OpEvo: An Evolutionary Method for Tensor Operator Optimization

Xiaotian Gao, Wei Cui, Lintao Zhang, Mao Yang
Microsoft Research
xiaotian.gao, weicu, lintaoz, maoyang@microsoft.com

Abstract

Training and inference efficiency of deep neural networks highly rely on the performance of tensor operators on hardware platforms. Manually optimizing tensor operators has limitations in terms of supporting new operators or hardware platforms. Therefore, automatically optimizing device code configurations of tensor operators is getting increasingly attractive. However, current methods for tensor operator optimization usually suffer from poor sample-efficiency due to the combinatorial search space. In this work, we propose a novel evolutionary method, OpEvo, which efficiently explores the search spaces of tensor operators by introducing a topology-aware mutation operation based on q-random walk to leverage the topological structures over the search spaces. Our comprehensive experiment results show that compared with state-of-the-art (SOTA) methods OpEvo can find the best configuration with the lowest variance and least efforts in the number of trials and wall-clock time. All code of this work is available online.

1 Introduction

Abundant applications raise the demands of training and inference deep neural networks (DNNs) efficiently on diverse hardware platforms ranging from cloud servers to embedded devices. Moreover, computational graph-level optimization of deep neural network, like tensor operator fusion (Wang, Lin, and Yi 2010), may introduce new tensor operators. Thus, manually optimized tensor operators provided by hardware-specific libraries have limitations in terms of supporting new operators or hardware platforms, so automatically optimizing tensor operators on diverse hardware platforms is essential for large-scale deployment and application of deep learning technologies in the real-world problems.

Tensor operator optimization is essentially a combinatorial optimization problem. The objective function is the performance of a tensor operator on specific hardware platform, which should be maximized with respect to the hyper-parameters of corresponding device code, such as how to tile a matrix or whether to unroll a loop. Thereafter, we will refer to a tuple of hyper-parameters determining device code as a configuration, and the set of all possible configurations as a configuration space or search space. Unlike many typical problems of this type, such as travelling salesman problem, the objective function of tensor operator optimization is a black box and expensive to sample. One has to compile a device code with a specific configuration and run it on real hardware to get the corresponding performance metric. Therefore, a desired method for optimizing tensor operators should find the best configuration with as few samples as possible.

The expensive objective function makes solving tensor operator optimization problem with traditional combinatorial optimization methods, for example, simulated annealing (SA) (Kirkpatrick, Gelatt, and Vecchi 1983) and evolutionary algorithms (EA) (Bäck and Schwefel 1993), almost impossible. Although these algorithms inherently support combinatorial search spaces (Youssef, Sait, and Adiche 2001), they do not take sample-efficiency into account, thus thousands of or even more samples are usually needed, which is unacceptable when tuning tensor operators in product environments. On the other hand, sequential model based optimization (SMBO) methods are proved sample-efficient for optimizing black-box functions with continuous search spaces (Srinivas et al. 2009; Hernández-Lobato, Hoffman, and Ghahramani 2014; Wang and Jegelka 2017). However, when optimizing ones with combinatorial search spaces, SMBO methods are not as sample-efficient as their continuous counterparts (Hutter, Hoos, and Leyton-Brown 2011), because there is lack of prior assumptions about the objective functions, such as continuity and differentiability in the case of continuous search spaces. For example, if one could assume an objective function with a continuous search space is infinitely differentiable, a Gaussian process with a radial basis function (RBF) kernel could be used to model the objective function. In this way, a sample provides not only a single value at a point but also the local properties of the objective function in its neighborhood or even global properties, which results in a high sample-efficiency. In contrast, SMBO methods for combinatorial optimization suffer from poor sample-efficiency due to the lack of proper prior assumptions and corresponding surrogate models.

Besides sample-efficiency, another weakness of SMBO methods is the extra burden introduced by training and optimizing surrogate models. Although it can be safely ignored for many ultra-expensive objective functions, such as hyperparameter tuning and architecture search for neural networks (Elsken, Metzen, and Hutter 2018), in which a trial usually needs several hours or more, but it is not the case in the context of tensor operator optimization, since compiling and executing a tensor operator usually need at most tens of seconds.

In this work, we propose a lightweight model-free method, OpEvo (**Op**erator **Evo**lution), which combines both advantages of EA and SMBO by leveraging prior assumptions on combinatorial objective functions in an evolutionary framework. Although there is no nice property like continuity or differentiability, we construct topological structures over search spaces of tensor operators by assuming similar configurations of a tensor operator will result in similar performance, and then introduce a topology-aware mutation operation by proposing a q-random walk distribution to leverage the constructed topological structures for better trade-off between exploration and exploitation. In this way, OpEvo not only inherits the support of combinatorial search spaces and model-free nature of EA, but also benefits from the prior assumptions about combinatorial objective functions, so that OpEvo can efficiently optimize tensor operators. The contributions of the paper are four-fold:

- We construct topological structures for search spaces of tensor operator optimization by assuming similar configurations of a tensor operator will result in similar performance;
- We define q-random walk distributions over combinatorial search spaces equipped with topological structures for better trade-off between exploitation and exploration;
- We propose OpEvo, which can leverage the topological structures over search spaces by introducing a novel topology-aware mutation operation based on q-random walk distributions;
- We evaluate the proposed algorithm with comprehensive experiments on both Nvidia and AMD platforms. Our experiments demonstrate that compared with state-of-the-art (SOTA) methods OpEvo can find the best configuration with the lowest variance and least efforts in the number of trials and wall-clock time.

The rest of this paper is organized as follows. We summarize the related work in Section 2, and then introduce a formal description of tensor optimization problem and construct topological structures in Section 3. In Section 4, we describe OpEvo method in detail and demonstrate its strength with experiments of optimizing typical tensor operators in Section 5. Finally, we conclude in Section 6.

2 Related Work

As a class of popular methods for expensive black-box optimization, SMBO methods are potential solutions for tensor operator optimization. Although classic SMBO methods, such as Bayesian optimization (BO) with Gaussian process surrogate, are usually used to optimize black-box functions with continuous search spaces, many works have been done in using SMBO to optimize combinatorial black-box functions. Hutter, Hoos, and Leyton-Brown (2011) proposed SMAC, which uses random forest as a surrogate model to optimize algorithm configuration successfully. Bergstra et al. (2011) proposed TPE, which uses tree-structured Parzen estimator as a surrogate model to optimize hyperparameters of neural networks and deep belief networks. As for tensor operator optimization, TVM (Chen et al. 2018a) framework implemented a SMBO method called AutoTVM (Chen et al. 2018b) to optimize configurations of tensor operators. Specifically, AutoTVM fits a surrogate model with either XGBoost (Chen and Guestrin 2016) or TreeGRU (Tai, Socher, and Manning 2015), and then uses SA to optimize the surrogate model for generating a batch of candidates in an ϵ-greedy style. Ahn et al. (2020) proposed CHAMELEON to further improve AutoTVM with clustering based adaptive sampling and reinforcement learning based adaptive exploration to reduce the number of costly hardware and surrogate model measurements, respectively. Ansor (Zheng et al. 2020) is another work built upon TVM. It used EA instead of SA to optimize surrogate models and devised an end-to-end framework to allocate computational resources among subgraphs and hierarchically generate TVM templates for them. Although these methods are successfully used in many combinatorial optimization problems, they are not as sample-efficient as their continuous counterparts due to the lack of proper prior assumptions and corresponding surrogate models. OpEvo, on the other hand, introduces and leverages topological structures over combinatorial search spaces thus obtains better sample and time-efficiency than previous arts.

AutoTVM and CHAMELEON also claimed that they are able to transfer knowledge among operators through transfer learning and reinforcement learning, respectively. However, they seem not so helpful in the context of tensor operator optimization. For transfer learning, the pre-training dataset should be large and diverse enough to cover main information in fine-tuning datasets, like ImageNet (Deng et al. 2009) and GPT-3 (Brown et al. 2020) did, otherwise using a pre-trained model is more likely harmful than beneficial. Many tensor operators needing optimizing are either new types of operators generated by tensor fusion or expected to run on new devices. Neither is suitable for transfer learning. Even if transfer learning works in some cases, pre-training a surrogate model with a large dataset before starting a new search and executing and fine-tuning such model during searching are probably more time and money-consuming than just sampling more configurations on hardwares. For reinforcement learning, its brittle convergence and poor generalization have been widely questioned for many years (Haarnoja et al. 2018; Cobbe et al. 2019b,a). There seems no guarantee that the policy learned by CHAMELEON can generalize to unseen operators so that improve sample-efficiency.

Two operator-specific methods, Greedy Best First Search (G-BFS) and Neighborhood Actor Advantage Critic (N-A2C), have been recently proposed to tune matrix tiling schemes of matrix multiplication (MatMul) operators by taking the relation between different configurations into account (Zhang et al. 2019). They actually introduce a topology over the configuration space of MatMul operator by defining a neighborhood system on it, and further employ a Markov Decision Process (MDP) for exploration over the configuration space. By leveraging a domain-specific topological structure, G-BFS and N-A2C outperform AutoTVM in optimizing MatMul operators. However, these two methods are only designed for tuning tiling schemes of multiplication of matrices with only power of 2 rows and columns, so they are not compatible with other types of configuration spaces. Further, they tend to encounter curse of dimensionality as the number of parameters needed tuning getting bigger, because they only change one parameter at a time based on the MDP they defined. Thus, generalizing them to more general tensor operators is not straightforward. OpEvo, on the other hand, constructs topological structures in a general way and uses evolutionary framework rather than MDP framework to explore search spaces, so that the aforementioned problems encountered by G-BFS and N-A2C are overcame.

3 Problem Formulation

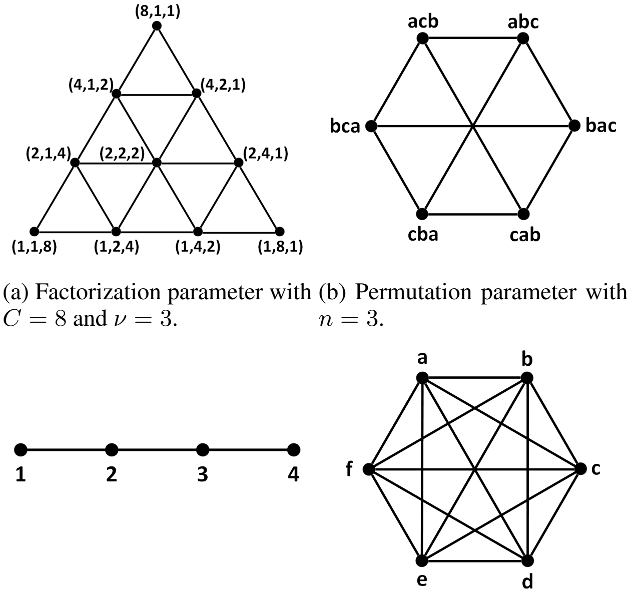
As earlier mentioned, tensor operator optimization is essentially a black-box optimization problem with a combinatorial search space. It can be formally written as

$$x^\star = \underset{x \in \mathbb{X}}{\arg\max}\, f(x),\ \mathbb{X} = \prod_{i=1}^{\mu} \mathcal{X}_i. \qquad (1)$$

Here, $f(x)$ is a black-box function that measures the performance of a specific tensor operator with configuration x. We use trillion floating-point operations per second (TFLOPS) as the measurement in this work. Configuration x is an ordered μ-tuple $(x_1, ..., x_\mu)$ and each component $x_i \in \mathcal{X}_i$ corresponds to a hyperparameter of a device code, so the entire search space $\mathbb{X}$ is the Cartesian product of all component feasible sets $\prod_{i=1}^{\mu} \mathcal{X}_i$. Our aim is to find the optimal configuration $x^\star \in \mathbb{X}$ that corresponds to the maximum TFLOPS.

A topological structure over each $\mathcal{X}_i$ can be introduced by defining an undirected graph $G = (V, E)$, where the set of vertices V is $\mathcal{X}_i$, and the set of edges $E = \{\{u, v\} | u, v \in V,\ u \neq v,\ g_V(u, v) = 1\}$. Here $g_V(u, v)$ is an adjacency function mapping from $V \times V$ to $\{0, 1\}$. $g_V(u, v) = 1$ represents vertices u and v are adjacent, otherwise u and v are not adjacent. In this way, one can introduce a topological structure over V by defining an adjacency function $g_V(u, v)$ according to prior assumptions on V. For example, it is intuitive to treat u and v as adjacent if similar performance can be expected when changing from u to v. Search process can benefit from the topological structures introduced this way by obtaining information about neighborhood of samples, like the performance of configurations in the neighborhood of a poor performance configuration are probably poor as well, so that better sample-efficiency could be achieved.

Different tensor operators may have different types of hyperparameters and corresponding feasible sets. In the rest part of this section, we will discuss four kinds of hyperparameters used in this work, and construct topological structures for them. It should be noted that, besides them, one can easily introduce other types of hyperparameters and construct corresponding topological structures based on concrete demands in a similar way.



(a) Factorization parameter with $C = 8$ and $\nu = 3$.

(b) Permutation parameter with $n = 3$.

(c) Discrete parameter with feasible set $\{1, 2, 3, 4\}$.

(d) Categorical parameter with feasible set $\{a, b, c, d, e, f\}$.

Figure 1: A simple illustration of topological structures introduced over the search spaces of tensor operators.

First is the ν-tuple with a factorization constraint, $\mathcal{X}_i = \{(a_j)_{j=1}^{\nu} | \prod_{j=1}^{\nu} a_j = C,\ a_j \in \mathbb{N}_+\}$, where $\nu, C \in \mathbb{N}_+$ are constants depending on specific tensor operators. We will refer to this type of parameter as factorization parameter thereafter. The factorization parameter is required by a popular technique called matrix tiling for improving the cache hit rate of memory access. It iteratively splits computation into smaller tiles to adapt memory access patterns to a particular hardware. From the implementation perspective, it transforms a single loop into nested loops, where ν is the number of nested loops, C is the total loop length and a_j is the loop length of each nested loop. We define two factorizations of C are adjacent if one of them can be transformed to the other by moving z, a prime factor of C, from the n-th factor to the m-th factor, which is a basic transformation of the tiling scheme. This adjacency function can be formally written as $g(u, v) = 1$ if $\exists n, m, z$ such that $u_m = v_m \cdot z$ and $u_n = v_n \cdot z^{-1}$, and 0 otherwise, where $n, m = 1, ..., \nu$ are distinct indices. A simple example of the topology defined this way with $C = 8$ and $\nu = 3$ is illustrated in Figure 1a.

The second type is the permutation parameter, $\mathcal{X}_i = \mathcal{M}!$, where $\mathcal{M}$ is a set with n distinct elements and $\mathcal{M}!$ represents the symmetric group over $\mathcal{M}$. The order of nested loops in device code can be modeled by this type of parameter, where n is the number of nested loops and each element in the feasible set corresponds to a particular order of nested loops. We define two permutations of $\mathcal{M}$ are adjacent if one of them can be transformed to the other by a two-cycle permutation, which is a basic transformation of the order. This adjacency function can be formally written as $g(u, v) = 1$ if there exists a two-cycle permutation σ of $\mathcal{M}$ such that

$u = \sigma v$, and 0 otherwise. Figure 1b shows the topology defined this way when $n = 3$.

The third type is the discrete parameter, $\mathcal{X}_i = \{a_j | j = 1, ..., J \text{ and } J \in \mathbb{N}_+,\ a_j \in \mathbb{R}\}$, in which there are finite numbers. The maximum step of loop unrolling is an example of discrete type parameter. There is a natural adjacency among discrete parameters since they have well-defined comparability. This natural adjacency function can be formally written as $g(u, v) = 1$ if $\nexists w \in V$ such that $(w - u) \cdot (w - v) < 0$, and 0 otherwise. A simple example of the topology defined this way with $\mathcal{X}_i = \{1, 2, 3, 4\}$ is illustrated in Figure 1c.

The last type is the categorical parameter, $\mathcal{X}_i = \{a_j | j = 1, ..., J \text{ and } J \in \mathbb{N}_+\}$, in which there are finite elements that can be any entity. The choices like whether to unroll a loop and which thread axis to dispatch computation are examples of categorical type parameter. Unlike discrete parameters, there is no natural adjacency among categorical parameters, so all elements in the feasible set of categorical parameter are treated as adjacent, which can be formally written as $g(u, v) = 1$ for all $u, v \in V$ and $u \neq v$, and 0 otherwise. A simple example of the topology defined this way with $\mathcal{X}_i = \{a, b, c, d, e, f\}$ is illustrated in Figure 1d.

4 Methodology

4.1 Evolutionary Algorithm

EA is a kind of stochastic derivative-free optimization methods, which can be used to solve problems defined by Equation 1. EA imitates the natural selection in the evolution process of biological species to find the optimal configuration of an objective function. Evolutionary concepts are translated into algorithmic operations, i.e., selection, recombination, and mutation (Kramer 2016), which significantly influence the effectiveness and efficiency of EA.

To efficiently search the best configuration of a tensor operator, OpEvo leverages topological structures defined in Section 3 with an evolutionary framework. In specific, OpEvo evolves a population of configurations, which are also called individuals in EA terminology. The TFLOPS of executing tensor operators on a target hardware is a measure of the individuals' quality or fitness. At each evolutionary step, we select top ranked individuals to be parents based on their fitnesses, and then recombine and mutate them to generate new individuals or children. After evaluation, children are added to the population to be candidates of new parents at the next evolutionary step. This iteration will repeat until some termination criteria are met.

In the rest of this section, we will describe the selection, recombination and mutation operations of OpEvo in detail and illustrate how OpEvo leverages the topological structures and why OpEvo can outperform previous arts in this way.

4.2 Selection and Recombination

Suppose we already have a list of individuals which are ranked by their fitnesses. Top-λ ranked individuals are chosen to be parents, where $\lambda \in \mathbb{N}_+$ governs the diversity in evolutionary process. Evolution with large λ tends to get rid of suboptimum but sacrifices data efficiency, while one with small λ is easier to converge but suffers from suboptimum.

A child will be generated by recombining these selected parents in a stochastic way. Specifically, we sample below categorical distribution with λ categories μ times to decide which parents each parameter of a child should inherit from.

$$P(x_i = x_i^j) = \frac{f(x^j)}{\sum_{k=1}^{\lambda} f(x^k)}, \qquad (2)$$
$$\text{for} \quad i = 1, ..., \mu, \quad j = 1, ..., \lambda,$$

where μ is the number of parameters in a configuration, superscripts represent different parents, and subscripts represent different parameters in a configuration. x_i is the i-th parameter of generated child x.

It is worthwhile to mention that many SOTA methods suffer invalid configurations in the search spaces, which is inevitable since the constraints in search spaces are usually black-box as well. OpEvo can mitigate this problem by assigning zero fitnesses to the invalid configurations so that their characters have no chance to be inherited. In this way, invalid configurations will have less and less probability to be sampled during evolution.

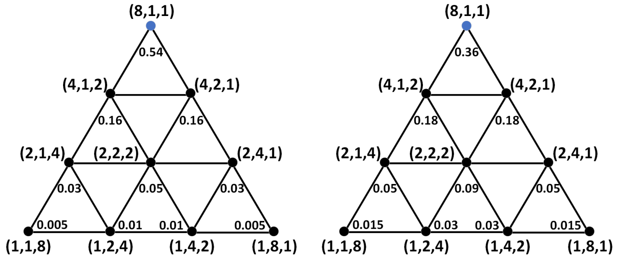
4.3 Mutation

OpEvo mutates each parameter x_i of each child by sampling a topology-aware probability distribution over corresponding feasible set $\mathcal{X}_i$. Given a topology over $\mathcal{X}_i$ and current vertex, such topology-ware probability distribution can be constructed by a random walk-like process. The transition probability from vertex u to an adjacent vertex v is

$$P_{uv} = \frac{q}{|S(u)|}, v \in S(u), \qquad (3)$$

where $q \in (0, 1)$ is the mutation rate which trade-offs the exploration and exploitation. OpEvo tends to exploration as q approaches 1, while tends to exploitation as q approaches 0. $S(u) = \{v | g(u, v) = 1\}$ is the set of all vertices adjacent to u, and $|S(u)|$ denotes the cardinality of set $S(u)$. The major difference between the "random walk" defined by Equation 3 and the regular random walk is that the summation of transition probability over all adjacent vertices is q rather than 1, so the "random walk" we introduced is not a Markov chain since there is a probability of $1 - q$ to stop walking. In this way, given a current vertex $u \in \mathcal{X}_i$, the topology-aware probability distribution $P_u(v)$ for all $v \in \mathcal{X}_i$ could be defined as the probability of walking started from u and stopped at v. We will refer to the distribution defined this way as q-random walk distribution thereafter. Appendix A formally proved that the q-random walk distribution is a valid probability distribution over $\mathcal{X}_i$.

For revealing the intuition behind q-random walk distribution, two q-random walk distributions over the feasible set of factorization parameter with $C = 8$ and $\nu = 3$ are illustrated in Figure 2. They start from the same vertex (the blue vertex) but mutate with different q. It could be easily observed that the vertices with smaller distance to the start vertex have higher probability to be sampled, which ensures a good trade-off between exploration and exploitation. Further, the



(a) $q = 0.5$

(b) $q = 0.7$

Figure 2: Two q-random walk distributions with different q.

distribution with a larger q has a wider spread than one with a smaller q, because larger q encourages more jumps in the q-random walk process. Considering the asymptotic case of $q = 1$, the q-random walk degenerates into a regular random walk on an undirected graph, which keeps jumping forever and eventually traverses all vertices on the graph, while in the case of $q = 0$, the q-random walk vanishes and no mutation acts on parameter x_i. Thus, q is a hyperparameter for OpEvo to trade off exploitation and exploration.

Considering a regular random walk on an undirected graph, i.e. $q = 1$, the probability of visiting a vertex in the graph is determined by the graph topology when the Markov chain induced by the random walk is converged. That's why random walk can be used for embedding graphs in many works (Perozzi, Al-Rfou, and Skiena 2014). q-random walk distribution also inherits this topology-aware nature. Observing vertices with the same distance to the start vertex in Figure 2, the vertices with more complex neighborhood have larger probability. For example, vertices $(2, 1, 4)$, $(2, 2, 2)$ and $(2, 4, 1)$ have the same distance to start vertex $(8, 1, 1)$, but vertex $(2, 2, 2)$ has larger probability since it has larger degree. This property of q-random walk distribution helps explore search spaces efficiently, because sampling vertices with more complex neighborhood will provide us more knowledge about objective functions.

4.4 Summary

The OpEvo algorithm is summarized in Algorithm 1. At first, λ configurations are randomly generated and evaluated to initialize a priority queue $\mathcal{Q}$ ordered by decreasing fitness. Next, taking top λ configurations from $\mathcal{Q}$ as parents and recombining them to generate ρ children according to Section 4.2. Then, each child is mutated based on Section 4.3. Note that the same configuration will not be sampled twice in the whole process of OpEvo, since the noise of TFLOPS of executing a tensor operator on a hardware is relatively small and data efficiency can benefit from non-duplicated samples. As a result, when a mutated child has already in $\mathcal{Q}$, we will mutate the child again until it is not already sampled. Finally, the fitnesses of ρ children are evaluated on target hardware, and enqueued into $\mathcal{Q}$. This iteration will repeat until some termination criteria are met.

Algorithm 1 OpEvo

```
Input: all component feasible sets $\mathcal{X}_i, i = 1, .., \mu$, parents size $\lambda$, offspring size $\rho$, mutation rate $q$
Output: optimal configuration $x^\star$
randomly generate $\lambda$ configurations $\{x^j\}_{j=1}^{\lambda}$
evaluate $\{x^j\}_{j=1}^{\lambda}$ to get associated fitness, and enqueue $\{x^j, f(x^j)\}_{j=1}^{\lambda}$ into a priority queue $\mathcal{Q}$
repeat
    select $\lambda$ parents from $\mathcal{Q}$ and recombine them to generate $\rho$ children according to Section 4.2
    mutate $\rho$ children according to Section 4.3
    evaluate $\rho$ children on hardware, and enqueue $\{x^j, f(x^j)\}_{j=1}^{\rho}$ into $\mathcal{Q}$
until termination criterion is met
return the best configuration so far
```

5 Experiments

We now evaluate the empirical performance of the proposed method with three typical kinds of tensor operators, MatMul, BatchMatMul, 2D Convolution, and a classic CNN architecture AlexNet (Krizhevsky, Sutskever, and Hinton 2012) on both Nvidia (GTX 1080Ti) and AMD (MI50 GPU) platforms. All tensor operators in our experiments are described and generated with TVM framework, and then compiled and run with CUDA 10.0 or RCOM 2.9. Additionally, we compare OpEvo with three aforementioned SOTA methods, G-BFS, N-A2C and AutoTVM. In our experiments, OpEvo, G-BFS and N-A2C are implemented by ourselves with the framework of Neural Network Intelligence (NNI, *https://github.com/microsoft/nni/*), and AutoTVM is implemented by its authors in the TVM project (*https://github.com/apache/incubator-tvm*). All codes for OpEvo, G-BFS, N-A2C and our benchmarks are publicly available with the NNI project. Please refer to Appendix B for more details about the experiments and Appendix C for specific numbers about figures presented in this section.

5.1 MatMul

Three different MatMul operators are chosen from BERT (Devlin et al. 2018) to evaluate proposed method. The maximum performance obtained so far versus number of trials and wall-clock time which have been used is illustrated in Figure 3. For the upper two rows, the lines denote the averages of 5 runs, while the shaded areas indicate standard deviations. For the lower two rows, each line denotes a specific run. Different colors and line styles represent different algorithms. From the results, it can be easily concluded that the methods leveraging predefined topology, OpEvo, G-BFS and N-A2C, much outperform the general SMBO method, AutoTVM. G-BFS and N-A2C leverage the underlying topology by introducing a MDP, so just local topology is considered and leveraged to explore the configuration space, while OpEvo can consider the global topology thanks to the mutation operation based on the q-random walk distribution. Therefore, OpEvo performs better than G-BFS and N-A2C in most cases in terms of mean and variance of best

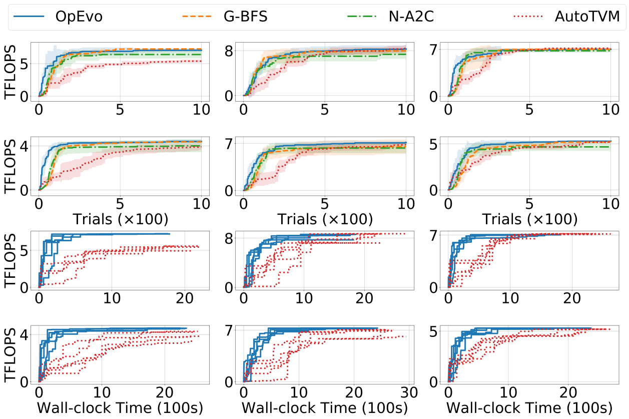


Figure 3: Algorithms comparison for three MatMul operators. The first and third rows are results on Nvidia platform, while the second and fourth rows are results on AMD platform. Three columns correspond to three operators MM1, MM2 and MM3 described in Appendix B.1 from left to right, respectively.

TFLOPS and data-efficiency. Further, as earlier mentioned, OpEvo is a lightweight model-free method, so the extra burden for training and optimizing surrogate models is avoided. It can be seen from Figure 3 that OpEvo can save around 30% and 10% wall-clock time when optimizing CUDA and ROCM operators, respectively. This is because the CUDA compilation speed is usually faster than ROCM, so the extra burden of tuning CUDA operators takes a larger share of the total wall-clock time.

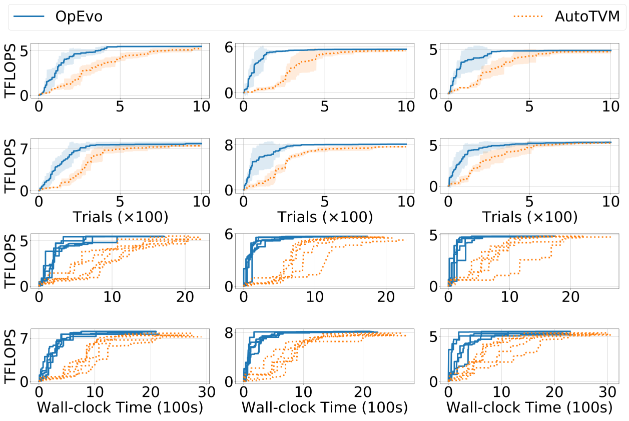


Figure 4: Algorithms comparison for three BatchMatMul operators. The first and third rows are results on Nvidia platform, while the second and forth rows are results on AMD platform. Three columns correspond to three operators BMM1, BMM2 and BMM3 described in Appendix B.2 from left to right, respectively.

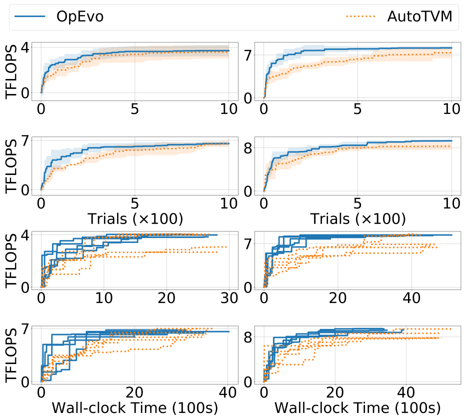


Figure 5: Algorithms comparison for two 2D Convolution operators. The first and third rows are results on Nvidia platform, while the second and forth rows are results on AMD platform. Two columns correspond to two operators C1 and C2 described in Appendix B.3 from left to right, respectively.

5.2 BatchMatMul

There are also three BatchMatMul operators selected from BERT for evaluation. All these operators have batch size 960, so G-BFS and N-A2C are not capable to optimize them because they can only deal with matrices with power of 2 rows and columns. The comparison between OpEvo and AutoTVM is shown in Figure 4. Compared with MatMul operators, BatchMatMul has an order of magnitude bigger search space since one more parameter needed to be optimized. Also, the generated BatchMatMul device code is more likely to overflow the device memory as tile size of BatchMatMul is bigger than that of MatMul, which leads to sparser performance measurement. Although these challenges exist, OpEvo performs still well thanks to the globally exploration mechanism. The variance of best performance even better than that of MatMul because of the sparsity.

5.3 2D Convolution

Two 2D convolution operators are chosen from AlexNet for evaluation. They have more complex search spaces and thus harder to model compared with tensor operators discussed before, since, besides factorization parameter, discrete and categorical parameters are also involved. As a result, G-BFS and N-A2C are not capable to tune them. Figure 5 shows the comparison between OpEvo and AutoTVM. Although XGBoost is a tree boosting model which is relatively friendly to discrete and categorical parameters, AutoTVM still performs worse than OpEvo, because EA inherently supports complex search space and OpEvo further improves sample-efficiency by leveraging predefined topology. We note that the time-saving effect of OpEvo is not significant in the 2D convolution cases, because compiling and executing convolution operators are much more time-consuming than MatMul and BatchMatMul Operators and thus dominate the total tuning time.

5.4 End-to-end Evaluation

A classic CNN architecture, AlexNet, is used to evaluate the end-to-end performance of the proposed method, where there are 26 different kinds of tensor operators covering the most commonly used types. Figure 6 shows the comparison between OpEvo and AutoTVM in terms of inference time, form which it can be easily concluded that OpEvo is more data-efficient than AutoTVM on both NVIDIA and AMD platforms. For OpEvo, the end-to-end inference time rapidly deceases and reaches the minimum at around 200 trials, while AutoTVM needs at least 400 trails to reach the same performance.

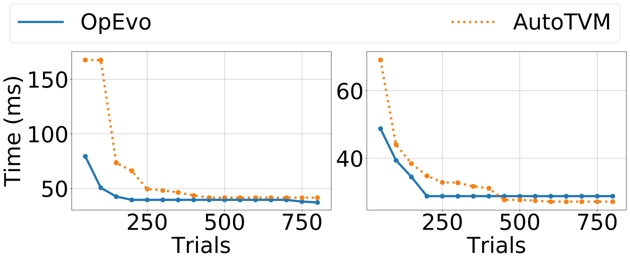


Figure 6: Algorithms comparison in terms of end-to-end inference time. The left figure is the result on Nvidia platform, while the right one is the result on AMD platform.

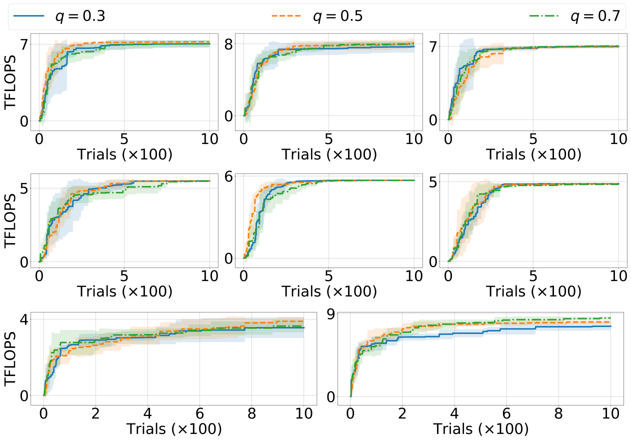


Figure 7: The effect of mutation rate q on OpEvo.

5.5 Hyperparameter Sensitivity

As earlier mentioned, OpEvo has two important hyperparameters, the mutation rate q which trade-offs the exploration and exploitation and the parent size λ which governs the diversity in the evolutionary process. In this part, we evaluate OpEvo with different q and λ for better understanding of each introduced technique and the hyperparameter sensitivity. From left to right, the first rows of Figure 7 and 8 correspond to MM1, MM2 and MM3, the second rows correspond to BMM1, BMM2 and BMM3, and the third rows correspond to C1 and C2.

It can be concluded from the Figure 7 and 8 that OpEvo is quite stable with the choice of q and λ in most cases. The effect of q is only visible in the 2D convolution cases, where insufficient exploration leads to suboptima and large variance. As for λ, the influences are only considerable in the cases of BMM2 and C1, where large λ results in a significant reduction of sample-efficiency while small λ results in suboptima and large variance due to insufficient diversity in the evolutionary process.

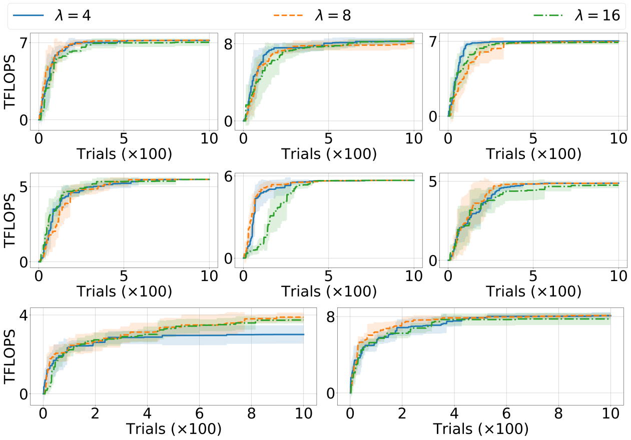


Figure 8: The effect of parent size λ on OpEvo.

6 Conclusion

In this paper, we proposed OpEvo, a novel evolutionary method which can efficiently optimize tensor operators. We constructed topological structures for tensor operators and introduced a topology-aware mutation operation based on q-random walk distribution, so that OpEvo can leverage the constructed topological structures to guide exploration. Empirical results show that OpEvo outperforms SOTA methods in terms of best FLOPS, best FLOPS variance, sample and time-efficiency. Please note that all experiments in this work are done with 8 CPU threads for compiling and a single GPU card for executing. The time-saving effect of OpEvo will be more significant if more CPU threads and GPU cards are used. Further, the analysis of hyperparameter sensitivity illustrated the robustness of OpEvo. This work also demonstrated that good leverage of proper prior assumptions on objective functions is the key of sample-efficiency regardless of model-based or model-free methods. Even EA can beat SMBO in terms of sample-efficiency as long as proper prior assumptions are effectively leveraged. Please note the proposed method cannot only be used to optimize tensor operators, but can also be generally applicable to any other combinatorial search spaces with underlying topological structures. Since the performance of OpEvo highly depends on the quality of constructed topology, it is particularly suitable for the cases where abundant human knowledge exists but there is lack of methods to leverage them.

Acknowledgements

We would like to thank Lidong Zhou and Jing Bai for the inspiring discussions with them, all anonymous reviewers for their insightful comments, and all contributors of the NNI project for the powerful and user-friendly tools they created.

References

Ahn, B. H.; Pilligundla, P.; Yazdanbakhsh, A.; and Esmaeilzadeh, H. 2020. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743* .

Bäck, T.; and Schwefel, H.-P. 1993. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation* 1(1): 1–23.

Bergstra, J. S.; Bardenet, R.; Bengio, Y.; and Kégl, B. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, 2546–2554.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* .

Chen, T.; and Guestrin, C. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785–794. ACM.

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; Guestrin, C.; and Krishnamurthy, A. 2018a. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–594. Carlsbad, CA: USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chen, T.; Zheng, L.; Yan, E.; Jiang, Z.; Moreau, T.; Ceze, L.; Guestrin, C.; and Krishnamurthy, A. 2018b. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, 3389–3400.

Cobbe, K.; Hesse, C.; Hilton, J.; and Schulman, J. 2019a. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588* .

Cobbe, K.; Klimov, O.; Hesse, C.; Kim, T.; and Schulman, J. 2019b. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning*, 1282–1289. PMLR.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, 248–255. Ieee.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* .

Elsken, T.; Metzen, J. H.; and Hutter, F. 2018. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377* .

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290* .

Hernández-Lobato, J. M.; Hoffman, M. W.; and Ghahramani, Z. 2014. Predictive entropy search for efficient global optimization of black-box functions. In *Advances in neural information processing systems*, 918–926.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, 507–523. Springer.

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

Kramer, O. 2016. *Machine learning for evolution strategies*, volume 20. Springer.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 701–710.

Srinivas, N.; Krause, A.; Kakade, S. M.; and Seeger, M. 2009. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995* .

Tai, K. S.; Socher, R.; and Manning, C. D. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* .

Wang, G.; Lin, Y.; and Yi, W. 2010. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, 344–350. IEEE.

Wang, Z.; and Jegelka, S. 2017. Max-value entropy search for efficient Bayesian optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 3627–3635. JMLR. org.

Youssef, H.; Sait, S. M.; and Adiche, H. 2001. Evolutionary algorithms, simulated annealing and tabu search: a comparative study. *Engineering Applications of Artificial Intelligence* 14(2): 167–181.

Zhang, H.; Cheng, X.; Zang, H.; and Park, D. H. 2019. Compiler-Level Matrix Multiplication Optimization for Deep Learning. *arXiv preprint arXiv:1909.10616* .

Zheng, L.; Jia, C.; Sun, M.; Wu, Z.; Yu, C. H.; Haj-Ali, A.; Wang, Y.; Yang, J.; Zhuo, D.; Sen, K.; et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. *arXiv preprint arXiv:2006.06762* .