Deep Neural Network Hardware Deployment Optimization via Advanced Active Learning

Qi Sun, Chen Bai, Hao Geng, Bei Yu
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{qsun,cbai,hgeng,byu}@cse.cuhk.edu.hk

***Abstract*—Recent years have witnessed the great successes of deep neural network (DNN) models while deploying DNN models on hardware platforms is still challenging and widely discussed. Some works proposed dedicatedly designed accelerators for some specific DNN models, while some others proposed general-purpose deployment frameworks that can optimize the hardware configurations on various hardware platforms automatically. However, the extremely large design space and the very time-consuming on-chip tests bring great challenges to the hardware configuration optimization process. In this paper, to optimize the hardware deployment, we propose an advanced active learning framework which is composed of batch transductive experiment design (BTED) and Bootstrap-guided adaptive optimization (BAO). The BTED method generates a diverse initial configuration set filled with representative configurations. Based on the Bootstrap method and adaptive sampling, the BAO method guides the selection of hardware configurations during the searching process. To the best of our knowledge, these two methods are both introduced into general DNN deployment frameworks for the first time. We embed our advanced framework into AutoTVM, and the experimental results show that our methods reduce the model inference latency by up to** 28.08% **and decrease the variance of inference latency by up to** 92.74%**.**

I. Introduction

Deep neural networks (DNNs) have achieved unprecedented successes on a wide range of applications, such as objective detection, image classification, natural language processing, and even design for manufacturing [1]–[4]. In DNN models, there are some common layers such as convolutional layers and fully-connected layers. In these layers, we conduct multiplication and addition operations on input features and weights to generate output features as results. These features and weights usually have large scales and therefore the number of operations and memory consumptions are very large.

To deploy DNN models on hardwares efficiently, great efforts have been made recently and various platforms are considered, *e.g.*, ASIC [5], FPGA [6], memristor [7], mobile devices [8], and general-purposed GPU [9]. Some researchers propose to build complicated formulations or analysis models [10]–[12] to characterize the hardware usage and on-chip model scheduling, *i.e.*, deployment configurations. In these works, hardware architecture and model structure are analyzed in detail, and then a slew of candidate deployment configurations are enumerated. Typically, all of the configurations are estimated by the formulations or models to find the best one. These works are highly dependent on the accuracy of the analytical models. They may suffer from inflexibility and are difficult to be extended because hardware and analysis models are highly coupled. To some extent, they are regarded as hardware-and-model-specific solutions.

To alleviate the inflexibility issues, developing general frameworks has been a new trend for hardware deployments. Halide [13] and TVM [14] decouple algorithms from schedules and treat the hardware architectures as black boxes in their optimization flow. The design flow of the general framework is shown in Fig. 1. In the general frameworks, a DNN model is represented as a computational graph, and operators (layers) in it are represented as nodes. Some

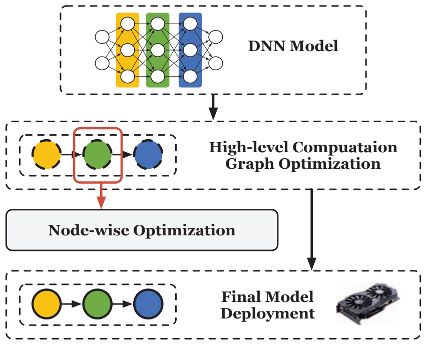

Fig. 1 The general framework of DNN hardware deployment optimization. For each node, we can conduct the node-wise optimization to find the best deployment configuration. Our advanced active learning framework is proposed as the node-wise optimization algorithm.

graph level tuning strategies are adopted to optimize the graph. For example, *operator fusion* combines multiple operators into a single kernel (*i.e.*, node). Subsequently, node-wise optimization is applied to find the best configuration for each node. The final model deployment solution is the combination of solutions to all the nodes. For each node, all of the deployment configurations are enumerated and encoded as design points to construct a large design space. For simplicity, in this paper, we do not distinguish between layer and node. Traditionally, the active learning algorithm which consists of an initialization stage and an iterative optimization stage is utilized here as the solution searching strategy. Firstly, some initial points are sampled from the original solution space to initialize an evaluation function. Then the optimization process is performed iteratively until satisfying the stopping criteria. In each optimization iteration, a configuration point is selected according to the evaluation function and then deployed on hardware to get real performance to update the evaluation function. Some machine learning or deep learning algorithms can be applied in the active learning flow. For example, XGBoost regression [15] can be used as the evaluation function. Simulated annealing (SA) [16] can be used as the iterative optimization algorithm. Transfer learning [17] can accelerate the iterative optimization process by utilizing history deployment information. AutoTVM [18], which integrates the above three methods, is an automatic optimization framework in TVM [14] and achieves state-of-the-art performance. CHAMELEON [19] leverages reinforcement learning to improve AutoTVM. However, it is too difficult to implement and train the reinforcement learning models. Therefore limited numbers of experiments are conducted in [19] and results are not outstanding.

With the help of these general frameworks and active learning algorithms, researchers can concentrate on the optimizations of backend code translations so that more hardware characteristics are considered to generate elaborately designed codes. For example, based on TVM, HeteroCL [20] produces highly efficient FPGA spatial architectures by incorporating systolic arrays and stencils with dataflow architectures. FlexTensor [21] which is also based on TVM promotes the DNN deployments on heterogeneous systems composed of CPU, GPU, and FPGA.

However, the extremely large design space and expensive time costs of on-chip deployments bring great challenges to these frameworks and make them powerless. For example, in TVM, the first optimization node in VGG-16 [22] has approximately 0.2 billion configuration points. Besides, it is difficult to reproduce advanced deep learning algorithms which are highly dependent on large amounts of training data and computation powers. It is also unfriendly to fast commercial developments that have strict requirements on the development cycles and algorithm stability. Overall, in the traditional algorithms, there are three unsolved problems:

1) Initialization in lack of rich data information.
2) Un-scalability of the optimization process.
3) Inaccuracy of evaluation functions.

To solve these problems, we propose a general advanced active learning framework. Our contributions are summarized as follows:

- An advanced active learning framework is proposed, which is composed of batch transductive experimental design (BTED) and Bootstrap-guided adaptive optimization (BAO) methods.
- The BTED method generates a diverse initial configuration set filled with representative configurations in a batch fashion. BTED can solve the initialization and scalability problems.
- Based on Bootstrap method and adaptive sampling, the BAO method guides the selection of better deployment configurations more accurately in the iterative optimization process. BAO solves the scalability and accuracy problems.
- Our framework is embedded into TVM and the results show the uplifting improvements for DNN model deployments on general hardware.

The rest of this paper is organized as follows. Section II introduces the problem to be addressed and some preliminaries. Section III explains our proposed batch transductive experimental design (BTED) and Bootstrap-guided adaptive optimization (BAO) in detail. Section IV summarizes our advanced active learning framework. Section V demonstrates the experiments and results. Finally, we conclude this paper in Section VI.

II. Preliminaries

A. DNN Hardware Deployment

To deploy DNN models on hardware, typically, features and weights in each layer are partitioned into some tensors, and consequently, the computation task of this layer is split into some smaller tasks. Hardware resources are carefully allocated to conduct computations of these small tasks. Upon the partitions of data and the allocations of hardware resources, elegant scheduling strategies are necessary to organize these small tasks to maximize data reuse, reduce communication costs, increase system parallelism, and achieve optimal deployment performance. During this process, it is challenging to find a compatible scheduling solution which takes a good balance between resource allocations and data partitions.

Different DNN accelerator platforms have various structures while the design ideas are similar. CUDA GPU programming architecture [23] in Fig. 2 is taken as an example to illustrate this. The device is divided into some grids. Each grid possesses several blocks and a shared global memory. Each block consists of some threads, shared memory, and some local registers. Each thread can be regarded as a basic computation unit which can perform a certain amount of calculations. The inter-thread communication relies on the shared memory and each local register is exclusively occupied by a thread. The grids can compute the tensors in parallel and they can be further decomposed and assigned to the blocks and threads. To deploy a model on CUDA GPU efficiently, there are some crucial settings to be determined, *e.g.*, thread binding, thread cooperation, memory locality, tensorization, *etc.*.

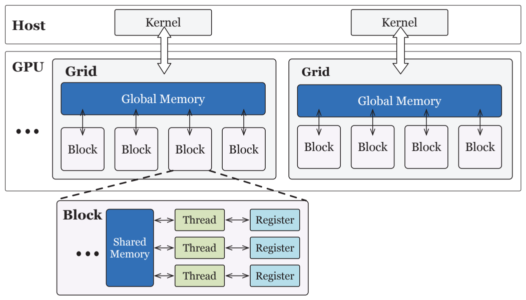

Fig. 2 The Nvidia CUDA programming Architecture.

B. Active Learning

The active learning algorithm, also termed optimal experimental design, is an iterative optimization mechanism [24]. Typically, it contains two stages: initialization, and iterative optimization. Firstly, it samples some data points from the whole dataset to initialize an evaluation function. Secondly, to find data points with better performance from the dataset, it iteratively interacts with the environment. In each iteration, it selects a new point from the whole dataset according to the evaluation function and the searching strategy, and then updates the evaluation functions accordingly. Usually, the searching strategy relies on the machine learning model to evaluate the qualities of new points. This process continues until the stopping criteria are reached.

C. Bootstrap Method

Bootstrap method is to approximate a population distribution by a sample distribution [25]. Assume that original data set is $\mathcal{D}$, and its real distribution is $\varphi(\mathcal{D})$. Limited by sampling techniques or influenced by random noises, the sampled data set $\mathcal{X} \in \mathcal{D}$ is inefficient to characterize $\varphi(\mathcal{D})$ by $\varphi(\mathcal{X})$. Machine learning models built on these data are therefore inaccurate and unstable. The Bootstrap method is proposed to help solve these problems [25], [26]. Generally, the Bootstrap method takes $\mathcal{X}$ as input. A batch of data sets with cardinalities equal to $\mathcal{X}$ is re-sampled from $\mathcal{X}$ uniformly. Assume that there are Γ sets sampled from the $\mathcal{X}$, denoted as $\tilde{\mathcal{X}}_t$, and their corresponding distributions are $\varphi(\tilde{\mathcal{X}}_t)$, with $t \in \{1, \dots, \Gamma\}$. The original distribution $\varphi(\mathcal{D})$ can be better approximated by $\{\varphi(\tilde{\mathcal{X}}_1), \varphi(\tilde{\mathcal{X}}_2), \cdots, \varphi(\tilde{\mathcal{X}}_\Gamma)\}$. The probability of an item in $\mathcal{X}$ is picked at least once is $1 - (1 - 1/\Gamma)^\Gamma$, which for large Γ becomes $1 - e^{-1} \approx 0.632$. Hence, the number of unique data points in a Bootstrap set is $0.632 \times \Gamma$ on average.

Bootstrap, also named bagging method, is an ensembling learning method. Bagging method can reduce model variance and increase robustness, which is especially useful when the solution space is very large, *e.g.*, the DNN hardware deployment problem. Another famous ensembling method is boosting, which aggregates several weak models to form a better one. In comparison, the boosting method follows a voting mechanism, which can reduce the bias among several weak models.

D. Problem Formulation

Definition 1 (Deployment Configuration)**.** *All of the deployment settings (e.g., thread binding, thread cooperation, etc.) to be determined are encoded as the attributes of a feature vector which is termed as deployment configuration. The feature vector of a deployment configuration is denoted as $\boldsymbol{x}$.*

Definition 2 (GFLOPS)**.** *Giga floating operations per second (GFLOPS) measures the number of floating-point operations conducted by the hardware per second.*

Definition 3 (Latency)**.** *Latency computes end-to-end model inference time and intuitively reflects the performance of model deployment.*

With the above definitions, our problem can be formulated.

Problem 1 (DNN Hardware Deployment Optimization)**.** *For each layer in a DNN model, given a search space $\mathcal{D}$ where each deployment configuration is regarded as a point, the objective of the hardware deployment optimization is to find the best deployment configuration $\boldsymbol{x}_* \in \mathcal{D}$ which maximizes GFLOPS. The deployment for the whole model is the combination of configurations for all of the layers, which is measured by latency.*

III. Advanced Active Learning Framework

In this section, the proposed advanced active learning framework is explained in detail.

A. Batch Transductive Experimental Design

In the initialization stage of active learning, as mentioned above, two important problems are unsolved, *i.e.*, initialization in lack of rich information and un-scalability.

Limited by the computational resources and experimental costs, Researchers tend to sample as few initial configurations as possible while building an evaluation function. Meanwhile, to guarantee the performance of evaluation functions, the sampled configurations should contain rich data information. From the perspective of data distribution, containing rich information means that the initial configurations should have high diversities and scatter across the input design space [27]. In other words, the sampled configurations should be as far as possible from each other in the input design space. A naive solution is to randomly sample some configurations from the design space. Further, some researchers have proposed to use SVM or similar methods in which all the configurations are taken into computations to learn the scattered data [28]–[30]. Inspired by [27], in this paper, we propose to use transductive experimental design (TED) method , as shown in Algorithm 1. Given the input un-sampled set $\mathcal{V}$, we will sample a subset $\mathcal{X}$ from $\mathcal{V}$ which maximizes the intra-set diversity. $\boldsymbol{K}_{\mathcal{V}\mathcal{V}} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the distance matrix of configurations in $\mathcal{V}$. $k(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{K}_{\mathcal{V}\mathcal{V}}$ is computed as Euclidean distance, with $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathcal{V}$. According to Algorithm 1, the configurations that are the most contributive to initialization are sampled. In summary, it is an easy-to-implement method which has low computation workloads.

The huge amounts of configurations in DNN deployment problems make it difficult to find representative initial configurations. To tackle this efficiently, we further propose batch transductive experimental design (BTED) by using batch mechanism and randomness, as shown in Algorithm 2. We will randomly sample a batch of sets from the original set and then conduct Algorithm 1 on these sampled sets. The final output $\mathcal{X}$ of Algorithm 2 is the initialization set in our advanced active learning framework. By introducing randomness into TED, *i.e.*, line 2 in Algorithm 2, we can reduce the computation complexity and thus improve the scalability. Besides, our batch method can stimulate the parallelism in our framework, and enlarge the random space which is used to generate the initial set. By using the batch method with a fixed number of initial configurations, our framework can compute on as many configurations as possible without delaying the system.

Algorithm 1 Transductive Experimental Design – **TED**$(\mathcal{V}, \mu, m)$

```
Require: ($\mathcal{V}$, $\mu$, $m$), where $\mathcal{V}$ is the un-sampled configuration set, $\mu$ is the normalization coefficient, $m$ is the number of configurations we will sample.
Ensure: Newly sampled configuration set $\mathcal{X}$.
1: $\boldsymbol{K} \leftarrow \boldsymbol{K}_{\mathcal{V}\mathcal{V}}$, $\mathcal{X} \leftarrow \emptyset$;
2: for $i = 1 \to m$ do
3:     $\boldsymbol{x} = \arg\max_{\boldsymbol{v}\in\mathcal{V}} \frac{\|\boldsymbol{K}_{\boldsymbol{v}}\|^2}{k(\boldsymbol{v},\boldsymbol{v})+\mu}$;   ▷ $\boldsymbol{K}_{\boldsymbol{v}}$ and $k(\boldsymbol{v},\boldsymbol{v})$ are $\boldsymbol{v}$'s corresponding column and diagonal entry in $\boldsymbol{K}$.
4:     $\mathcal{X} = \mathcal{X} \cup \boldsymbol{x}$;
5:     $\boldsymbol{K} = \boldsymbol{K} - \frac{\boldsymbol{K}_{\boldsymbol{x}}\boldsymbol{K}_{\boldsymbol{x}}^\top}{(k(\boldsymbol{x},\boldsymbol{x})+\mu)}$;
6: end for
7: return Newly sampled configuration set $\mathcal{X}$;
```

Algorithm 2 Batch Transductive Experimental Design – **BTED**$(\mathcal{V}, \mu, M, m, B)$

```
Require: ($\mathcal{V}$, $\mu$, $M$, $m$, $B$), where $\mathcal{V}$ is the un-sampled configuration set, $\mu$ is the normalization coefficient, $B$ is the batch size, $M$ is the number of randomly sampled points and $m$ is the number of points to be sampled as the initial set.
Ensure: Newly sampled configuration set $\mathcal{X}$.
1: for $b = 1 \to B$ do
2:     Randomly sample a set $\mathcal{V}_b$ from $\mathcal{V}$, with $|\mathcal{V}_b| = M$;
3:     $\tilde{\mathcal{X}}_b \leftarrow$ TED$(\mathcal{V}_b, \mu, m)$;   ▷ Algorithm 1
4: end for
5: Temporal union set $\tilde{\mathcal{X}}_U = \tilde{\mathcal{X}}_1 \cup \tilde{\mathcal{X}}_2 \cup \cdots \cup \tilde{\mathcal{X}}_B$;
6: $\mathcal{X} \leftarrow$ TED$(\tilde{\mathcal{X}}_U, \mu, m)$;   ▷ Algorithm 1
7: return Newly sampled configuration set $\mathcal{X}$;
```

B. Bootstrap-guided Adaptive Optimization

Except for the problems in the initialization stage, there still exist some crucial problems like inaccuracy and un-scalability in the iterative optimization stage. Firstly, the complexities of hardware characteristics and DNN models make the evaluation function hard to simulate the real environment accurately. The evaluation function would be misled by the already-sampled configurations and make wrong decisions when selecting new configurations. Besides, as mentioned before, the configuration space is usually extremely large, which means that in each optimization step, the searching range of the next sampled point is too large to be analyzed.

Bootstrap re-sampling has been proven to be an effective technique to correct and quantify optimization of model performance [26]. To improve the model accuracy, for the first time, we introduce the Bootstrap re-sampling technique into the DNN hardware deployment community. Firstly, we randomly re-sample a batch of sets from the already-sampled configuration set. Then, we build new evaluation functions for each of these re-sampled sets. The final evaluation function is built as the summation of the evaluation functions of these re-sampled sets. Assume that there are Γ re-sampled sets $\tilde{\mathcal{X}}_\gamma$ from $\mathcal{X}$, with $\gamma \in \{1, \dots, \Gamma\}$. Accordingly, Γ evaluation functions are built, denoted as $f_\gamma, \gamma \in \{1, \dots, \Gamma\}$. The next optimization configuration point $\boldsymbol{x}^*$ is the configuration that maximizes the summation of these Γ evaluation functions. $\boldsymbol{x}^*$ will be deployed on hardware to get real performance y^* for further usage. The pseudo-code is shown in Algorithm 3.

Algorithm 3 Bootstrap-guided Sampling – **BS**$(\mathcal{X}, \mathcal{Y}, \mathcal{C}, \Gamma)$

```
Require: $(\mathcal{X}, \mathcal{Y}, \mathcal{C}, \Gamma)$, where $\mathcal{X}$ is the already sampled configuration
    set and $\mathcal{Y}$ is its performance set, $\mathcal{C}$ is the current searching space,
    $\Gamma$ is the number of re-sampled sets.
Ensure: New configuration $\boldsymbol{x}^*$.
1: for $\gamma = 1 \to \Gamma$ do
2:     Randomly sample $\tilde{\mathcal{X}}_\gamma$ from $\mathcal{X}$, with $|\tilde{\mathcal{X}}_\gamma| = |\mathcal{X}|$;
3:     Get $\tilde{\mathcal{X}}_\gamma$'s performance set $\tilde{\mathcal{Y}}_\gamma$ from $\mathcal{Y}$;
4:     Build evaluation function $f_\gamma$, according to $\tilde{\mathcal{X}}_\gamma$ and $\tilde{\mathcal{Y}}_\gamma$;
5: end for
6: $\boldsymbol{x}^* \leftarrow \max_{\boldsymbol{x} \in \mathcal{C}} \sum_{\gamma=1}^{\Gamma} f_\gamma(\boldsymbol{x})$;
7: return $\boldsymbol{x}^*$;
```

To make the searching algorithm scalable, in each optimization step, instead of traversing the whole configuration space, we adjust the searching space adaptively. In the model deployment problem, an acceptable assumption is that if a configuration has good deployment performance, it is very likely that we can find better configurations in its neighborhood. The basic idea behind this assumption is that the value space is local smooth, or at least approximately smooth in a certain range. Therefore, to find a suitable searching space to reduce the searching costs and compensate for the loss when the results are not satisfying, we improve Algorithm 3 by using adaptive neighborhood adjustment strategy. If the relative improvement of the performance values between the previous two consecutive optimization steps is satisfying, *i.e.*, greater than a threshold η, we will keep the radius of the neighborhood as a constant $\mathcal{R}$. Otherwise, we will enlarge the searching space, *i.e.*, adopting a larger radius $\tau\mathcal{R}$ where $\tau > 1$ is a hyperparameter. The relative improvement is computed via Equation (1).

$$r_t = \left\lceil \frac{\mathrm{y}_{t-1}^* - \mathrm{y}_{t-2}^*}{\mathrm{y}_{t-1}^*} \right\rceil, \tag{1}$$

where y_{t-1}^* and y_{t-2}^* are the optimal performance values found in step $(t-1)$ and $(t-2)$, respectively. After determining the optimization configuration $\boldsymbol{x}_t^*$, we will deploy it to get performance y_t^* and add them into the already-sampled set. The pseudo-code of the Bootstrap-guided adaptive optimization is shown in Algorithm 4.

Our sampling algorithm is general enough to handle various types of evaluation function f_γ and therefore can be easily extended.

IV. Framework Summary

As shown in Fig. 1, a given DNN model to be deployed on hardware is usually represented as a computational graph in which each node represents a layer. Our advanced active learning framework is adopted to conduct node-wise optimization in the computational graph, to find the best deployment configurations.

The overall flow of our advanced active learning framework is visualized in Fig. 3. For each node, the input to our framework is its configuration space $\mathcal{D}$. BTED (Algorithm 1 & Algorithm 2) is used to generate initial set, which is aiming at tackle the scalability and

Algorithm 4 Bootstrap-guided Adaptive Optimization – **BAO**$(T, \mathcal{X}, \mathcal{Y}, \eta, \Gamma)$

```
Require: $(T, \mathcal{X}, \mathcal{Y}, \eta, \Gamma)$, where $T$ is number of optimization itera-
    tions, $\mathcal{X}$ is the already sampled initialization configuration set
    and $\mathcal{Y}$ is its performance set, $\eta$ is a threshold, and $\Gamma$ is the
    number of re-sampled sets.
Ensure: Updated $\mathcal{X}$ and $\mathcal{Y}$.
1: Select $x_0^*$ from $\mathcal{X}$ with the best performance value;
2: for $t = 1 \to T$ do
3:     $\mathcal{C}_t \leftarrow$ neighborhood of $\boldsymbol{x}_{t-1}^*$ with radius $\mathcal{R}$;
4:     if $t \geq 2$ then
5:         Compute $r_t$;                                      ▷ Equation (1)
6:         if $r_t < \eta$ then
7:             $\mathcal{C}_t \leftarrow$ neighborhood of $\boldsymbol{x}_{t-1}^*$ with radius $(\tau \cdot \mathcal{R})$;
8:         end if
9:     end if
10:    $\boldsymbol{x}_t^* \leftarrow \mathrm{BS}(\mathcal{X}, \mathcal{Y}, \mathcal{C}_t, \Gamma)$;                     ▷ Algorithm 3
11:    Deploy $\boldsymbol{x}_t^*$ on hardware to get GFLOPS $\mathrm{y}_t^*$;
12:    $\mathcal{X} = \mathcal{X} \cup \boldsymbol{x}_t^*$ and $\mathcal{Y} = \mathcal{Y} \cup \mathrm{y}_t^*$;
13: end for
14: return Updated $\mathcal{X}$ and $\mathcal{Y}$;
```

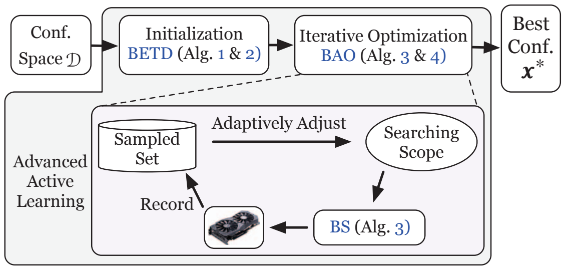
Conf. Space $\mathcal{D}$
Initialization BETD (Alg. 1 & 2)
Iterative Optimization BAO (Alg. 3 & 4)
Best Conf. $\boldsymbol{x}^*$
Advanced Active Learning
Sampled Set
Adaptively Adjust
Searching Scope
Record
BS (Alg. 3)

Fig. 3 Our advanced active learning framework.

informativeness problems mentioned in Section I. BAO (Algorithm 3 & Algorithm 4), consisting of Bootstrap-guided re-sampling and adaptive sampling, is used to guide the iterative optimization process and solves the accuracy and scalability problems. In one iteration of Algorithm 4, Algorithm 3 is invoked and the searching scope is adjusted adaptively. The framework interacts with the real hardware to get real GFLOPS values. The output $\boldsymbol{x}^*$ is the best deployment configuration with the highest GFLOPS value for this layer.

To the best of our knowledge, both BTED and BAO algorithms are used in general DNN deployment frameworks for the first time. Except for solving the problems mentioned in Section I, more importantly, our framework is independent of the specific forms of evaluation functions, thus making it compatible with various algorithms. The superior generality of the proposed framework keeps pace with the fast development of the community.

V. Experimental Results

In this section, we embed our advanced active learning framework into the most popular general DNN deployment framework TVM (state-of-the-art stable released version v0.6.1 [31] before the paper submission) to validate the performance. The hardware platform is Nvidia GeForce GTX 1080 Ti. Some public models which can be found in TVM tutorial are tested, including AlexNet [32], ResNet-18 [33], VGG-16 [22], MobileNet-v1 [34], and SqueezeNet-v1.1 [35]. In total, there are 58 nodes that need to be optimized in

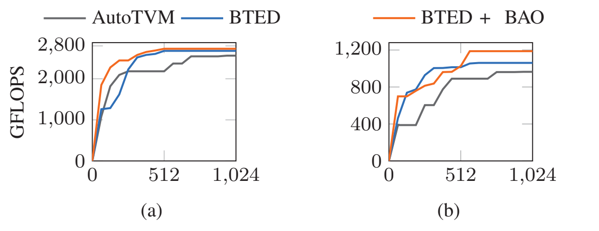

Fig. 4 Convergence trends of GFLOPs for the first 2 layers of MobileNet-v1, (a) the first layer, (b) the second layer.

these models. On average, each node has more than 50 million configuration points. The representative DNN layers widely used in both industries and academia are all covered in these models, including conventional convolutional layers, shortcut layers, multi-branch layers, fully connected layers, depth-wise convolutional layers, batch-normalization layers, and *etc.*.

A. Experimental Settings

AutoTVM [18], which integrates XGBoost, simulated annealing, transfer learning, and *etc.*, is the state-of-the-art academic optimization framework. In AutoTVM, by default, 64 points are sampled from the configuration space as the initialization set. Early stopping is adopted in the searching process as the stopping criterion and the stopping threshold is set as 400. For each layer in the DNN model, GFLOPS is used as the optimization objective while the latency of the whole model is reported as the final deployment performance metric, as mentioned in Section II-D.

The experiments conducted in this paper are as below:

- `AutoTVM`: The automatic optimization framework in TVM.
- `BTED`: Embed BTED initialization algorithm into AutoTVM.
- `BTED+BAO`: Embed our advanced active learning framework (BTED and BAO) into AutoTVM.

The input pair in Algorithm 2 is $(\mathcal{V} = \mathcal{D}, \mu = 0.1, M = 500, m = 64, B = 10)$, where $\mathcal{D}$ is the default configuration space generated by TVM. Each batch randomly samples 500 points from $\mathcal{D}$, and then 64 points are selected from each batch via Algorithm 1. Thus, the union set $\tilde{\mathcal{X}}_U$ contains 640 points, and finally 64 points are sampled from $\tilde{\mathcal{X}}_U$ as the initial set. In Algorithm 4, η is set as 0.05, Γ is 2, and τ is set as 1.5. The radius $\mathcal{R}$ is set as 3 which means that the Euclidean distance between points. For fairness, except for the aforementioned hyper-parameters, we follow the same experimental settings as AutoTVM. To avoid disturbances caused by hardware workload uncertainties, in each experiment trial, we run the deployed model 600 times. Therefore, in experiments, the average latency as well as the variance of these 600 tests are recorded. Further, to reduce randomness, each algorithm is performed 10 trials to obtain the corresponding configuration solutions for each DNN model, and the final results are the averages of these 10 trials.

B. Results and Discussions

Convergence. We take Fig. 4 as an example to compare the convergence trends of GFLOPS over the number of configurations. Even if there are millions of configurations, our method can outperform AutoTVM with a faster converge speed and a higher GFLOPS value.

GFLOPS and Workloads. Our method can achieve much better GFLOPS performance without sampling more configurations. Limited by paper length, not all results are plotted. Fig. 5 shows the number of sampled configurations *v.s.* GFLOPS values for all 19 layers in MobileNet-v1. Since different layers have diverse GFLOPS values, for clarity, all of the GFLOPS results are represented as ratios to the results of AutoTVM. As shown in Fig. 5(a), comparing to AutoTVM, BTED tends to sample more configurations, while (BTED + BAO) method samples roughly equal ones. On average, BTED and (BTED + BAO) improve the GFLOPS values by up to 36.74% and 47.94% respectively. The results reveal that, with occasionally sacrificing the cost of optimization workloads, BTED behaves much better than AutoTVM. With the help of the BAO, we can reduce the optimization workload without degrading performance.

Latency and Variance. The inference latencies of end-to-end models and corresponding variances are recorded in TABLE I. For convenience, the improvement ratios with respect to AutoTVM are computed. Our method can reduce the latency by up to 28.08% and decrease the variance by up to 92.74%, on MobileNet-v1. Averagely, our framework reduces inference latency and variance by 13.83% and 67.74% respectively on these representative models.

Discussions. The uplifting performance improvements of our advanced frameworks show that the three unsolved problems can be handled efficiently. The foreseeable development trend of DNN model deployment is that more and more hardware platforms will be developed and used. Therefore, the size of the deployment configuration space will increase continuously. Besides, huge amounts of newly proposed models would also enlarge the configuration space. From this perspective, our advanced framework would be more remarkable. In addition, it is believed that our framework can be integrated with more optimization methods, *e.g.*, deep learning algorithms. Our framework is also easy to be implemented, with high algorithm stability and low development workloads.

VI. Conclusion

In this paper, an advanced active learning framework composed of BTED and BAO has been proposed to improve the general DNN hardware deployment framework and solve three crucial problems. The BTED method can generate an initial set with rich information. The BAO method can help sample configurations, improve model accuracy, and adjust searching scope adaptively. Both of them have high scalability. We believe that our pioneering study will guide the community to solve some fundamental problems and further improve the DNN model deployments.

Acknowledgment

This work is partially supported by Tencent and SmartMore.

References

[1] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. NIPS*, 2015.
[2] X. Zhang, Y. Li, C. Hao, K. Rupnow, J. Xiong, W.-m. Hwu, and D. Chen, "SkyNet: A champion model for DAC-SDC on low power object detection," *arXiv preprint*, 2019.
[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint*, 2018.
[4] H. Geng, H. Yang, Y. Ma, J. Mitra, and B. Yu, "SRAF insertion via supervised dictionary learning," in *Proc. ASPDAC*, 2019.
[5] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. R. Pinckney, P. Raina *et al.*, "MAGNet: A modular accelerator generator for neural networks." in *Proc. ICCAD*, 2019.
[6] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Automatic compilation of diverse cnns onto high-performance fpga accelerators," *IEEE TCAD*, 2018.

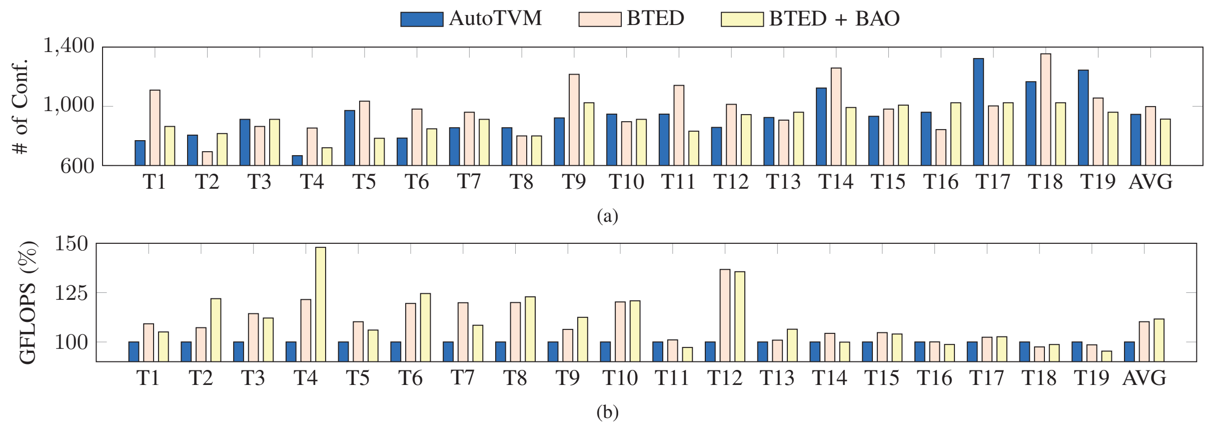

Fig. 5 The number of sampled configurations and GFLOPS values of MobileNet-v1. AVG represents the average results of the 19 tasks.

TABLE I Comparisons of End-to-end Model Inference Latency and Variance

Model	AutoTVM		BTED				BTED + BAO			
	Latency (ms)	Variance	Latency (ms)	Δ (%)	Variance	Δ (%)	Latency (ms)	Δ (%)	Variance	Δ (%)
`AlexNet`	1.3639	0.1738	1.3373	- 1.95	0.2246	+29.23	1.3304	**- 2.46**	0.0711	**-59.09**
`ResNet-18`	1.8323	0.4651	1.7935	- 2.12	0.4487	- 3.53	1.7519	**- 4.39**	0.3848	**-17.27**
`VGG-16`	6.5176	2.3834	5.6808	-12.84	0.6574	-72.42	5.6183	**-13.80**	0.3617	**-84.82**
`MobileNet-v1`	1.0597	0.9290	0.8738	-17.54	0.5398	-41.89	0.7621	**-28.08**	0.0674	**-92.74**
`SqueezeNet-v1.1`	0.8697	1.1208	0.7436	-14.50	0.5533	-50.63	0.6920	**-20.43**	0.1709	**-84.75**
Average	2.3286	1.0144	2.0858	- 9.79	0.4848	-27.85	**2.0309**	**-13.83**	**0.2112**	**-67.74**

[7] X. Ma, G. Yuan, S. Lin, C. Ding, F. Yu, T. Liu, W. Wen, X. Chen, and Y. Wang, "Tiny but accurate: A pruned, quantized and optimized memristor crossbar framework for ultra efficient dnn implementation," in *Proc. ASPDAC*, 2020.

[8] H. Li, M. Bhargav, P. N. Whatmough, and H.-S. P. Wong, "On-chip memory technology design space explorations for mobile deep neural network accelerators," in *Proc. DAC*, 2019.

[9] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proc. PPoPP*, 2019.

[10] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *Proc. DATE*, 2018.

[11] Q. Sun, T. Chen, J. Miao, and B. Yu, "Power-driven DNN dataflow optimization on FPGA," in *Proc. ICCAD*, 2019.

[12] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid dnn accelerator design and implementation," *arXiv preprint*, 2020.

[13] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in Halide," *ACM SIGGRAPH*, 2018.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI*, 2018.

[15] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. KDD*, 2016.

[16] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[17] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE TKDE*, 2009.

[18] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Proc. NIPS*, 2018.

[19] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *Proc. ICLR*, 2020.

[20] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. ICCAD*, 2019.

[21] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proc. ASPLOS*, 2020.

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *Proc. ICLR*, 2015.

[23] D. Kirk *et al.*, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007.

[24] B. Settles, "Active learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2009.

[25] L. Breiman, "Bagging predictors," *Machine learning*, 1996.

[26] E. W. Steyerberg, *Overfitting and Optimism in Prediction Models*. Springer International Publishing, 2019, pp. 95–112.

[27] K. Yu, J. Bi, and V. Tresp, "Active learning via transductive experimental design," in *Proc. ICML*, 2006.

[28] K. Brinker, "Incorporating diversity in active learning with support vector machines," in *Proc. ICML*, 2003.

[29] D. Wu, "Pool-based sequential active learning for regression," *IEEE TNNLS*, 2018.

[30] H. Yang, S. Li, C. Tabery, B. Lin, and B. Yu, "Bridging the gap between layout pattern sampling and hotspot detection via batch active learning," *IEEE TCAD*, 2020.

[31] "TVM-v0.6.1," https://github.com/apache/incubator-tvm/.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016.

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[35] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.