Value Function Based Performance Optimization of Deep Learning Workloads

Benoit Steiner
Facebook AI Research
benoitsteiner@fb.com

Chris Cummins
Facebook AI Research
cummins@fb.com

Horace He
Facebook
chilli@fb.com

Hugh Leather
Facebook AI Research
hleather@fb.com

Abstract

As machine learning techniques become ubiquitous, the efficiency of neural network implementations is becoming correspondingly paramount. Frameworks, such as Halide and TVM, separate out the algorithmic representation of the network from the schedule that determines its implementation. Finding good schedules, however, remains extremely challenging. We model this scheduling problem as a sequence of optimization choices, and present a new technique to accurately predict the expected performance of a partial schedule. By leveraging these predictions we can make these optimization decisions greedily and rapidly identify an efficient schedule. This enables us to find schedules that improve the throughput of deep neural networks by $2.6\times$ over Halide and $1.5\times$ over TVM. Moreover, our technique is two to three orders of magnitude faster than that of these tools, and completes in seconds instead of hours.

1 Introduction

The rise of deep learning has been accompanied by the development of frameworks such as PyTorch [1] and TensorFlow [2]. The majority of these tools provide a set of primitive tensor operators to perform tasks such as matrix multiplication, convolution and pooling, which are applied in sequence by a runtime interpreter to derive the outputs of the neural network from its input tensors.

Though pervasive, this approach has two main downsides. First, it requires the development, optimization, and maintenance of a large set of operators, which necessitates scarce human expertise. As a result, frameworks tend to focus on the most common operators which are only optimized for a limited set of use cases, leaving a lot of performance on the table. Second, these operators can only exchange data through global memory. This is a significant bottleneck, especially in the case of operators of low arithmetic intensity such as activation functions.

To avoid these problems, projects such as Halide [3] and TVM [4] proposed to represent tensor computations using a declarative domain specific language based on Einstein's notation. This high-level representation is then compiled into assembly code that can be executed directly on hardware. This approach abstracts away key implementation choices such as loop ordering, blocking, vectorization, unrolling, inlining, or parallelization, and leaves it up to the compiler to figure out which solution, a.k.a. schedule, most efficiently leverages the available hardware resources.

34th Conference on Neural Information Processing Systems (NeurIPS 2020).

Optimization	Description
`split`	Transform a loop into two nested loops.
`reorder`	Exchange two loops.
`vectorize`	Use SIMD instructions to encode the loop.
`parallel`	Parallelize the computation over multiple CPU cores.
`compute_at`	Inline the evaluation of a loop into another one.
`store_at`	Store the values generated by a layer into a temporary buffer.

Table 1: Primitive scheduling actions used to optimize each layer of a neural network.

Previous work [5, 6, 7, 8, 9] has attempted to tackle scheduling by framing the problem as a search in the space of valid implementations. However, these approaches rely on an extensive exploration of the optimization landscape, coupled with aggressive pruning of the solution space. Nevertheless, given the combinatorial nature of the problem, they take several hours to identify a solution and often end up generating suboptimal code.

In this work, we propose a method to overcome these limitations. We iteratively improve a value function capable of predicting, given a partial set of scheduling decisions, the best achievable performance over all remaining decisions. This "look-ahead" allows us to incrementally schedule an entire neural network by making a set of local decisions that are globally optimal. We show that our technique is able to identify schedules which are on average $1.5\times$ and $2.6\times$ faster than TVM and Halide respectively. We also demonstrate that our approach identifies solutions in two or more orders of magnitude less time.

2 Automated Scheduling

We model the problem of choosing the best schedule amongst all the possible options as a deterministic Markov Decision Process (MDP) over a finite horizon with a dynamic action space. In our formulation, a state s_i is a partial schedule – a set of scheduling decisions applied to the first i layers L_i of a neural network. The set of actions a_i available in state s_i is the set of valid scheduling options available for layer L_{i+1} given the loop structure for the previous layers imposed by the schedule s_i. The set of candidate actions we consider is listed in Table 1.

Algorithm 1 Scheduling

Input1: NN with n layers L_1 ... L_n
Input2: Value function $V(s)$
s_0 = InitialState
for $i = 1$ **to** n **do**
  C_i = CandidateActions(L_i, s_{i-1})
  $v_i = \infty$
  for s **in** C_i **do**
    if $V(s) < v_i$ **then**
      v_i = V(s)
      $s_i = s$
    end if
  end for
end for
Return: s_1 ... s_n

We solve the MDP by learning an approximation $V(s)$ of the optimal value function $V^*(s)$. In layman's terms, $V^*(s)$ is a function capable of predicting the lowest runtime achievable from a state s assuming that we make optimal scheduling decisions for all the subsequent layers of the neural network. Once we have our value function approximation $V(s)$, we greedily schedule the neural network layer by layer following the steps outlined in Algorithm 1.

For a N-layer neural network, with an average of M choices available per layer, we only need to consider $N*M$ candidates out of the M^N available complete schedules. This enables us to schedule deep learning workloads extremely quickly as we will see in Section 4.2.

Note that if we could learn the true value function $V^*(s)$ instead of an approximation $V(s)$ our approach would ensure that we find the optimum solution. We'll see in section 4.1 how each iteration improve the performance of the schedules identified by our approach.

3 Value Function Estimation

3.1 Iterative Approximation

Our technique is inspired from the well known value iteration approach summarized in [10]. However, we cannot use this approach directly. Indeed, we face two main hurdles. First, obtaining the reward is very expensive – it requires compilation of our schedule as well as benchmarking the schedule on

actual hardware. Second, it would be impractical to exhaustively visit all the states associated with the scheduling of a single neural network for all but the smallest networks.

On the other hand, we can make a few simplifications. First, we do not need to uniformly sample all the actions available from a state s. Although we need a precise estimate of the value function associated with the "best" states, we only need a rough estimate for the less interesting states. Consequently, we can sample less often the actions that lead to the less interesting states. Moreover, we can derive an upper bound for the value function for a state s by searching for the best schedule starting from s.

Based on these observations, we designed an iterative algorithm, that, starting from an initial approximation of the value function V_0, builds progressively more accurate estimates $V_i(s)$.

Algorithm 2 Single iteration of our value function approximation

Input: set of neural networks N
Input: value function $V_{i-1}(s)$
Initialize $V_i(s)$ to $V_{i-1}(s)$
for n **in** N **do**
  for k **in** $[0, 100]$ **do**
    $s_0, ...s_n = \text{BestSchedule}(n, V_{i,\epsilon_k})$
    for s_j **in** $s_0, ...s_n$ **do**
      $t_{j+1}, ...t_n = \text{BeamSearch}(s_j, V_{i-1}))$
      $r = \text{Benchmark}(s_0, ...s_j, t_{j+1}, ...t_n)$
      $V_i(s_j) = \min(r, V_i(s_j))$
    end for
  end for
end for
Return: $V_i(s)$

Algorithm 2 details how we perform each iteration: we extract 100 schedules for each neural network in our training set. We inject a small amount of random noise ϵ to the predictions made by the value function to ensure that we cover a significant portion of the interesting states of each pipeline. Starting from each state s_i in the previously identified schedules, we run a beam search guided by our previous estimate of the value function, and benchmark the resulting schedule. We use the measured runtime to refine the estimate of the value function V for the state s_i. To bootstrap the process, we modify algorithm 2 to generate and benchmark end to end schedules purely randomly.

3.2 Implementation

The throughput of a neural network depends on two main factors: the amount of computation and data access to be performed, and the overall organization of the computation. Consequently, we devised two groups of input features to capture this information: a set of intrinsic features that are invariant to the schedule, and a collection of schedule dependant features that are acquired as the process of scheduling a pipeline progresses. In this setting, the intrinsic features enable our model to predict how fast each stage could be executed if scheduled optimally, while the acquired features enable us to capture how well the scheduled stages are expected to perform.

Since the set of scheduling decisions made for the initial layers of a neural network impact the performance of the yet to be scheduled layers, we architected our value function around an LSTM, that we feed with the normalized values of our intrinsic and acquired features. To predict the expected performance of the whole neural network, we sum the predictions of each of the timesteps of the LSTM.

4 Evaluation

We ran all our experiments on an Intel Xeon E5-2698 running at 2.2GHz with 48GB of RAM and a NVidia Tesla M40 GPU. We used Halide [3] to compile our schedules into assembly code.

4.1 Value Function

First, we measured the impact of the inductive bias inherent in our model architecture on its ability to encode the value function. Across our rounds of value improvement V_i, our neural network was able to predict the expected values with an average error inferior to 5% and an R^2 greater than 0.955.

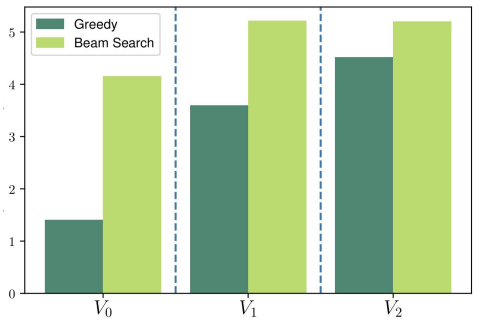


Figure 1: Improvement in our value function through 2 rounds of value improvement. Each round significantly improves the quality of our generated schedules.

Next, we would like to demonstrate that our iterative value learning process is effective. However, we cannot show that our successive value function approximations $V_i(s)$ converge towards the optimal value function $V^*(s)$ since determining the exact value of $V^*(s)$ would require an exhaustive search in a combinatorially large solution space. Instead we show that with each iteration our value function estimates are better able to guide a search. In Figure 1, we plot the relative performance of the schedules selected by our greedy search as well as a standard beam search under the guidance of three successive estimates V0, V1, and V2 on the 12 models used in Figure 2. As V0, V1, and V2 refine the estimates of our value function the gap between the quality of the schedules identified by a greedy and a beam search decreases, while the overall performance of the schedules increases

4.2 Benchmarks

We evaluate our search strategy on a diverse set of deep learning workloads encompassing text, speech, and image processing tasks.

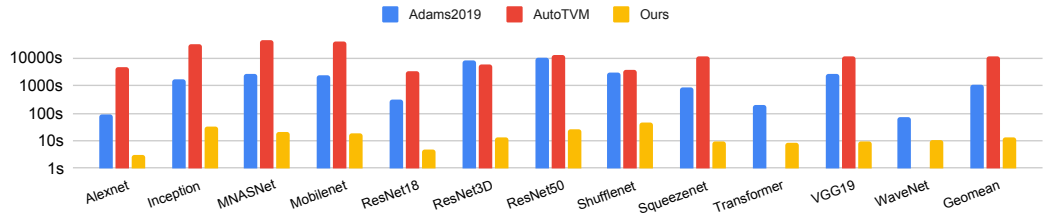
In Figure 2 we compare our implementation against the following systems: PyTorch 1.5, AutoTVM version 0.6, and the Halide auto-scheduler version 8.0.0. To run AutoTVM, we followed the steps outlined in its documentation [11]. We ran the Halide auto-scheduler with its default settings of 5 search passes with each pass identifying 32 candidate schedule, and benchmarking to do the final ranking of the 160 candidates.

First, we show the search time of the systems in Figure 2a. PyTorch is extremely fast, since it does not search for good solutions. AutoTVM takes considerable time, requiring 3 hours on average and up to 12 hours to complete the search. Halide takes an average of 20 minutes and up to 2.5 hours. Our approach schedules the neural networks in 13 seconds on average, and 47 seconds in the worst case.

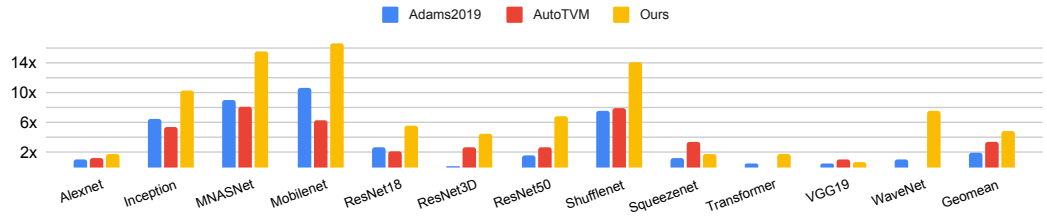
However, this significantly faster search time does not come at the cost of lower quality schedules, as can be seen in Figure 2b. Our search finds schedules that outperform PyTorch by a factor of $5.1\times$. We also improve on AutoTVM and Halide with a speedup of $1.5\times$ and $2.6\times$ respectively.

5 Conclusion

Our results demonstrate that automatically learning complex neural network scheduling policies using reinforcement learning is feasible and lead to better results in a fraction of the search time. We hope that this will allow these techniques to be used in a broader range of applications.



(a) Schedule search time. Lower is better.



(b) Speedup of schedules over PyTorch. Higher is better.

Figure 2: (a) Time taken to schedule deep learning workloads, and (b) performance of final schedules relative to PyTorch. We plot the search times on a log scale. We compare our results against the Halide autoscheduler (Adams2019) and AutoTVM using the published configuration. AutoTVM failed to load the Transformer and Wavenet models.

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[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, Dandelion 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] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics - TOG*, 31, 07 2012.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, 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)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[5] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 3393–3404, USA, 2018. Curran Associates Inc.

[7] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[9] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 254–264, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[11] Yao Wang and Eddie Yan. Auto-tuning a convolutional network for x86 cpu. `https://tvm.apache.org/docs/tutorials/autotvm/tune_relay_x86.html`. Accessed: 2020-09-30.