a driver library for VTA with a C runtime API that constructs instructions and pushes them to the target accelerator for execution. Our code generation algorithm then translates the accelerator program to a series of calls into the runtime API. Adding the specialized accelerator back-end took $\sim$2k LoC in Python.

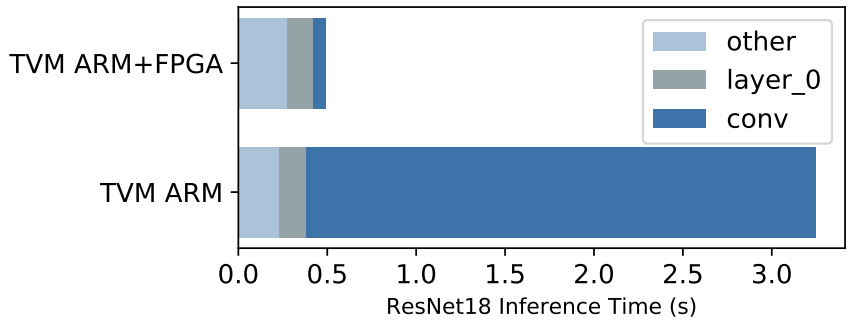


Figure 4.19: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.

End-to-End ResNet Evaluation. We used TVM to generate ResNet inference kernels on the PYNQ platform and offloaded as many layers as possible to VTA. We also used it to generate both schedules for the CPU only and CPU+FPGA implementation. Due to its shallow convolution depth, the first ResNet convolution layer could not be efficiently offloaded on the FPGA and was instead computed on the CPU. All other convolution layers in ResNet, however, were amenable to efficient offloading. Operations like residual layers and activations were also performed on the CPU since VTA does not support these operations.

Figure 4.19 breaks down ResNet inference time into CPU-only execution and CPU+FPGA execution. Most computation was spent on the convolution layers that could be offloaded to VTA. For those convolution layers, the achieved speedup was 40×. Unfortunately, due to Amdahl's law, the overall performance of the FPGA accelerated system was bottlenecked by the sections of the workload that had to be executed on the CPU. We envision that extending the VTA design to support these other operators will help reduce cost even further. This FPGA-based experiment showcases TVM's ability to adapt to new architectures and the hardware intrinsics they expose.

Chapter 5

AUTOTVM: LEARNING TO OPTIMIZE TENSOR PROGRAMS

In the previous chapter, we discussed how to build a system with a rich set of schedule primitives for tensor program optimizations. This chapter explores the following question: can we use learning to alleviate this engineering burden and automatically optimize tensor operator programs for a given hardware platform? Our affirmative answer is based on statistical cost models we built that predict program run time using a given low-level program. These cost models, which guide our exploration of the space of possible programs, use transferable representations that generalize across different workloads to accelerate search.

5.1 Problem Formalization

We begin by walking through the motivating example in Figure 5.1. To enable automatic code generation, we specify tensor operators using index expressions (e.g., $C_{ij} = \sum_k A_{ki} B_{kj}$). Let $\mathcal{E}$ denote the space of index expressions. The index expression leaves many low-level implementation details, such as loop order, memory scope, and threading unspecified. As a result, we can generate multiple variants of low-level code that are logically equivalent to the expression for a given $e \in \mathcal{E}$. We use $\mathcal{S}_e$ to denote the space of possible transformations (schedules) from e to low-level code. For an $s \in \mathcal{S}_e$, let $x = g(e, s)$ be the generated low-level code. Here, g represents a compiler framework that generates low-level code from e, s. We are interested in minimizing $f(x)$, which is the real run time cost on the hardware. Importantly, we do not know an analytical expression for $f(x)$ but can query it by running experiments on the hardware. For a given tuple of $(g, e, \mathcal{S}_e, f)$, our problem can be formalized as the following objective:

$$\arg\min_{s \in \mathcal{S}_e} f(g(e, s)) \tag{5.1}$$

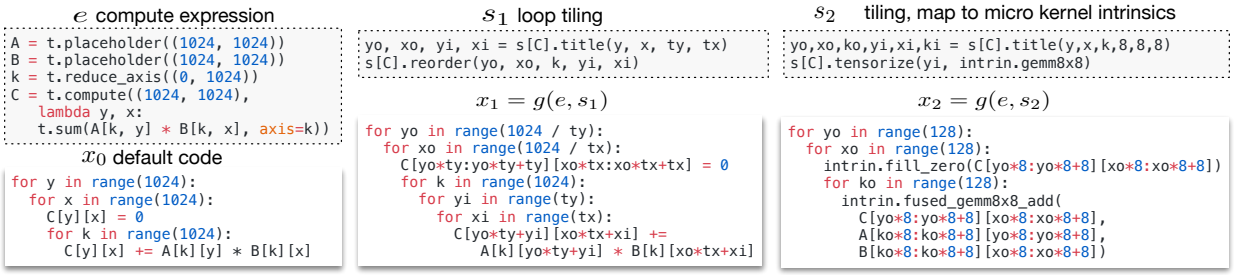


Figure 5.1: Sample problem. For a given tensor operator specification ($C_{ij} = \sum_k A_{ki} B_{kj}$), there are multiple possible low-level program implementations, each with different choices of loop order, tiling size, and other options. Each choice creates a logically equivalent program with different performance. Our problem is to explore the space of programs to find the fastest implementation.

This problem formalization is similar to that of traditional hyper-parameter optimization problems [87, 83, 88, 38, 48, 60] but with several key differentiating characteristics:

Relatively Low Experiment Cost. Traditionally, hyper-parameter optimization problems incur a high cost to query f, viz., running experiments could take hours or days. However, the cost of compiling and running a tensor program is a few seconds. This property requires that model training and inference be *fast* ; otherwise, there is no benefit over profiling execution on real hardware. It also means that we can collect more training data during optimization.

Domain-Specific Problem Structure. Most existing hyper-parameter optimization algorithms treat the problem as a black box. As we optimize programs, we can leverage their rich structures to build effective models.

Large Quantity of Similar Operators. An end-to-end DL system must optimize tensor operator programs for different input sizes, shapes, and data layout configurations. These tasks are similar and can offer opportunities for transfer learning.

We describe two key prerequisites for automatic code generation that is competitive with hand-optimized code. (**1**) We need to define an exhaustive search space $\mathcal{S}_e$ that covers all hardware-aware optimizations in hand-tuned libraries. (**2**) We need to efficiently find an optimal schedule in $\mathcal{S}_e$.

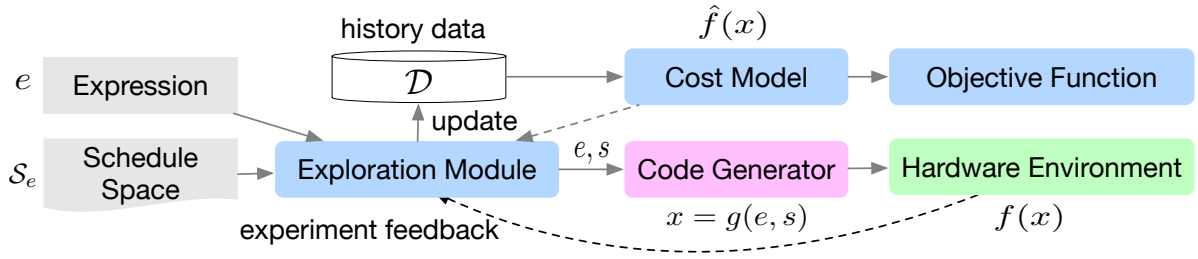


Figure 5.2: Framework for learning to optimize tensor programs.

We use primitives from previous chapter to form $\mathcal{S}_e$. Our search space includes multi-level tiling on each loop axis, loop ordering, shared memory caching for GPUs, and annotations such as unrolling and vectorization. The search space size $|\mathcal{S}_e|$ can be on the order of billions of possible implementations for a single GPU operator.

5.2 Learning to Optimize Tensor Programs

We propose a machine learning (ML)-based framework to solve this problem. Figure 5.2 presents the framework and its modules. We build a statistical cost model $\hat{f}(x)$ to estimate the cost of each low-level program x. An exploration module proposes new schedule configurations to run on the hardware. The run time statistics are collected in a database $\mathcal{D} = \{(e_i, s_i, c_i)\}$, which can in turn be used to update $\hat{f}$. We discuss module-specific design choices in the following subsections.

5.2.1 Statistical Cost Model

The first statistical model we support is based on gradient boosted trees [34](GBTs). We extract domain-specific features from a given low-level abstract syntax tree (AST) x. The features include loop structure information (e.g., memory access count and data reuse ratio) and generic annotations (e.g., vectorization, unrolling, thread binding). We use XGBoost [17], which has proven to be a strong feature-based model in past problems. Our second model is a TreeGRU[94], which recursively encodes a low-level AST into an embedding vector. We map the embedding vector to

Algorithm 8: Learning to Optimize Tensor Programs

Input : Transformation space $\mathcal{S}_e$

Output: Selected schedule configuration s^*

$\mathcal{D} \leftarrow \emptyset$

while *n_trials* $<$ *max_n_trials* **do**

```
// Pick the next promising batch
```

$Q \leftarrow$ run parallel simulated annealing to collect candidates in $\mathcal{S}_e$ using energy function $\hat{f}$

$S \leftarrow$ run greedy submodular optimization to pick $(1-\epsilon)b$-subset from Q by maximizing Equation 5.3

$S \leftarrow S \cup \{$ Randomly sample ϵb candidates. $\}$

```
// Run measurement on hardware environment
```

for s ***in*** S **do**

  $c \leftarrow f(g(e,s)); \mathcal{D} \leftarrow \mathcal{D} \cup \{(e,s,c)\}$

end

```
// Update cost model
```

update $\hat{f}$ using $\mathcal{D}$

n_trials $\leftarrow$ n_trials $+ b$

end

$s^* \leftarrow$ history best schedule configuration

a final predicted cost using a linear layer.

GBT and TreeGRU represent two distinct ML approaches to problem resolution. Both are valuable, but they offer different benefits. GBT relies on precise feature extraction and makes fast predictions using CPUs. TreeGRU, the deep learning-based approach, is extensible and requires no feature engineering, but it lags in training and predictive speed. We apply batching to the TreeGRU model and use GPU acceleration to make training and prediction fast enough to be usable in our framework.

5.2.2 *Training Objective Function*

We can choose from multiple objective functions to train a statistical cost model for a given collection of data $\mathcal{D} = \{(e_i, s_i, c_i)\}$. A common choice is the regression loss function $\sum_i (\hat{f}(x_i) - c_i)^2$, which encourages the model to predict cost accurately. On the other hand, as we care only about the relative order of program run times rather than their absolute values in the selection process, we can instead use the following rank loss function [14]:

$$\sum_{i,j} \log(1 + e^{-\operatorname{sign}(c_i - c_j)(\hat{f}(x_i) - \hat{f}(x_j))}). \tag{5.2}$$

We can use the prediction $\hat{f}(x)$ to select the top-performing implementations.

5.2.3 *Exploration Module*

The exploration module controls the search loop, which is summarized in Algorithm 8. At each iteration, it must pick a batch of candidate programs based on $\hat{f}(x)$ and query $f(x)$ on real hardware. We cannot simply enumerate the entire space of S_e and pick the top-b candidates due to the size of the search space. Instead, we use simulated annealing [53] with $\hat{f}(x)$ as the energy function. Specifically, we use a batch of parallel Markov chains to improve the prediction throughput of the statistical cost model. We select the top-performing batch of candidates to run on real hardware. The collected performance data is used to update $\hat{f}$. We make the states of the Markov chains persistent across $\hat{f}$ updates. We also apply the ϵ-greedy to select ϵb (e.g. 0.05) candidates randomly to ensure exploration.

Diversity-Aware Exploration. We consider both quality and diversity when selecting b candidates for hardware evaluation. Assume that the schedule configuration s can be decomposed into m components $s = [s_1, s_2, \cdots s_m]$. We maximize the following objective to select candidate set S from the top λb candidates:

$$L(S) = -\sum_{s \in S} \hat{f}(g(e, s)) + \alpha \sum_{j=1}^{m} |\cup_{s \in S} \{s_j\}| \tag{5.3}$$

The first term encourages us to pick candidates with low run time costs. The second term counts the number of different configuration components that are covered by S. $L(S)$ is a submodular function, and we can apply the greedy algorithm [70, 57] to get an approximate solution.

Uncertainty Estimator. Bayesian optimization methods [87, 83, 88, 48] use acquisition functions other than the mean when an uncertainty estimate of $\hat{f}$ is available. Typical choices include expected improvement (EI) and upper confidence bound (UCB). We can use bootstrapping to get the model's uncertainty estimate and validate the effectiveness of these methods. As we will see in section 5.4 , considering uncertainty does not improve the search in our problem. However, the choice of acquisition function remains a worthy candidate for further exploration.

5.3 Accelerating Optimization via Transfer Learning

Thus far, we have focused only on learning to optimize a single tensor operator workload. In practice, we need to optimize for many tensor operators with different input shapes and data types. In a real world setting, the system collects historical data $\mathcal{D}'$ from previously seen workloads. We can apply transfer learning to effectively use $\mathcal{D}'$ to speed up the optimization.

The key to transfer learning is to create a **transferable representation** that is **invariant** to the source and target domains. We can then share the cost model using the common representation across domains. Different choices of representations may have different levels of invariance.

A common practice in Bayesian optimization methods is to directly use configuration s as the model's input. However, the search space specification can change for different workloads or when the user specifies a new search space for the same workload. The configuration representation s is not invariant to changes in the search space.

On the other hand, the low-level loop AST x (Figure 5.3a) is a shared representation of programs that is invariant to the search space. To leverage this invariance, our cost model $\hat{f}(x)$ takes the low-level loop AST x as input. We also need to encode x into a vector space to perform prediction. The specific encoding of x can also result in different levels of invariance.

Context Relation Features for GBT. We define context features at each loop level to represent

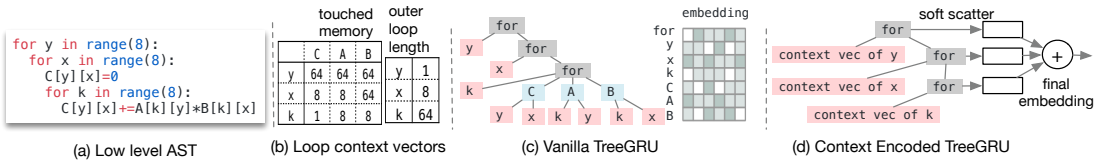


Figure 5.3: Possible ways to encode the low-level loop AST.

loop characteristics. A simple representation of context features is a vector (e.g., in Figure 5.3b, where each loop has a row of features). Context features are informative but, crucially, cannot generalize across different loop nest patterns; we define context relation features to overcome this issue.

To build context relation features, we treat context vectors as a bag of points and extract features that model relations between feature axes. Formally, let Z be the context feature matrix such that Z_{ki} corresponds to the i-th feature of loop k. We define a set of $\log_2$-spaced constant thresholds $\beta = [\beta_1, \beta_2, \cdots \beta_m]$. The relation feature between feature i and j is defined as: $R_t^{(ij)} = \max_{k \in \{k | Z_{kj} < \beta_t\}} Z_{ki}$. This encoding summarizes useful relations, such as loop count vs. touched memory size (related to the memory hierarchy of the access), that affect run time cost.

Context Encoded TreeGRU. The invariant representation also exists for the neural-based model. Figure 5.3c shows a way to encode the program by learning an embedding vector for each identifier and summarizing the AST using TreeGRU. This model works well for modeling a single workload. However, the set of loop variables can change across different domains, and we do not have embedding for the new loop variables. We instead encode each loop variable using the context vector extracted for GBT to summarize the AST (Figure 5.3d). We scatter each loop level, embedding h into m vectors using the rule $out_i = \text{softmax}(W^T h)_i h$. Conceptually, the softmax classifies the loop level into a memory slot in out. Then, we sum the scattered vectors of all loop levels to get the final embedding.

Once we have a transferable representation, we can use a simple transfer learning method by

Workload Name	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
H, W	224,224	56,56	56,56	56,56	56,56	28,28	28,28	28,28	14,14	14,14	14,14	7,7
IC, OC	3,64	64,64	64,64	64,128	64,128	128,128	128,256	128,256	256,256	256,512	256,512	512,512
K, S	7,2	3,1	1,1	3,2	1,2	3,1	3,2	1,2	3,1	3,2	1,2	3,1

Table 5.1: Configurations of all conv2d operators in a single batch ResNet-18 inference. H,W denotes height and width, IC input channels, OC output channels, K kernel size, and S stride size.

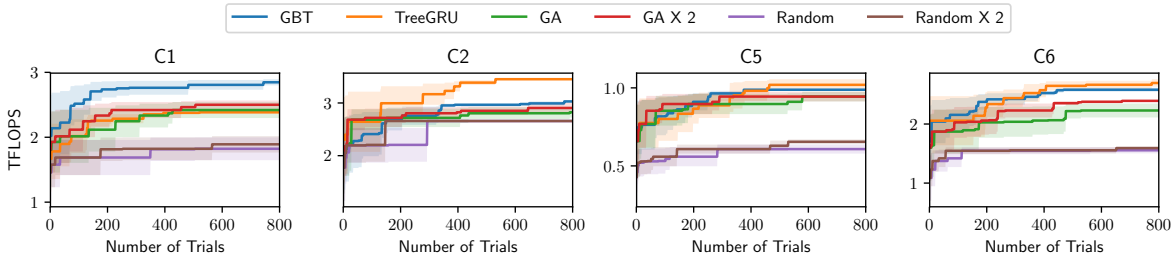


Figure 5.4: Statistical cost model vs. genetic algorithm (GA) and random search (Random) evaluated on NVIDIA TITAN X. 'Number of trials' corresponds to number of evaluations on the real hardware. We also conducted two hardware evaluations per trial in Random $\times 2$ and GA $\times 2$. Both the GBT- and TreeGRU-based models converged faster and achieved better results than the black-box baselines.

combining a global model and an in-domain local model, as follows:

$$\hat{f}(x) = \hat{f}^{(global)}(x) + \hat{f}^{(local)}(x). \tag{5.4}$$

The global model $\hat{f}^{(global)}(x)$ is trained on $\mathcal{D}'$ using the invariant representation; it helps to make effective initial predictions before we have sufficient data to fit $\hat{f}^{(local)}(x)$.

5.4 Evaluations

5.4.1 Component Evaluations

We first evaluated each design choice in the framework. Component evaluations were based on convolution workloads in ResNet-18 [42] for ImageNet classification (Table 5.1). Due to space

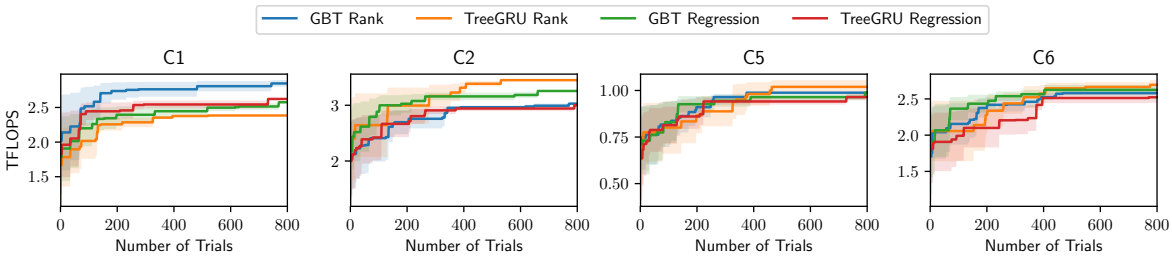


Figure 5.5: Rank vs. Regression objective function evaluated on NVIDIA TITAN X. The rank-based objective either outperformed or performed the same as the regression-based objective in presented results.

limitations, we show component evaluation results only on representative workloads; the complete set of results is reported in the supplementary material. All methods compared in this subsection were initialized with no historical data. Section 5.4.2 evaluates the transfer learning setting.

Importance of Statistical Cost Model. Figure 5.4 compares the performance of the statistical cost model to black-box methods. Both the GBT and TreeGRU models outperformed the black-box methods and found operators that were $2\times$ faster than those found with random searches. This result is particularly interesting compared to prior results in hyper-parameter tuning [60], where model-based approaches were shown to work only as well as random searching. Our statistical models benefit from domain-specific modeling and help the framework find better configurations.

Choice of Objective Function. We compared the two objective functions in Figure 5.5 on both types of models. In most cases, we found that using a rank-based objective was slightly better than using a regression-based one: the rank-based objective may have sidestepped the potentially challenging task of modeling absolute cost values. We chose rank as our default objective.

Impact of Diversity-Aware Exploration. We evaluated the impact of the diversity-aware exploration objective in Figure 5.6. Most of the workloads we evaluated showed no positive or negative impact for diversity-based selection. However, diversity-aware exploration improved C6, which shows some potential usefulness to the approach. We adopted the diversity-aware strategy since it

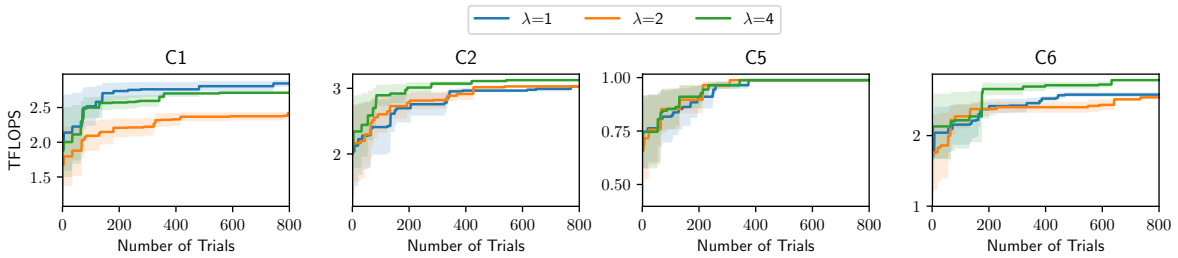


Figure 5.6: Impact of diversity-aware selection with different choices of λ evaluated on NVIDIA TITAN X. Diversity-aware selection had no positive or negative impact on most of the evaluated workloads.

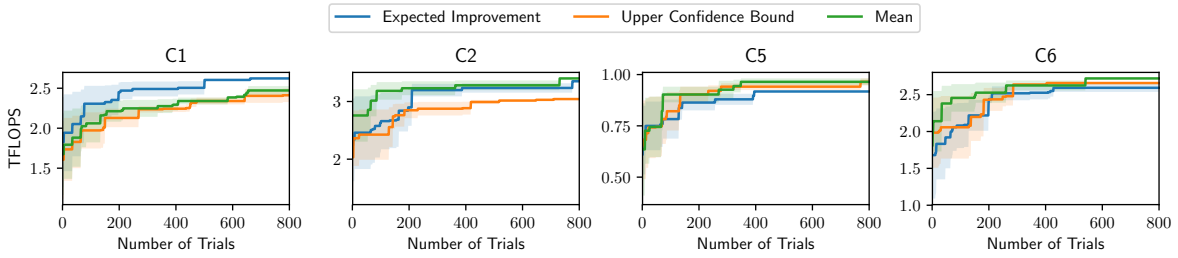


Figure 5.7: Impact of uncertainty-aware acquisition functions evaluated on NVIDIA TITAN X. Uncertainty-aware acquisition functions yielded no improvements in our evaluations.

can be helpful, has no meaningful negative impact, and negligibly affects run time.

Impact of Uncertainty Estimator. We evaluated the usefulness of uncertainty-aware acquisition functions in Figure 5.7. The uncertainty measurement was achieved by training five models using bootstrapping. We used the regression objective in this setting—similar to its use in most Bayesian optimization methods. Results show that uncertainty estimation was not as important in our problem, possibly because our models were trained with more training samples than traditional hyper-parameter optimization problems.

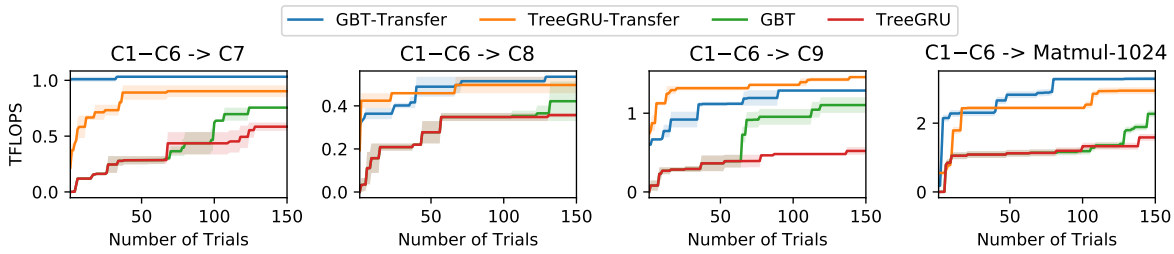


Figure 5.8: Impact of transfer learning. Transfer-based models quickly found better solutions.

5.4.2 Transfer Learning Evaluations

The evaluations presented so far used no historical data. This subsection evaluates the improvements obtainable with transfer learning.

Improvements by Transfer. We first evaluated general improvements made possible by transfer learning. We randomly picked samples from $\mathcal{D}'$ collected from C1,C2,C3,C4,C5,C6 and used them to form the source domain (30000 samples in the TITAN X experiment and 20000 samples in the ARM GPU and ARM A53 experiments). We then compared the performance of transfer-enabled methods to learning from scratch for target workloads C7,C8,C9. Results are shown in Figure 5.8. Overall, using transfer learning yielded a $2\times$ to $10\times$ speedup. This approach is especially important for real DL compilation systems, which continuously optimize incoming workloads.

Invariant Representation and Domain Distance. As discussed in Section 5.3, different representations have different levels of invariance. We used three scenarios to study the relationship between domain distance and the invariance of feature representations: (1) running optimizations on only one target domain; (2) C1–C6→7: C1–C6 as source domain and C7 as target domain (transfer within same operator type); (3) C1–C6→Matmul-1024: C1–C6 as source domain and matrix multiplication as target domain (transfer across operator types). Results (Figure 5.9) show the need for more invariance when domains are farther apart. Using our transferable feature representation, our model generalized across different input shapes and operator types. We also ran a preliminary study on transfer from an ARM Mali GPU to an ARM Cortex-A53 (Figure 5.9d), showing

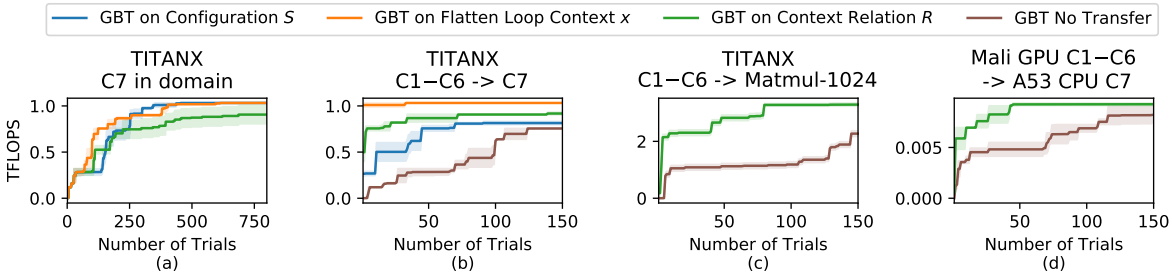


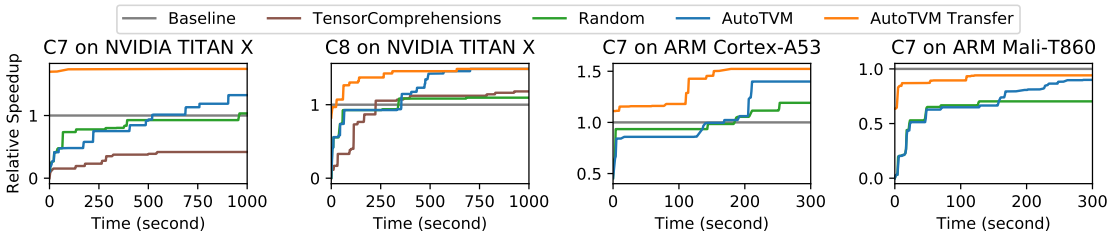
Figure 5.9: Comparison of different representations in different transfer domain settings. The configuration-based model can be viewed as a typical Bayesian optimization approach (batched version of SMAC [48]). We found that models using configuration space features worked well within a domain but were less useful across domains. The flattened AST features worked well when transferring across convolution workloads but were not useful across operator types. Context relation representation allowed effective transfer across operator types.

that the proposed representation enabled transfer across devices. Developing an invariant feature representation poses a difficult problem worthy of additional research.

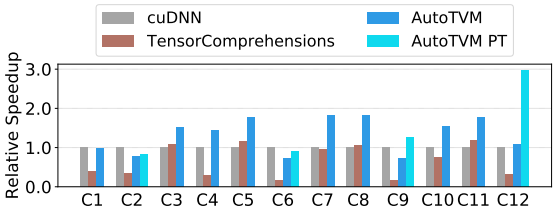
5.4.3 Comparing against Existing Frameworks

Thus far, our evaluation has focused on specific design choices in our framework. We now segue to the natural follow-up question: can learning to optimize tensor programs improve real-world deep learning systems on diverse hardware targets? We call our framework AutoTVM. We compared our approach to existing DL frameworks backed by highly engineered hardware-specific libraries on diverse hardware back-ends: a server class GPU, an embedded CPU, and a mobile GPU. Note that AutoTVM performs optimization and code generation *with no external operator library*.

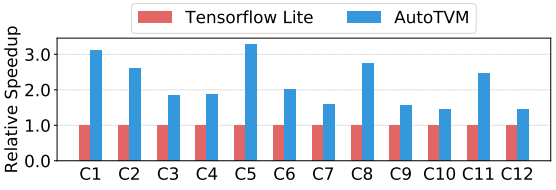
We evaluated single-operator optimization against baselines that used hardware-specific libraries. The baselines were: cuDNN v7 for the NVIDIA GPU, TFLite(commit: 7558b085) for the Cortex-A53, and the ARM Compute Library (v18.03) for the ARM Mali GPU. We also included TensorComprehensions (commit: ef644ba) [99] as an additional baseline for the TITAN



(a) Optimization curves in wall clock time. (We set cuDNN v7, Tensorflow Lite and ARM ComputeLibrary v18.03 as the baselines for TITAN X, ARM A53 and ARM Mali-T860, respectively.)



(b) NVIDIA TITAN X Single Op



(c) ARM Cortex-A53 Single Op

Figure 5.10: Single operator performance optimizations on the TITAN X and ARM CPU/GPU.

X [1] TensorComprehensions used 2 random seeds $\times 25$ generations $\times 200$ population for each operator, and padding was removed (TC does not yet support padding). The results are shown in Figure 5.10. AutoTVM generated high-performance tensor programs across different hardware back-ends.

[1] According to personal communication [98], TC is not yet intended for use in compute-bound problems. However, it still provides a good reference baseline for inclusion in the comparison.

Chapter 6

CONCLUSION

This thesis introduced learning systems that scale up machine learning and use machine learning to improve those very systems. This chapter summarizes our results and discuss potential future directions.

6.1 Thesis Summary

Chapter 2 described the lessons we learned when building XGBoost, a scalable tree boosting system that is widely used by data scientists and provides state-of-the-art results on many problems. We proposed a novel sparsity aware algorithm for handling sparse data and a theoretically justified weighted quantile sketch for approximate learning. Our experience shows that cache access patterns, data compression, and sharding are essential elements for building a scalable end-to-end system for tree boosting. These lessons can be applied to other machine learning systems as well. By combining these insights, XGBoost is able to solve real-world scale problems using a minimal amount of resources.

We introduced MXNet in Chapter 3. We discussed the necessary programming interface to make the system accessible. We then proposed a systematic approach to reduce the memory consumption of training By combining these insights, MXNet is able to scale up real-world deep learning workloads using a minimal amount of resources.

In Chapter 4, we proposed an end-to-end compilation stack to solve fundamental optimization challenges for deep learning across a diverse set of hardware back-ends. Our system includes automated end-to-end optimization, which is historically a labor-intensive and highly specialized task. We hope this work will encourage additional studies of end-to-end compilation approaches and open new opportunities for DL system software-hardware co-design techniques.

In Chapter 5, we presented AutoTVM: a machine learning-based framework that automatically optimizes the implementation of tensor operators in deep learning systems. Our statistical cost model allows effective model sharing between workloads and speeds up the optimization process via model transfer. The positive experimental results of this new approach show promise for DL deployment. Beyond our solution framework, the specific characteristics of this new problem make it an ideal testbed for innovations in related areas, such as neural program modeling, Bayesian optimization, transfer learning, and reinforcement learning.

6.2 Discussion: Elements of Learning Systems

Although the three learning systems introduced in this thesis are independent of each other, we can find several common elements from them. These elements are key ingredients of many other learning systems as well.

Scalability is the first element. It is also a quite natural one, as many of the progress are made by scaling learning to bigger datasets and more complicated models. XGBoost's ability to scale to billions of examples has driven a lot of industrial adoptions. The sublinear memory training also becomes more and more important due to the limited on-chip memory in hardware accelerators.

Accessiblity is the second element that is often ignored by people. A learning system would have much a bigger impact if it can be made accessible to more people. XGBoost's automatic missing value handling helped data scientists to directly use the system without worrying about how to clean up their data, and it has become one of the most popular features of the system. The mixed programming model in MXNet is also mainly designed for better usability. Accessiblity starts to get more attention from the community. The recent success of Keras [27] and PyTorch [73] are perfect examples of the importance of accessibility.

Full stack is the third element. The trend of hardware specialization forces us to rethink the boundaries in the software and hardware. By co-designing the compiler and hardware abstraction together, the VTA accelerator in TVM brought much complexity of the hardware into the software. This approach also makes the end to end solution more flexible to support a diverse set of deep

learning models.

Intelligence is the last element that is emerging. AutoTVM uses machine learning to automatically perform program optimizations that are traditionally done by system engineers. There are also other concurrent efforts in the community to put intelligence into existing systems and use machine learning to replace heuristics in these systems [67, 56].

6.3 Future Work

While this thesis touched many elements of learning systems mentioned in the last section, there are still many interesting open directions that we can continue to explore in the future, especially in the intelligence and full-stack directions. We should build future intelligent systems that will incorporate the following elements: automated system-level optimization via learning; higher degrees of hardware specialization to scale system capabilities; lifelong evolution as new data, models, and hyperparameters are being introduced. It is essential to rethink system design and machine learning under these disruptions.

Learning to explore and optimize structural spaces. Objects of interest in systems usually involve structured data, such as programs, query plans, and hardware designs. We could design novel models to take benefit of this structure and methods to explore this space effectively.

Full stack system co-design. End to end efficiency of the smart applications depends on the combined choice of model, system, and hardware abstraction. For example, models for optimized CPU performance(e.g., MobileNet) do not necessarily fit well into an ASIC due to the limitation of the ASIC's tensor core compute instruction. We can design model variants that maximize ASIC utilization. We also need to change the hardware to better support the models of interest. NLP applications present a new set of requirements such as sparse embedding lookup and dynamic control flow. We need to bake them into the hardware-software abstraction to build a better stack for these applications. We could design models, learning algorithms and hardware jointly, and use system-level requirements to inform the design of the layers that compose the hardware-software stack to better approach problems of interest in NLP, speech, vision and other domains.

Effective knowledge transfer and lifelong learning. To enable smooth transitions between stages in the lifelong model evolution process, We need to apply knowledge transfer to bootstrap new models with already pre-trained models. Our previous work [16] provided a step in this direction by proposing the first method for knowledge transfer from a small neural network to a bigger one. It would be interesting to apply knowledge transfer to new problems like data representation transfer and hardware-aware deep neural architecture search.

BIBLIOGRAPHY

[1] NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU, 2017.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Yongqiang Wang, Huaming Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, August 2014.

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[6] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed

Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 138–149, Washington, DC, USA, 2015. IEEE Computer Society.

[7] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[8] M. Bedford Taylor. Bitcoin and the age of bespoke silicon. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, Sep. 2013.

[9] Ron Bekkerman. The present and the future of the kdd cup competition: an outsider's perspective.

[10] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, New York, NY, USA, 2011.

[11] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of the KDD Cup Workshop 2007*, pages 3–6, New York, August 2007.

[12] Leo Breiman. Random forests. *Maching Learning*, 45(1):5–32, October 2001.

[13] C. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11:23–581, 2010.

[14] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[15] Olivier Chapelle and Yi Chang. Yahoo! Learning to Rank Challenge Overview. *Journal of Machine Learning Research - W & CP*, 14:1–24, 2011.

[16] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2Net: Accelerating learning via knowledge transfer. In *International Conference on Learning Representations (ICLR)*, 2016.

[17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[18] Tianqi Chen, Hang Li, Qiang Yang, and Yong Yu. General functional matrix factorization using gradient boosting. In *Proceeding of 30th International Conference on Machine Learning (ICML'13)*, volume 1, pages 436–444, 2013.

[19] 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. In *Neural Information Processing Systems, Workshop on Machine Learning Systems (LearningSys'15)*, 2015.

[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[21] Tianqi Chen, Sameer Singh, Ben Taskar, and Carlos Guestrin. Efficient second-order gradient boosting for conditional random fields. In *Proceeding of 18th Artificial Intelligence and Statistics Conference (AISTATS'15)*, volume 1, 2015.

[22] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[23] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Neural Information Processing Systems 2018*.

[24] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[25] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[26] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[27] François Chollet et al. Keras. `https://keras.io`, 2015.

[28] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015.

[29] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[31] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Sept 1997.

[32] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[33] Jerome H. Friedman and Bogdan E. Popescu. Importance sampled learning ensembles, 2003.

[34] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

[35] J.H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

[36] J.H. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28(2):337–407, 2000.

[37] M. Frigo and S. G. Johnson. Fftw: an adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, May 1998.

[38] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495. ACM, 2017.

[39] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 58–66, 2001.

[40] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, March 2000.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[43] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14, 2014.

[44] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[45] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[47] 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. *CoRR*, abs/1704.04861, 2017.

[48] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[49] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32th International Conference on Machine Learning (ICML'15)*, 2015.

[50] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[52] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[53] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[54] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[55] Andreas Klöckner. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*, Edinburgh, Scotland., 2014. Association for Computing Machinery.

[56] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[57] Andreas Krause and Daniel Golovin. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, February 2014.

[58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.

[59] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 4013–4021, 2016.

[60] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016.

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

[62] P. Li. Robust Logitboost and adaptive base class (ABC) Logitboost. In *Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI'10)*, pages 302–311, 2010.

[63] Ping Li, Qiang Wu, and Christopher J. Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in Neural Information Processing Systems 20*, pages 897–904. 2008.

[64] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 369–381, New York, NY, USA, 2015. ACM.

[65] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[66] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[67] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2430–2439. JMLR.org, 2017.

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

[69] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[70] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.

[71] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[72] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *Proceeding of VLDB Endowment*, 2(2):1426–1437, August 2009.

[73] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[74] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[75] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[76] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[77] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM.

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

[79] Steffen Rendle. Factorization machines. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 995–1000, Washington, DC, USA, 2010. IEEE Computer Society.

[80] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1135–1144, New York, NY, USA, 2016. ACM.

[81] Greg Ridgeway. *Generalized Boosted Models: A guide to the gbm package*.

[82] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 338–342, 2014.

[83] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, Jan 2016.

[84] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[85] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever,

Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[86] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[87] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012.

[88] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2171–2180, 2015.

[89] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[90] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. *arXiv preprint arXiv:1507.06228*, 2015.

[91] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press.

[92] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 609–616, USA, 2011.

[93] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.

[94] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[95] Andrew Tulloch and Yangqing Jia. High performance ultra-low-precision convolutions on mobile devices. *arXiv preprint arXiv:1712.02427*, 2017.

[96] S. Tyree, K.Q. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*, pages 387–396. ACM, 2011.

[97] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[98] Nicolas Vasilache. personal communication.

[99] 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. *CoRR*, abs/1802.04730, 2018.

[100] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[101] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[102] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, University of California at Berkeley, 2016.

[103] Richard Wei, Vikram Adve, and Lane Schwartz. Dlvm: A modern compiler infrastructure for deep learning systems. *CoRR*, abs/1711.03016, 2017.

[104] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[105] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[106] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09.

[107] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[108] Qi Zhang and Wei Wang. A fast algorithm for approximate quantiles in high speed data streams. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, 2007.

[109] Tong Zhang and Rie Johnson. Learning nonlinear functions using regularized greedy forest. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(5), 2014.

[110] Yu Zhang, Guoguo Chen, Dong Yu, Kaisheng Yao, Sanjeev Khudanpur, and James Glass. Highway long short-term memory rnns for distant speech recognition. *arXiv preprint arXiv:1510.08983*, 2015.

[111] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized stochastic gradient descent. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2595–2603. Curran Associates, Inc., 2010.