Resource-Guided Configuration Space Reduction for Deep Learning Models

Yanjie Gao[1], Yonghao Zhu[1], Hongyu Zhang[2], Haoxiang Lin[1†], Mao Yang[1]
[1]Microsoft Research, Beijing, China
[2]The University of Newcastle, NSW, Australia
Email: {yanjga, v-yonghz, haoxlin, maoyang}@microsoft.com, hongyu.zhang@newcastle.edu.au

***Abstract*—Deep learning models, like traditional software systems, provide a large number of configuration options. A deep learning model can be configured with different hyperparameters and neural architectures. Recently, AutoML (Automated Machine Learning) has been widely adopted to automate model training by systematically exploring diverse configurations. However, current AutoML approaches do not take into consideration the computational constraints imposed by various resources such as available memory, computing power of devices, or execution time. The training with non-conforming configurations could lead to many failed AutoML trial jobs or inappropriate models, which cause significant resource waste and severely slow down development productivity.**

In this paper, we propose DnnSAT, a resource-guided AutoML approach for deep learning models to help existing AutoML tools efficiently reduce the configuration space ahead of time. DnnSAT can speed up the search process and achieve equal or even better model learning performance because it excludes trial jobs not satisfying the constraints and saves resources for more trials. We formulate the resource-guided configuration space reduction as a constraint satisfaction problem. DnnSAT includes a unified analytic cost model to construct common constraints with respect to the model weight size, number of floating-point operations, model inference time, and GPU memory consumption. It then utilizes an SMT solver to obtain the satisfiable configurations of hyperparameters and neural architectures. Our evaluation results demonstrate the effectiveness of DnnSAT in accelerating state-of-the-art AutoML methods (Hyperparameter Optimization and Neural Architecture Search) with an average speedup from 1.19X to 3.95X on public benchmarks. We believe that DnnSAT can make AutoML more practical in a real-world environment with constrained resources.

***Index Terms*—configurable systems, deep learning, AutoML, constraint solving**

I. Introduction

Many traditional software systems, such as compilers and web servers, are highly configurable. They provide many configuration options, and different combinations of the options could lead to different system quality attributes. In recent years, deep learning (DL) models have become an integral part of many modern software-intensive systems such as speech recognition systems, chatbots, and games. Like traditional software systems, DL models also provide many configuration options for developers to tune. These configuration options can be largely classified into two categories: hyperparameter (such as the batch size and learning rate) and neural architecture (such as the number of layers and layer type). The numerous options provided by a DL model result in an exponential number of possible configurations, making manual tuning extremely tedious and time-consuming.

Recently, automated machine learning (AutoML) techniques have been widely adopted to automate and accelerate DL model tuning. AutoML tools such as NNI (Neural Network Intelligence) [1] usually apply *Hyperparameter Optimization* (HPO) [2] and *Neural Architecture Search* (NAS) [3] algorithms and launch hundreds or even thousands of AutoML trial jobs (or trials for short) to systematically explore diverse configurations of hyperparameters and neural architectures. It has been found that AutoML significantly boosts the productivity of DL development [2]–[4].

However, current AutoML approaches do not take into consideration the computational constraints imposed by various resources such as available memory, computing power of devices, or execution time, because the resources required by a DL model often remain unknown to developers before job execution. Non-conforming model configurations could lead to many failed AutoML trials or inappropriate models, which not only waste significant shared resources (such as GPU, CPU, storage, and network I/O) but also severely slow down development productivity. A typical resource is GPU memory, which is critical yet limited for training. If developers do not size the model carefully enough, trials will trigger *OOM* (out-of-memory) exceptions and fail. For instance, one PyTorch ResNet-50 [5] trial with an overlarge batch size of 256 causes an OOM when being scheduled on the NVIDIA Tesla P100 GPU; it requires 22 GB of GPU memory, but P100 has only 16 GB in total [6]. Even worse, since there can be more than ten tunable hyperparameters for the ResNet-50 model, other hundreds of trials with the same batch size could also experience OOM and crash. According to our recent empirical study on failed DL jobs in Microsoft [7], 8.8% of the 4960 job failures were caused by GPU memory exhaustion, which accounts for the largest category in all deep learning specific failures. Therefore, it is necessary for AutoML tools to enforce a constraint that a DL model cannot consume more GPU memory than the capacity when exploring model configurations. Another useful constraint is that the size of a model's weights cannot exceed a certain upper bound. Otherwise, the resulting DL application may be too large for efficient management and execution due to insufficient computing power of the target

†Corresponding author.

resource-restricted devices. If the unsatisfiable AutoML trials can be excluded ahead of time, resources will be saved to perform more trials, thus a larger configuration space could be explored.

A simple workaround is to run and profile trials for a while to estimate their resource consumption. Such a resource-consuming method is unaffordable in the scenario of AutoML, where there exist a large number of possible hyperparameter combinations and neural architectures. Some research work [8]–[10] incorporates certain resource quantification into the loss function for a global optimization with the model learning performance (*e.g.*, accuracy). However, their purpose is to reduce the resource consumption of the final model as much as possible while achieving an expected model learning performance, instead of excluding unsatisfiable trials in advance to improve the search efficiency. Therefore, such work could also cause failed trials/inappropriate models and resource waste. Besides, they are limited to NAS algorithms only and cannot be applied to HPO algorithms.

In this paper, we propose DnnSAT, a resource-guided AutoML approach for deep learning models, which can help existing AutoML tools efficiently reduce the configuration space ahead of time. We formulate such a resource-guided space reduction problem as a constraint satisfaction problem (CSP) [11]. DnnSAT includes a unified analytic cost model to construct common constraints with respect to the model weight size, number of floating-point operations (FLOPs), model inference time, and GPU memory consumption. It then utilizes an SMT (satisfiability modulo theories) solver (*e.g.*, Microsoft Z3 [12]) to obtain the satisfiable model configurations before trial execution. According to the characteristics of constraints (*e.g.*, monotonicity), we also apply some special optimizations to accelerate the solving.

We have implemented DnnSAT and evaluated it extensively on public AutoML benchmarks (HPOBench [13] and NAS-Bench-101 [14]) with various search methods (Random Search [15], Regularized Evolution [16], Hyperband [2], and Reinforcement Learning [14]) and representative DL models (VGG-16 [17] and LSTM [18]-based Seq2Seq [19]). The experimental results show that DnnSAT achieves an average speedup from 1.19X to 3.95X on public benchmarks and noticeably reduces the configuration space.

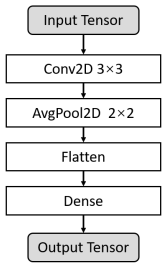
In summary, this paper makes the following contributions:

1) We propose a resource-guided AutoML approach for deep learning models to efficiently reduce the configuration space ahead of time.
2) We build a unified analytic cost model to construct common constraints and utilize an SMT solver to obtain the satisfiable configurations of hyperparameters and neural architectures.
3) We implement a tool named DnnSAT and demonstrate its practical effectiveness.

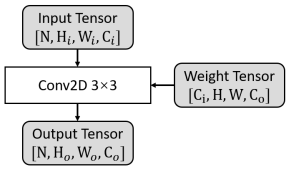
The rest of the paper is organized as follows. Section II introduces the background. Section III presents our analytic approach. Section IV details the design and implementation of DnnSAT. Section V shows experimental results. Section VI discusses extensibility and possible threats. We survey related work in Section VII and conclude this paper in Section VIII.

```
1 from tensorflow.keras import layers, models
2 ...
3 model = models.Sequential()
4 model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
  ↪  activation='relu', input_shape=(32, 32, 3)))
5 model.add(layers.AveragePooling2D(pool_size=(2, 2), padding
  ↪  ='same'))
6 model.add(layers.Flatten())
7 model.add(layers.Dense(units=64, activation='relu'))
8 model.compile(optimizer='adam',
  ↪  loss=tf.keras.losses.MeanSquaredError())
9 model.fit(train_images, train_labels, batch_size, epochs=10)
```

(a) Training program using Keras API.



(b) Computation graph for model inference.



(c) Tensors with shapes on which `Conv2D` operates.

Fig. 1: A sample TensorFlow model sequentially constructed by the framework built-in `Conv2D` (2D convolution), `AvgPool2D` (2D average pooling), `Flatten` (collapsing the input into one dimension without affecting the batch size), and `Dense` (fully connected layer) operators.

II. Background

Deep learning (DL) is a subfield of machine learning to learn layered data representations known as models. A DL model is formalized as a tensor-oriented computation graph [20] by frameworks such as TensorFlow (TF) [21], PyTorch [22], and MXNet [23], which is a *directed acyclic graph* (DAG). The inputs and outputs of such a computation graph and its nodes are *tensors* (multi-dimensional arrays of numerical values). The shape of a tensor is the element number in each dimension plus the element data type. Each node represents the invocation of a mathematical operation called an *operator* (*e.g.*, element-wise matrix addition). An edge delivers an output tensor and specifies the execution dependency. In this paper, we use the terms "operator" and "node" interchangeably since a node is completely determined by its invoked operator.

Fig. 1a shows a simple TensorFlow training program using the Keras [24] API, which sets up a sequential model with the framework built-in `Conv2D` (2D convolution with a 3×3 kernel size), `AvgPool2D` (2D average pooling with the "same" padding setting[1]), `Flatten` (collapsing the input into one dimension without affecting the batch size), and `Dense` (fully connected layer with 64 units) operators (lines 4-7). The above filter size, padding, and number of units are *hyperparameters*, which are parameters to control the training process. Fig. 1b demonstrates the corresponding computation graph for model

[1]See https://keras.io/api/layers/pooling_layers/average_pooling2d/ for an explanation of the padding argument.

```
# search_space.json
{
  "batch_size":{"_type":"choice", "_value":[16,32,64]},
  "kernel_size":{"_type":"choice", "_value":[3,5,7,11]},
  "filters":{"_type":"choice", "_value":[64,128,512]},
  "unit_size":{"_type":"choice", "_value":[64,512]},
  "lr":{"_type":"choice", "_value":[0.0001,0.001,0.01,0.1]}
}

# config.yml
...
searchSpacePath: search_space.json
tuner:
  builtinTunerName: Random
trial:
  command: python3 mnist.py
  gpuNum: 1
  cpuNum: 1
  memoryMB: 8196
...
```

Fig. 2: Settings of an MNIST training program supported by NNI with Hyperparameter Optimization.

inference. Fig. 1c illustrates the input, weight, and output tensors with shapes on which `Conv2D` operates.

To choose better configurations of hyperparameters and neural architectures, developers often adopt a trial-and-error strategy: submitting hundreds or even thousands of trial jobs with each being assigned a different model configuration. This strategy is very inefficient because of the overlarge configuration space. Recently, many AutoML tools, such as NNI (Neural Network Intelligence) [1], Auto-Keras [4], and Auto-Sklearn [25], are proposed to automate the exploration of model configurations. They help developers find a hyperparameter combination (Hyperparameter Optimization (HPO) [13]) or design an elaborate neural network (Neural Architecture Search (NAS) [14]), which can both minimize the loss and maximize the model learning performance (*e.g.*, accuracy).

Suppose that a DL model has m hyperparameters whose domains are $B_1, B_2, \cdots, B_m$. A model configuration is an instance of such a model with concrete hyperparameter values. All model configurations constitute the *configuration space*. HPO applies some search method (*e.g.*, random search, evolutionary strategies, or Bayesian optimization) in the enumeration of the configuration space to find a candidate with the optimal hyperparameter vector (combination) $\lambda \in B_1 \times B_2 \times \cdots \times B_m$, which best meets the training objective. Fig. 2 shows the settings of an MNIST training program supported by the AutoML tool NNI with HPO. The upper part (lines 1-8) defines the possible values of tunable hyperparameters (batch size, learning rate, *etc.*); the bottom part (lines 10-20) specifies the search method and runtime resource requirements (GPU count, main memory size, *etc.*). The *controller* process of NNI performs a random search (line 14) and may fork over a hundred trials executing the command on line 16. Trials send timely feedback such as the mean squared error or accuracy to the controller for the decision of early stopping.

Similarly, NAS automates the architecture engineering of a DL model. The configuration space consists of various automatically generated, syntactically legal neural network structures such as chained or multi-branch networks. NAS also applies different search methods including random search, gradient-based algorithms, and reinforcement learning (RL) [26] for the space exploration.

TABLE I
COMMON HYPERPARAMETERS OF DL MODELS.

Hyperparameter	Domain	Hyperparameter	Domain
Batch Size	$\mathbb{N}$	Kernel Size	$\mathbb{N}$
Output Channels	$\mathbb{N}$	Stride	$\mathbb{N}$
Padding	$\mathbb{N}$	# of RNN Layers	$\mathbb{N}$
# of Units	$\mathbb{N}$	Sequence Length	$\mathbb{N}$
Operator Type	$\mathbb{N}$	Edge	$\mathbb{B}$
Learning Rate	$\mathbb{R}^+$	Dropout	$[0, 1)$

III. Proposed Approach

A. Problem Formulation

Formally, a DL model $\boldsymbol{X}$ is represented as a directed acyclic graph (DAG) [20]:

$$\boldsymbol{X} = \langle \{u_i\}_{i=1}^{n}, \{(u_i, u_j)\}_{i \neq j}, \{p_k\}_{k=1}^{m} \rangle$$

Each node u_i is an operator (*i.e.*, a mathematical operation such as convolution and pooling). A directed edge (u_i, u_j) pointing from an output of node u_i to an input of u_j delivers the output tensor and specifies the execution dependency. Each p_k is a hyperparameter (*e.g.*, input tensor shape and batch size) whose domain is denoted as B_k. Table I lists some commonly used hyperparameters with their domains.

For $b_k \in B_k$ where $1 \leq k \leq m$, we use $\boldsymbol{X}(b_1, b_2, \cdots, b_m)$ to represent one model configuration of $\boldsymbol{X}$. Then, the configuration space of model $\boldsymbol{X}$, denoted by $\boldsymbol{\Delta_X}$, is defined as follows:

$$\boldsymbol{\Delta_X} = \{\boldsymbol{X}(b_1, b_2, \cdots, b_m) \mid b_k \in B_k \text{ for } k \in [1, m]\}$$

In an AutoML experiment, there may exist a series of DL models $\boldsymbol{X}_1, \boldsymbol{X}_2, \cdots, \boldsymbol{X}_N$ (*e.g.*, models searched by NAS). For such an experiment, its configuration space $\boldsymbol{\Delta}$ is defined as the union set of all models' configuration spaces:

$$\boldsymbol{\Delta} = \bigcup_{i=1}^{N} \boldsymbol{\Delta_{X_i}}$$

We formulate the resource-guided configuration space reduction for an AutoML experiment with N models as the following constraint satisfaction problem $\langle V, D, C \rangle$ [11]:

$$V = \{V_1, V_2, \cdots, V_N\}$$
$$D = \{D_1, D_2, \cdots, D_N \mid D_i = \boldsymbol{\Delta_{X_i}} \text{ for } i \in [1, N]\}$$
$$C = \{C_j : lb_j \leq f_j(V_i) \leq ub_j\}$$

V is a set of model configuration variables, D represents the respective variable domains, and C is the constraint set. Each variable $V_{i \in [1,N]}$ can take on the model configurations of $\boldsymbol{X}_i$ (*i.e.*, the domain of V_i is $\boldsymbol{\Delta_{X_i}}$). For a constraint C_j, f_j is a non-negative *restriction* function which denotes the demand for a certain resource; lb_j and ub_j are the lower and upper bounds of that resource, respectively. If one is more interested in the upper bound, we simply assume that $lb_j = 0$. Hyperparameters

are *decision variables* of the constraints, which are quantities controlled by the decision-maker to define the search space for optimization. As mentioned before, an example f_j calculates the GPU memory consumption for training a DL model, and ub_j is the memory capacity of the allocated GPU device (*e.g.*, 16 GB for NVIDIA Tesla P100).

B. The Proposed Analytic Approach

DnnSAT adopts an analytic approach to construct the restriction functions on resources. We observe that the algorithmic execution of a DL model can be represented as iterative forward and backward propagation on its computation graph.[2] Therefore, computing the resource consumption for one iteration is then reduced to the calculation of the resource required by each operator on the computation graph in accordance with a certain graph traversal order. Currently, a DL model is required to be deterministic without control-flow operators (*e.g.*, loops and conditional branches), thus we assume that the execution flow and resource consumption across different iterations are identical. We define an auxiliary function g on the operator set, which represents the *current* resource consumption when the operator under visiting has just finished execution in the iteration. Let $S = \langle u_{i_1}, u_{i_2}, \cdots, u_{i_n} \rangle$ be a topological (linear) order extended from the edge order of the model such that $u_{i_j} \prec_S u_{i_k} \implies (u_{i_k}, u_{i_j}) \notin \boldsymbol{X}$. S is called an *operator schedule*, representing the actual runtime execution order of operators. DnnSAT pre-generates S by referring to the framework implementations [27]–[29]. Suppose that r is the operator resource cost function, $IterCnt$ is the iteration count, and R_{init}, R_{fini} are the resource consumption of the one-off initialization and clean-up performed by DL frameworks which are assumed to be 0 if not specifically mentioned. We define g and the restriction function f as follows:

$$
\begin{aligned}
g(u_{i_1}) &= r(u_{i_1}) \\
g(u_{i_j}) &= h(\{\langle u_{i_1}, g(u_{i_1})\rangle, \cdots, \langle u_{i_{j-1}}, g(u_{i_{j-1}})\rangle\}) + r(u_{i_j}) \\
f(\boldsymbol{X}) &= R_{init} + R_{fini} + t(IterCnt, \{g(u_{i_j})\}_{j=1}^{n})
\end{aligned}
$$

So long as the above h and t are functions, g uniquely exists by the transfinite recursion theorem schema [30] since S is a well order, and f also exists. The formalization of g indicates that it could compute the current resource consumption by referring to extra information of previous consumption and visited operators. Examples of g and f can be found in Section IV-B. DnnSAT includes a unified analytic cost model to define operators' resource cost functions and common constraints. Table II lists the constraints that we have implemented.

The objective of DnnSAT is to reduce the configuration space $\boldsymbol{\Delta}$ by eliminating those $\boldsymbol{X}_i(b_{i_1}, b_{i_2}, \cdots, b_{i_m})$ which violate the enforced constraints before submitting AutoML trials. A naive method is to compute the values of the restriction functions for each model configuration in $\boldsymbol{\Delta}$ and then check whether such values fall within the allowed bounds. DnnSAT adopts a much more efficient approach: it specifies those constraints in the SMT-LIB (Satisfiability Modulo Theories Library [31]) syntax and uses the Microsoft Z3 solver [12] to obtain all the satisfiable model configurations. Such an approach is feasible because the restriction functions of the common constraints can be composed by SMT solvers' built-in functions (*e.g.*, multiplication and division) and the constraints are simple numerical inequalities. We also apply some special optimizations to accelerate the solving based on the constraint characteristics (*e.g.*, monotonicity), which are described in Section IV-E.

[2]This also applies to model inference which has a simplified representation with single-pass forward propagation and no backward propagation.

TABLE II
COMMON CONSTRAINTS IMPOSED BY THE COMPUTATIONAL RESOURCES.

Restriction Category	Resource Upper Bound	Scenario
Model Weight Size	Allowed Binary Size	Inference
Number of Floating-point Operations (FLOPs)	(Device FLOPS) × (SLA of Inference Latency)*	Inference
Model Inference Time	SLA of Inference Latency	Inference
GPU Memory Consumption	Device Memory Capacity	Training Inference

*"FLOPS" denotes floating-point operations per second, and "SLA" stands for service-level agreement.

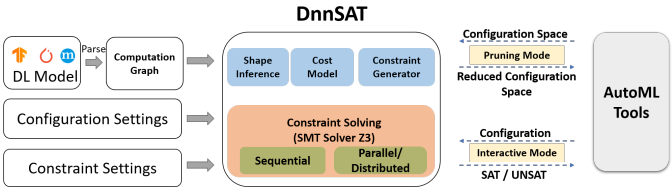


Fig. 3: Workflow of DnnSAT.

IV. DESIGN AND IMPLEMENTATION

A. Workflow

Fig. 3 shows the workflow of DnnSAT. It accepts a source DL model, configuration settings, and constraint settings as input. The model is parsed by a front-end parser and reconstructed to the corresponding computation graph. Some DL frameworks such as PyTorch employ the *define-by-run* approach [22] such that a saved model records only an execution trace instead of the full computation graph. DnnSAT currently relies on users to supply multiple models in case the graph may dynamically change (in the future, it may be possible to try extracting the full computation graph from a DL program by static code analysis). Configuration settings include the hyperparameters to be tuned and their domain definitions. Constraint settings contain the built-in constraints that need to be satisfied and their allowed bounds, as well as constraint-related runtime constants of the DL framework (*e.g.*, type and version) and target device (*e.g.*, FLOPS).

DnnSAT has defined four common constraints (Section IV-B) and a set of analytic and framework-independent resource cost functions for DL operators (Section IV-C). It traverses the computation graph in accordance with a predefined operator schedule (*i.e.*, operator execution order) to automatically generate the constraint specifications in SMT-LIB for later solving (Section IV-D). There are two working modes for

integrating DnnSAT with existing AutoML tools. One is the *interactive* mode, in which AutoML tools work as usual but each configuration will be sent to DnnSAT for solving via an API call before launching a trial. Such a mode is simple, non-intrusive, and requires less effort. We have run DnnSAT with HPOBench [13] and NAS-Bench-101 [14] interactively to evaluate the effectiveness in speeding up AutoML methods (Section V-A), and similar integration work could be done in AutoML tools such as NNI and Auto-Keras. The other is the *pruning* mode, in which DnnSAT eliminates the unsatisfiable model configurations in advance and feedbacks a reduced configuration space to AutoML tools during their initialization. To solve the constraints, the Microsoft Z3 solver is invoked with some optimizations (Section IV-E). DnnSAT is extensible to support user-defined constraints by permitting users to specify their own constraint specifications in SMT-LIB with those defined hyperparameters.

B. The Constraints

DL models are both compute-intensive and memory-intensive, making them difficult to be trained or deployed on systems and platforms with limited resources. In this paper, we consider four representative computational constraints with respect to the model, namely weight size, number of floating-point operations, inference time, and GPU memory consumption. The meaning of the notations and symbols can be found in Section III.

Model Weight Size. Weights (including biases) are the numerical learnable parameters of operators, being saved in the model file and taking up most of the space. The size of weights is important, especially on resource-restricted devices such as mobile phones. An overlarge DL model causes inefficient model/application management, expends unaffordable energy [32], or even cannot fit in the target devices' main memory. The total model weight size is calculated by accumulating the weight size of each operator. Assuming that WT is the restriction function and WT_{min}, WT_{max} are the lower and upper bounds in bytes, the constraint is then defined as follows:

$$\begin{aligned} g(u_{i_j}) &= g(u_{i_{j-1}}) + r(u_{i_j}) \\ WT(\boldsymbol{X}) &= \max\{g(u_{i_j})\}_{j=1}^{n} = \textstyle\sum_{j=1}^{n} r(u_{i_j}) \\ WT_{min} &\le WT(\boldsymbol{X}) \le WT_{max} \end{aligned}$$

Number of Floating-point Operations (FLOPs). FLOPs is considered as a stronger predictor of energy usage and inference time [33]. The total FLOPs for inference is calculated by accumulating the FLOPs of each operator in accordance with the operator schedule. Assuming that F is the restriction function and F_{min}, F_{max} are the minimal and maximal values allowed, the constraint is then defined as follows:

$$\begin{aligned} g(u_{i_j}) &= g(u_{i_{j-1}}) + r(u_{i_j}),\ IterCnt = 1 \\ F(\boldsymbol{X}) &= IterCnt \times \max\{g(u_{i_j})\}_{j=1}^{n} = \textstyle\sum_{j=1}^{n} r(u_{i_j}) \\ F_{min} &\le F(\boldsymbol{X}) \le F_{max} \end{aligned}$$

Model Inference Time. This is a critical runtime performance indicator for DL applications. The total time is calculated by accumulating the execution time of each operator in accordance with the operator schedule. Assuming that T is the restriction function and T_{min}, T_{max} are the lower and upper bounds, the constraint is then defined as follows:

$$\begin{aligned} g(u_{i_j}) &= g(u_{i_{j-1}}) + r(u_{i_j}),\ IterCnt = 1 \\ T(\boldsymbol{X}) &= IterCnt \times \max\{g(u_{i_j})\}_{j=1}^{n} = \textstyle\sum_{j=1}^{n} r(u_{i_j}) \\ T_{min} &\le T(\boldsymbol{X}) \le T_{max} \end{aligned}$$

GPU Memory Consumption. As mentioned before, GPU OOM accounts for the largest DL failure category [7], therefore controlling the GPU memory consumption is critical to reduce OOM exceptions and save shared resources. However, the calculation is rather complicated since there are many hidden framework factors observably affecting the final GPU memory consumption [6]. We adopt a simplified yet common approach for both inference and data-parallel training, which accumulates the GPU memory required by each operator under forward propagation in accordance with the operator schedule. Assuming that M is the restriction function and M_{min}, M_{max} are the minimal and maximal GPU memory consumption allowed in bytes (*e.g.*, taking the memory capacity as the maximal value), the constraint is then defined as follows:

$$\begin{aligned} g(u_{i_j}) &= g(u_{i_{j-1}}) + r(u_{i_j}) \\ M(\boldsymbol{X}) &= R_{init} + \max\{g(u_{i_j})\}_{j=1}^{n} = R_{init} + \textstyle\sum_{j=1}^{n} r(u_{i_j}) \\ M_{min} &\le M(\boldsymbol{X}) \le M_{max} \end{aligned}$$

If u is an operator under backward propagation, we let $r(u) = 0$. R_{init} represents the GPU memory consumed by the CUDA context and initial input tensors during the framework initialization. The CUDA context contains managing information to control and use GPU devices, which is assumed to be a constant obtained by profiling.

C. Resource Cost Functions of Operators

Operators are mathematical functions on various types of tensors. DnnSAT defines analytic and framework-independent resource cost functions for operators. Such a solution is technically feasible because: (1) frequently used operators are well-defined with clear syntax and semantics; (2) DL frameworks implement them similarly (*e.g.*, calling NVIDIA CUDA, cuDNN, or cuBLAS APIs). In this section, we take the `Conv2D` operator in Fig. 1b as an example to illustrate the four resource cost functions with respect to the studied constraints.

The following symbols are used to denote the hyperparameters and tensor shapes. S_f is the size of input data type (*e.g.*, 4 bytes for `FLOAT32` data). N represents batch size. H_k and W_k are kernel (filter) height and width; they are usually equal. flt_i and s_i are filter size and stride size at index i. flt represents the number of filters. Padding pad_i "controls the amount of implicit zero-paddings on both sides for padding number of points for each dimension", and dilation d_i "controls the spacing between the kernel points" [34]. H_{in}, W_{in}, and

C_{in} are input height, width, and channels, respectively. H_o, W_o, and C_o are output height, width, and channels, which have the following relations with other symbols:

$$H_o = 1 + (H_{in} + 2 \times pad_0 - d_0 \times (H_k - 1) - 1)/s_0$$
$$W_o = 1 + (W_{in} + 2 \times pad_1 - d_1 \times (W_k - 1) - 1)/s_1$$
$$C_o = flt$$

Then, the resource cost functions with respect to the four constraints **model weight size** (WT), **FLOPs** (F), **model inference time** (T), and **GPU memory consumption** (M) are defined as follows:

$$WT(\texttt{Conv2D}) = S_f \times (C_{in} \times H_k \times W_k \times C_o + C_o)$$
$$F(\texttt{Conv2D}) = 2 \times C_o \times (H_k \times W_k \times C_{in} + 1) \times N \times H_o \times W_o$$
$$M_{it} = N \times C_{in} \times H_{in} \times W_{in}$$
$$M_{wt} = WT(\texttt{Conv2D})$$
$$M_{ot} = S_f \times N \times C_o \times H_o \times W_o$$
$$T(\texttt{Conv2D}) = (M_{it} + M_{wt} + M_{ot})/B_d + F(\texttt{Conv2D})/F_{lops}$$
$$M(\texttt{Conv2D}) = M_{wt} + M_{ot}$$

M_{it}, M_{wt}, and M_{out} denote the GPU memory occupied by the input, weight, and output tensors, respectively. B_d and F_{lops} are memory bandwidth and floating-point operations per second (FLOPS) of the target device, which can be assumed to be constants. The first item of $T(\texttt{Conv2D})$ represents the data access time from and to GPU memory [35]. We do not count M_{it} in the GPU memory consumption because an operator's input tensors reuse the GPU memory of either the initial inputs or predecessors' outputs, which have been calculated in the initialization cost (R_{init}) and predecessor operators' cost functions. More details about the estimation of FLOPs and model inference time can be found at [35] and [36].

Currently, DnnSAT supports 70+ operators. It is also extensible and can support new operators, which are discussed in Section VI.

D. Constraint Specification

In this section, we describe how to automatically generate a constraint specification in SMT-LIB. The DL model in Fig. 1 is used as an example, and the enforced constraint is *the model weight size must be less than or equal to 10 MB*. A snippet of the constraint setting is shown as follows:

```
{"constraint":"weight_size","max":10485760,"min":0}
```

Fig. 4 lists the illustrated SMT-LIB code. First, DnnSAT specifies the constraint bounds read from the constraint setting file (lines 3-5). It then declares the hyperparameters of the size of input data type (S_f), number of input channels (C_{in}), kernel size (H_k and W_k), filter size (flt), and unit size (U) of the `Dense` operator (lines 7-12) and writes their domains (lines 14-16), according to the contents of the configuration setting file. As the batch size (N) does not contribute to the model weight size, we simply ignore it.

Next, DnnSAT traverses the computation graph from `Conv2D` to `Dense` one after another. For each operator type, we prepare SMT-LIB templates in Python via Z3 APIs for

```
1  (set-logic QF_UFNIA) ; Non-linear integer arithmetic logic
2  ; Constraint bounds
3  (declare-const WT_Min Int) ; Lower bound: 0 MB
4  (declare-const WT_Max Int) ; Upper bound: 10 MB
5  (assert (and (= WT_Min 0) (= WT_Max 10485760)))
6  ; Hyperparameters
7  (declare-const Sf Int) ; Size of input data type
8  (declare-const Cin Int) ; Conv2D input channels
9  (declare-const Hk Int) ; Conv2D kernel height
10 (declare-const Wk Int) ; Conv2D kernel weight
11 (declare-const flt Int) ; Conv2D number of filters
12 (declare-const U Int) ; Dense unit size
13 ; Hyperparameter domains
14 (assert (and (and (= Sf 4) (= Cin 3)) (and (= Hk Wk) (or (=
   ↪  U 64) (= U 512)))))
15 (assert (or (or (= Hk 3) (= Hk 5)) (or (= Hk 7) (= Hk 11))))
16 (assert (or (or (= flt 64) (= flt 128)) (= flt 512)))
17 ; Compute the weight size of Conv2D
18 (declare-const WT_Conv2D Int) ; Conv2D weight size
19 (declare-const Co Int) ; Conv2D output channels
20 (assert (= flt Co))
21 (assert (= (* Sf (+ (* (*(* Hk Wk) Cin) Co) Co)) WT_Conv2D))
22 ; Compute the weight size of Dense
23 (declare-const WT_Dense Int)
24 (assert (= (* Sf (+ U 1)) WT_Dense))
25 ; Compute the model weight size
26 (declare-const WT Int)
27 (assert (= (+ WT_Conv2D WT_Dense) WT))
28 ; Specify the constraint
29 (assert (and (<= WT_Min WT) (<= WT WT_Max)))
30 (check-sat)
31 (exit)
```

Fig. 4: Illustrated constraint specification in SMT-LIB, which enforces that the weight size of the model in Fig. 1 must be within the interval [0 MB, 10 MB].

the resource cost functions. When an operator is visited, DnnSAT locates the matched template and generates the Python wrapper of SMT-LIB code using those declared hyperparameter symbols. For example, lines 18-21 correspond to calculating the weight size of `Conv2D`. Since `AvgPool2D` uses the "same" padding setting (line 5 in Fig. 1a), its output tensor shape keeps unchanged with that of `Conv2D`. Both `AvgPool2D` and `Flatten` do not have weights, so we also ignore them. `Dense` has a weight tensor of a 64-element array plus a bias of 1 data element (lines 23-24). Therefore, the total model weight size is equal to the sum of the weight sizes of both `Conv2D` and `Dense` (lines 26-27). Finally, DnnSAT asserts that the result must fall between the lower and upper bounds (line 29). Note that although the example uses only integer variables, DnnSAT can easily replace the variable statements to support real-valued hyperparameters because the underlying SMT solver Z3 supports real values and real functions.

E. Constraint Solving

The resource-guided configuration space reduction is formulated as a constraint satisfaction problem (CSP). DnnSAT chooses Microsoft Z3 to solve the constraints because the SMT solver is very efficient to handle higher-order and nonlinear functions. However, the solving may slow down significantly when dealing with an overlarge configuration space or a very complicated restriction function. We summarize below our major optimization techniques for accelerating the solving speed:

1) **Parallel and distributed solving**. DnnSAT partitions the full configuration space into multiple smaller subspaces and solves them in parallel. The process is shown schematically in Fig. 5. For the parallel solving, each worker thread is assigned a standalone Z3 context. The distributed solving is built on top of Spark [37], which handles configuration space partitioning, distributed task deployment (via the `mapPartitions` API), scheduling, and fault tolerance.
2) **Tiny subspaces**. Proper partitioning of the configuration space is very important for tackling the skew problem [38], [39] in the parallel and distributed solving. DnnSAT adopts the idea of tiny tasks [40] and divides the original space into numerous tiny subspaces (*e.g.*, each containing less than 100 configurations). Our approach has two advantages: (1) it will not result in observable computation skew across subspaces; (2) the Z3 solver cannot return all the satisfiable model configurations at once like what ALLSAT [41] does, hence DnnSAT has to iteratively call Z3 by providing the conjunction of the negation of each existing solution to derive the next one. A tiny subspace does not make the conjunction long and complicated, thus the solving efficiency is significantly improved. DnnSAT currently implements a simple work queue to manage the tiny subspaces. We will consider dynamic partitioning and work stealing [42] for better load balancing in the future.
3) **Interval filtering**. The restriction function of a constraint may be monotonic with regard to (w.r.t. for short) some hyperparameters. For instance, the model weight size function is monotonically increasing w.r.t. kernel size, filter size, and unit size mentioned in Fig. 4. Another example is that the FLOPs function is monotonically increasing w.r.t. batch size but not to kernel size. This is because the output height and width of the `Conv2D` operator decrease with kernel size increasing, which may reduce the FLOPs of subsequent operators. With such monotonicity information, we can apply the interval filtering technique which safely discards specific value intervals of the hyperparameters. Suppose that the restriction function is monotonically increasing w.r.t. the hyperparameter p_1 whose domain is $[v_{min}, v_{max}]$. If the function value of a configuration $\langle p_1 = v_1, p_2 = v_{p_2}, \cdots, p_m = v_{p_m} \rangle$ exceeds the upper bound, any configurations $\langle p_1 \in (v_1, v_{max}], p_2 = v_{p_2}, \cdots, p_m = v_{p_m} \rangle$ will not satisfy the constraint either; if it is smaller than the lower bound, any configurations $\langle p_1 \in [v_{min}, v_1), p_2 = v_{p_2}, \cdots, p_m = v_{p_m} \rangle$ will also violate the constraint. Furthermore, if two configurations $\langle p_1 \in \{v_1, v_2\}, p_2 = v_{p_2}, \cdots, p_m = v_{p_m} \rangle$ are satisfied and $v_1 < v_2$, any configurations $\langle p_1 \in (v_1, v_2), p_2 = v_{p_2}, \cdots, p_m = v_{p_m} \rangle$ will satisfy the constraint as well.

V. Evaluation

To evaluate the proposed DnnSAT, we experiment with different representative AutoML benchmarks and DL models.

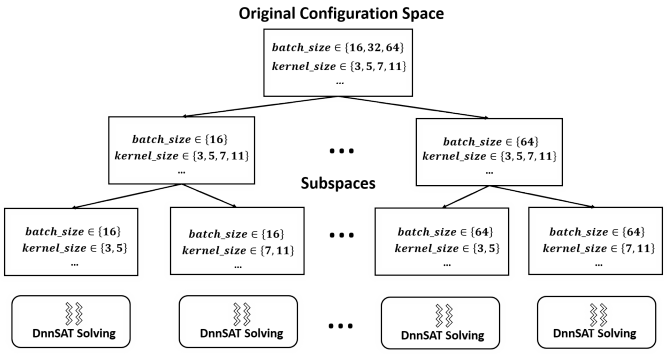


Fig. 5: Parallel and distributed solving by partitioning the configuration space.

We aim to answer the following Research Questions (RQs):
RQ1: How effective is DnnSAT in speeding up AutoML methods?
RQ2: How effective is DnnSAT in reducing the configuration space?
RQ3: How efficient is DnnSAT in constraint solving?

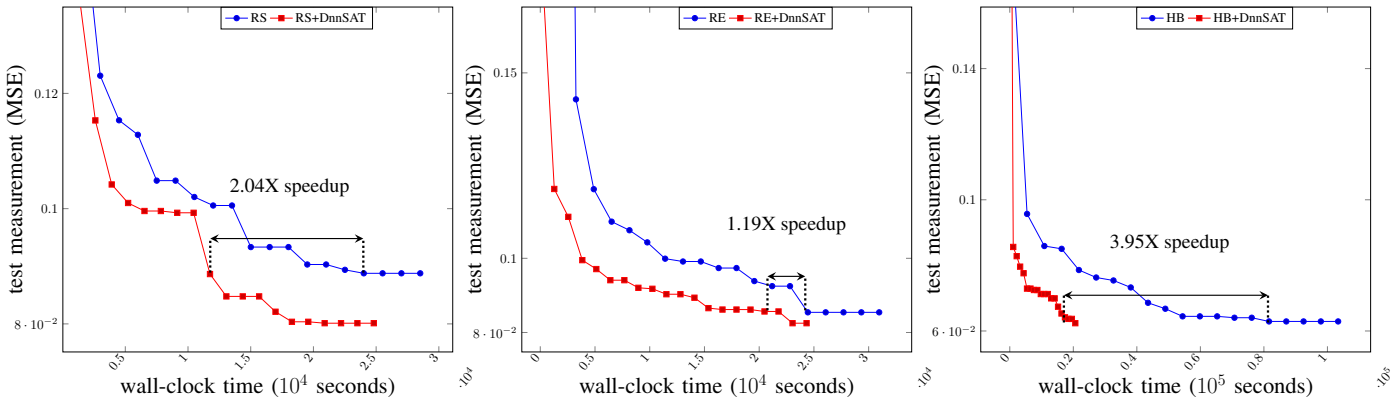
Our experiments are conducted on an Azure Standard ND12s virtual machine with 12 Intel Xeon E5-2690 vCPUs, 224 GB main memory, and 2 NVIDIA Tesla P40 (24 GB GDDR5X memory) GPUs, running Ubuntu 16.04.

A. RQ1: How effective is DnnSAT in speeding up AutoML methods?

In this section, we consider the model weight size constraint and evaluate the effectiveness of DnnSAT on the following two benchmarks:

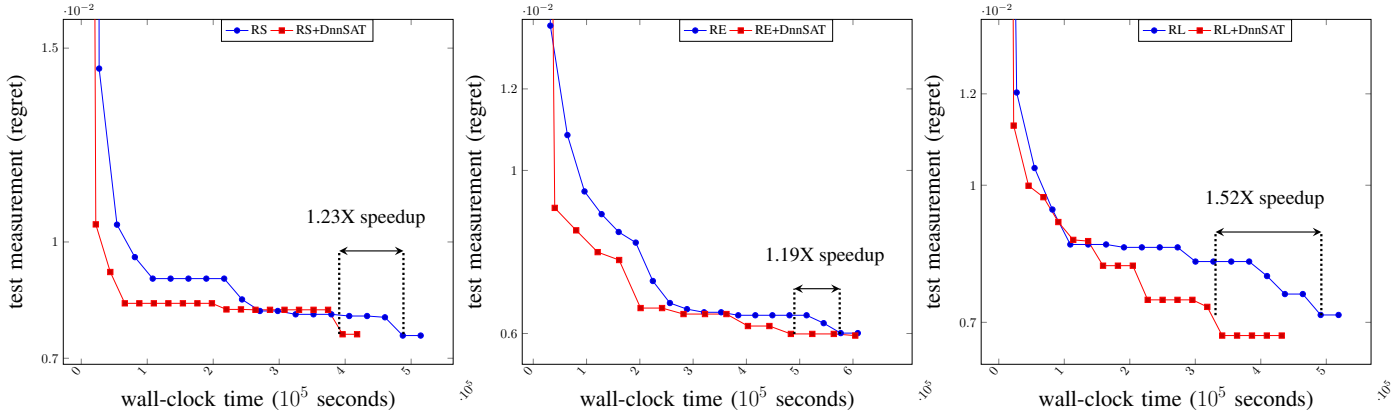
1) **HPOBench** is for HPO methods and consists of "a large grid of hyperparameter configurations of feedforward neural networks for regression" [13]. The model has two tunable `Dense` operators followed by a non-tunable `Dense` on top. There are nine hyperparameters (*e.g.*, batch size, unit size, and initial learning rate) and 62208 model configurations in total. We use the HPO-Bench-Protein dataset. The maximal model weight size is 256 KB.
2) **NAS-Bench-101** is for NAS methods and "constructs a compact, yet expressive, search space, exploiting graph isomorphisms to identify 423k unique convolutional architectures" [14]. The dataset is CIFAR10 [43]. The maximal model weight size is about 47.7 MB.

Both benchmarks generated DL models and collected a rich set of runtime statistics (final training/validation/test error and accuracy, total training time, *etc.*) from the trained models. These statistics can be used to simulate the execution of AutoML trials and evaluate different configuration search methods. To evaluate the learning performance of a model configuration resulted from an AutoML trial, following the existing work [8], [13], [14], [44], we use the mean squared error (MSE) for HPOBench and *regret* (*i.e.*, 1 − accuracy) for NAS-Bench-101 as the test measurement. The smaller the value, the better the model. We choose the model weight



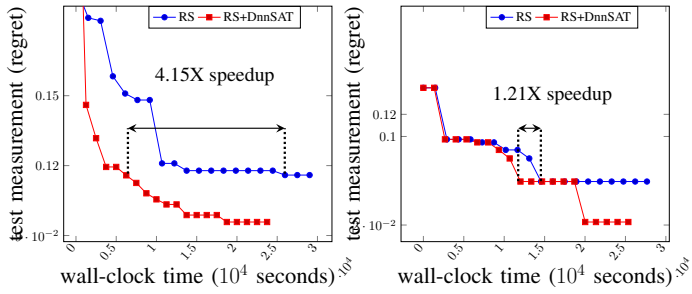
(a) Random Search (RS) (b) Regularized Evolution (RE) (c) Hyperband (HB)

Fig. 6: Test measurement curves and speedups of the HPOBench experiments on 100 trials using different search methods, with an 8 KB upper bound of the model weight size.



(a) Random Search (RS) (b) Regularized Evolution (RE) (c) Reinforcement Learning (RL)

Fig. 7: Test measurement curves and speedups of the NAS-Bench-101 experiments on 500 trials using different search methods, with a 38 MB upper bound of the model weight size.



(a) Random Search (RS), 2 KB (b) Random Search (RS), 16 KB

Fig. 8: Test measurement curves and speedups of the HPOBench experiments on 100 trials using Random Search, with three upper bounds (2 KB, 8 KB, and 16 KB) of the model weight size.

size constraint because it is one of the most representative constraints for DL applications and for performing model inference on resource-restricted devices such as mobile phones.

On each benchmark, we perform two sets of experiments: baseline experiments and DnnSAT-guided experiments. We choose the following commonly used search methods in AutoML as baselines: Random Search (RS) [15], Regularized Evolution (RE) [16], Hyperband (HB; HPOBench only) [2], and Reinforcement Learning (RL; NAS-Bench-101 only) [14]. The two sets of experiments perform the same number of trials (100 for HPOBench and 500 for NAS-Bench-101). We then measure the speedup achieved by DnnSAT over each baseline method on each benchmark. The speedup is calculated as $\frac{T_{base}}{T_{\text{DnnSAT}}}$, where T_{base} is the estimated time of a baseline (*i.e.*, RS, RE, and HB) reaching the lowest test measurement (*i.e.*, the best learning performance) and T_{DnnSAT} is the time of a DnnSAT-guided method reaching the same or lower test measurement. We repeat each experiment 10 times and compute the average speedup value. DnnSAT runs in the interactive mode because of the simplicity of the integration effort. To solve the first model configuration, DnnSAT needs to build the constraint from scratch (*i.e.*, warm-up), which spends more time than solving later individual configurations of the same model. For HPOBench, the solving time is 0.1s (warm-up: 2.5s) on average. For NAS-Bench-101, it is 0.6s (warm-up: 14.0s) on average.

Fig. 6 demonstrates the test measurement (MSE) curves of HPOBench on 100 evaluated trials. The upper bound of the

model weight size is set to a small value of 8 KB (which is set for deploying KB-sized DL models to resource-restricted IoT devices [45]). The x-axis is the wall-clock time, and the y-axis denotes the test measurement value. Overall, DnnSAT achieves a speedup of 2.04X (RS), 1.19X (RE), and 3.95X (HB) on HPOBench, for obtaining the same optimal model learning performance that can be found by the baseline. From the experiment results, we also find that DnnSAT helps the curves go down faster and reach smaller test measurement values, which means that better model configurations are found. The reason is that DnnSAT reduces the configuration space so that HPOBench can perform a much more efficient search than before. We also notice that the experiment time has been observably shortened when DnnSAT is enabled since a smaller model size implies fewer FLOPs and thus less training time. In Fig. 6c, time reduction is particularly significant because of the mechanism of Hyperband: the more resources saved by DnnSAT, the more training budget allocated to Hyperband for high-efficiency search.

Fig. 7 demonstrates the test measurement (regret) curves of NAS-Bench-101 on 500 evaluated trials, with the upper bound of the model weight size set to 38 MB. DnnSAT achieves a speedup of 1.23X (RS), 1.19X (RE), and 1.52X (RL).

To understand the generality of our approach under different constraint bounds, we additionally choose two upper bounds (2 KB and 16 KB) and conduct HPOBench experiments using Random Search. The results of Fig. 8 and Fig. 6a indicate that DnnSAT is generally effective and achieves a speedup of 4.15X (2 KB), 2.04X (8 KB), and 1.21X (16 KB). The results also show that the stricter the constraint, the greater the improvement achieved by DnnSAT.

B. RQ2: How effective is DnnSAT in reducing the configuration space?

In this section, we evaluate the reduction effectiveness of DnnSAT on real-world DL models. We choose two representative models as our experimental subjects, namely VGG-16 (VGG model with 16 layers) [17] and LSTM [18]-based Seq2Seq [19]. For VGG-16, we select batch size (interval $[1, 256]$), kernel size (1, 3, and 5), and unit size (128, 512, 1024, 4096, and 10240) as the hyperparameters; hence there are 3840 model configurations in total. For Seq2Seq, we consider batch size (interval $[128, 512]$) and hidden size (interval $[16, 128]$), in which the configuration space consists of 43505 model configurations.

We apply all the four discussed constraints separately and collectively to both DL models. Each constraint is set with several upper bounds. The bound choices are based on actual conditions such as the model scale, device capability, and application SLA. For example, we choose 6, 8, and 12 GB for VGG-16 under the GPU memory consumption constraint because they correspond to three typical memory capacities of NVIDIA GPUs. Since the two models differ a lot, we cannot use the same bound value for both. Note that we use *batched inference time*, that is, the total inference time of a batch of data items. Specific upper bound values can be found in Table III.

TABLE III
CONFIGURATION SPACE REDUCTION ON REAL-WORLD DL MODELS.

Model Name	Max Weight Size (MB)	Max GPU Memory (GB)	Max BI Time (S)	Max FLOPs (GFLOPs)	All Constraints
VGG-16	1024 (80.0%)	12 (84.3%)	1000 (80.3%)	4096 (58.4%)	(53.4%)
	512 (60.0%)	8 (56.2%)	800 (64.0%)	3584 (51.1%)	(31.8%)
	128 (0.0%)	6 (42.1%)	10 (0.7%)	3072 (43.7%)	(0.0%)
Seq2Seq (LSTM)	128 (32.1%)	0.6 (34.7%)	50 (74.1%)	64 (35.2%)	(24.1%)
	64 (18.0%)	0.4 (15.3%)	30 (67.2%)	4 (26.0%)	(1.6%)
	32 (5.0%)	0.2 (0.0%)	1 (33.2%)	1 (4.2%)	(0.0%)

Note: "BI Time" stands for batched inference time, which calculates the total inference time of a batch of data items. The two values in a cell represent the upper bound and space ratio, respectively.

The *space ratio* (SR) is used to assess the reduction effectiveness of DnnSAT. Suppose that $\mathbf{\Delta}$ and $\mathbf{\Delta}_{\text{DnnSAT}}$ are the original and reduced configuration spaces, respectively. Then, $\text{SR} = \frac{|\mathbf{\Delta}_{\text{DnnSAT}}|}{|\mathbf{\Delta}|} \times 100\%$. A smaller SR means a stronger reduction effect. The promising experimental results in Table III demonstrate that DnnSAT is effective in configuration space reduction. For example, the SR of VGG-16 under the GPU memory consumption constraint ranges from 42.1% to 84.3%. Meanwhile, the SR of Seq2Seq under the batched inference (BI) time constraint ranges from 33.2% to 74.1%. Tighter bounds or a combination of multiple constraints will lead to a more significant reduction. The results can also give hints to developers and help them choose optimal settings of neural architectures, hyperparameters, and computational resources. For instance, the SR of VGG-16 equals 0% when the upper bound of the model weight size is set to 128 MB, which indicates that such a model may not be further dwindled by simply adjusting the hyperparameters. Therefore, developers need to look for advanced DL techniques (*e.g.*, model compression [32]) to embed it into a resource-restricted application. After applying four constraints collectively, the SR value further decreases and is below the minimum of the one-constraint SR values. The results demonstrate the stronger reduction effect when adopting multiple constraints collectively.

DnnSAT runs in both interactive and pruning modes with 12 threads, tiny subspaces containing 10 model configurations, and interval filtering being off. We show the runtime performance of DnnSAT under one constraint and four constraints as follows:

1) Interactive mode. For VGG-16, the solving time per configuration is 0.10s (warm-up: 6.13s) on average under one constraint, and 0.13s (warm-up: 6.18s) under four constraints. For Seq2Seq, the corresponding time is 0.05s (warm-up: 1.40s) on average and 0.12s (warm-up: 1.44s), respectively.
2) Pruning mode. For VGG-16, DnnSAT spends 226s on average under one constraint and 283s under four constraints. For Seq2Seq, the corresponding time is 806s on average and 1301s, respectively.

C. RQ3: How efficient is DnnSAT in constraint solving?

In this section, we evaluate the solving efficiency of the optimization techniques proposed in Section IV-E. We choose the LSTM-based Seq2Seq model with batch size (interval $[1, 4800]$) and hidden size (interval $[16, 20]$) as the hyperparameters. The configuration space then consists of 24000 model configurations in total. To increase the solving difficulty, we use a loose FLOPs constraint under which every configuration satisfies.

A series of experiments are conducted with different thread numbers (1, 4, 8, and 12), subspace sizes (10, 50, 100, 500, 1000, and equipartition), and interval filtering settings (ON and OFF). We do not create more threads since the experimental machine has only 12 vCPUs. "Equipartition" means that we divide the original configuration space equally by the number of threads and do not further split it into tiny subspaces. For example, in the case of 12 threads, each thread independently solves a subspace of 2000 configurations.

Table IV shows the end-to-end execution time (in seconds) and speedup relative to the baseline experiment (1 thread, equipartition, and interval filtering being off). The speedup ranges from 7.6X to 17892.1X, confirming the strong effectiveness of our optimizations. Simply increasing the number of threads achieves an ultra-linear speedup from 9.3X to 51.9X because a smaller configuration space notably reduces the overhead of ALLSAT solving (Section IV-E). Tiny subspaces achieve a speedup from 7.6X to 83.0X in the experiments that turn off interval filtering, with the same reason as the parallel solving. Nevertheless, as the subspace size getting smaller, the speedup grows slowly and then drops (from the size of 50) because the overhead of ALLSAT solving is no longer noticeable while the management cost of tiny subspaces increases. Interval filtering demonstrates dramatic improvements in the equipartition experiments and achieves a maximal speedup of 17892.1X. The reason is that DnnSAT divides the original configuration space on hidden size to keep as long a continuous interval of batch size as possible since FLOPs is monotonically increasing with regard to batch size. Therefore, DnnSAT solves only a small number of configurations to reach the conclusion that the entire subspace satisfies the constraint. However, if there exist many short intervals of batch size (*e.g.*, the domain of batch size is small or tiny subspaces are used), the effect of interval filtering will not be so significant.

TABLE IV
RUNTIME PERFORMANCE OF DNNSAT UNDER THE FLOPS CONSTRAINT WITH OPTIMIZATION TECHNIQUES OF PARALLEL SOLVING, TINY SUBSPACES, AND INTERVAL FILTERING.

Subspace Size	Number of Threads							
	Interval Filtering OFF				Interval Filtering ON			
	1	4	8	12	1	4	8	12
Equipartition	1.0 (19681.0s)	9.3 (2116.2s)	26.7 (737.1s)	51.9 (379.2s)	17892.1 (1.1s)	17,730.6 (1.11s)	17,572.3 (1.12s)	15,496.8 (1.27s)
1000	8.8 (2236.4s)	28.9 (681.0s)	50.3 (391.2s)	68.5 (287.3s)	1640.1 (12.0s)	5046.4 (3.9s)	7872.5 (2.5s)	9840.6 (2.0s)
500	10.5 (1874.3s)	33.7 (584.0s)	57.0 (345.2s)	77.0 (255.6s)	841.0 (23.4s)	2659.6 (7.4s)	4100.2 (4.8s)	5319.2 (3.7s)
100	12.1 (1626.5s)	38.9 (505.9s)	63.2 (311.4s)	83.0 (237.1s)	169.8 (115.9s)	560.7 (35.1s)	882.5 (22.3s)	1063.8 (18.5s)
50	11.6 (1696.6s)	37.6 (523.4s)	60.0 (328.0s)	76.0 (258.9s)	85.2 (231.0s)	258.2 (76.2s)	441.2 (44.6s)	533.3 (36.9s)
10	7.6 (2589.6s)	25.7 (765.8s)	40.0 (492.0s)	48.9 (402.4s)	17.1 (1150.9s)	57.3 (343.4s)	89.5 (219.9s)	107.1 (183.7s)

Note: The two values in a cell represent the speedup and execution time (in seconds).

VI. DISCUSSION

A. Extensibility of DnnSAT

Currently, DnnSAT supports 70+ commonly used operators. DnnSAT is extensible, and users can incorporate new operators and constraints. To add a new operator, users formulate the analytic resource cost functions based on its semantics and then implement the SMT-LIB templates. To support a new constraint, users formulate the analytic restriction function using defined hyperparameters and carry out the above operator-adding steps for each of the operators under consideration. Besides, users may need to reimplement the graph traversal to compute more accurate current resource consumption by employing additional information, including visited operators, edges, and previously calculated resource consumption.

B. Threats to Validity

We discuss the following threats to the validity of our work:

1) **Resource cost functions**. We examine the source code of frameworks to extract the resource cost functions of DL operators for inferring resource usage. However, the implementation of operators can call proprietary NVIDIA CUDA, cuDNN, or cuBLAS APIs, which may introduce some fluctuations in the cost functions. For example, `cudnnConvolutionForward()` could use temporary GPU memory called *workspace* to improve the runtime performance. Nevertheless, the workspace size is convolution algorithm-dependent and thus unpredictable. We mitigated this threat by refining the resource cost functions after carefully referring to the NVIDIA development documentation, dynamically profiling the APIs using NVIDIA *nvprof*, and analyzing the framework runtime logs.
2) **Hidden factors**. There are a number of hidden framework factors that can observably affect the GPU memory consumption and inference time of a DL model. For example, the GPU memory consumption has complicated dependencies on the allocation policy (*e.g.*, tensor fragmentation, alignment, garbage collection, and reservation), internal usage (*e.g.*, CUDA context), operator scheduling, *etc.* To mitigate this threat, we referred to the framework source code carefully to identify hidden factors. However, it is very challenging to directly formulate all such factors (*e.g.*, garbage collection and reservation) analytically. Hence, DnnSAT conservatively calculates the resource usage to reduce the influence of hidden factors. For instance, DnnSAT adopts a simplified yet common approach to accumulate the GPU memory required by each operator under only forward propagation (Section IV-B), which computes a smaller value than the actual GPU memory consumption. If the computed value (a conservative value) already exceeds

the GPU memory upper bound, the corresponding model configuration indeed does not satisfy the constraint. Therefore, valid (satisfiable) model configurations will not be discarded. However, some invalid (unsatisfiable) model configurations may not be eliminated correctly due to the inaccuracy in the calculation of hidden factors. In the experiment on VGG-16 (Section V-B), we notice that on average 9.53% of model configurations are actually invalid under four constraints yet passed DnnSAT, but no valid model configurations are discarded. In the future, we will identify more hidden factors and design more accurate restriction functions to obtain more precise results.

3) **SMT solving**. The simple and non-intrusive integration with existing AutoML tools is to run DnnSAT in the interactive mode. According to the experimental results in Section V, the cost of solving one model configuration is very low. In the pruning mode, it takes a longer time for constraint solving because of the large configuration space. For example, Table IV in SectionV-C shows that DnnSAT spends 19681 seconds (about 5.5 hours) to solve the 24000-configuration space of Seq2Seq. We currently propose some effective optimization techniques in Section IV-E to increase the scalability of constraint solving: (1) DnnSAT supports parallel and distributed solving (via Spark), in which the full configuration space can be partitioned into any number of independent subspaces and solved concurrently by different threads and machines; (2) the use of tiny subspaces reduces the complexity of solving individual subspaces, removes the computation skew across subspaces, balances the workload among threads, and thus avoids stragglers (*i.e.*, threads that take an unusually long time to finish) [40]; (3) interval filtering can further reduce the solving complexity significantly if the restriction function of a constraint is monotonic with regard to some hyperparameters. The experimental results in Table IV confirm the strong effectiveness of our optimizations. However, as the number of hyperparameters and their domains increase, the configuration space can enlarge exponentially, thus the computational complexity may still exceed the capabilities of an SMT solver. In addition to advancing the solvers, it is possible to tackle this problem by trying larger-scale distributed solving with better load balancing.

VII. Related Work

Many software systems are highly configurable by providing a rich set of configuration options. Configuration options are also considered as features in the software product line context. However, it is very time-consuming and error-prone for manual configuration tuning due to a large number of option combinations. Software engineering researchers have proposed various approaches to predicting the performance of configurable systems [44], [46], [47], checking the consistency of configurations [48]–[51], and understanding how configuration options and their interactions influence system performance [52], [53]. Like traditional software systems, DL models are also highly configurable. In this work, we analyze DL models and propose to optimize the configuration exploration (AutoML) through resource-guided space reduction.

Google Vizier [54] and Microsoft HyperDrive [55] are representative AutoML systems, which concentrate more on the system design and operation. Hyperband [2] is an HPO search method and focuses on speeding up random search through adaptive resource allocation and early stopping. ENAS [3] uses a controller to discover various neural architectures and search for the optimal one. These methods and systems are not aware of the constraints imposed by computational resources, which can cause an expensive waste of shared resources. Our work can help them efficiently reduce the configuration space ahead of time and accelerate training.

The authors of [56], [57] analyzed the resource budget constraint for HPO. However, they encoded the budget into the algorithm instead of formulating explicit resource constraints, and thus their method cannot be applied to other AutoML methods directly. Hernández-Lobato *et al.* [58] considered constraints for Bayesian Optimization, but the work is for a specific algorithm. Gordon *et al.* [59] proposed an approach to automate the design of neural structures via a resource-weighted sparsifying regularizer. AMS [60] generated the AutoML search space from an unordered set of API components. Our work formulates common constraints imposed by resources and uses a unified analytic approach to eliminate the unsatisfiable model configurations in advance.

There have been many program analysis methods [61]–[64] to determine the quantitative resource usage (*e.g.*, memory and heap space) of computer programs. For example, Hoffmann *et al.* [61] used the automatic amortized resource analysis (AARA) technique in analyzing the worst-case resource consumption of arbitrary multivariate polynomial functions. Jost *et al.* [64] employed a type-based approach by exploiting linearity and focusing on the reference number to an object. However, such work usually targets higher-order functional programs and cannot be directly applied to DL models because of the wide differences in the representation structures. Our work proposes an analytic and framework-independent cost model to infer the resource consumption of a DL model and utilizes an SMT solver to obtain all the satisfiable model configurations.

VIII. Conclusion

In this paper, we have presented DnnSAT, a resource-guided AutoML approach for deep learning models to efficiently reduce the configuration space under computational constraints. Powered by DnnSAT, we demonstrate that commonly used AutoML methods can efficiently prune unsatisfiable model configurations ahead of time to avoid unnecessary training costs and achieve significant speedups. We believe that DnnSAT can make AutoML more practical in a real-world environment with constrained resources.

References

[1] NNI, "Nni (neural network intelligence): a lightweight but powerful toolkit to help users automate feature engineering, neural architecture

search, hyperparameter tuning and model compression." https://github.com/microsoft/nni, 2019.

[2] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6765–6816, Jan. 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3122009.3242042

[3] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1802.03268

[4] H. Jin, Q. Song, and X. Hu, "Efficient neural architecture search with network morphism," *CoRR*, vol. abs/1806.10282, 2018. [Online]. Available: http://arxiv.org/abs/1806.10282

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[6] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating gpu memory consumption of deep learning models," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1342–1352. [Online]. Available: https://doi.org/10.1145/3368089.3417050

[7] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1159–1170. [Online]. Available: https://doi.org/10.1145/3377811.3380362

[8] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *CoRR*, vol. abs/1812.00332, 2018. [Online]. Available: http://arxiv.org/abs/1812.00332

[9] C. Hsu, S. Chang, D. Juan, J. Pan, Y. Chen, W. Wei, and S. Chang, "MONAS: multi-objective neural architecture search using reinforcement learning," *CoRR*, vol. abs/1806.10332, 2018. [Online]. Available: http://arxiv.org/abs/1806.10332

[10] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 6105–6114. [Online]. Available: http://proceedings.mlr.press/v97/tan19a.html

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[12] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08/ETAPS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[13] A. Klein and F. Hutter, "Tabular benchmarks for joint architecture and hyperparameter optimization," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1905.04970

[14] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1902.09635

[15] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012. [Online]. Available: http://jmlr.org/papers/v13/bergstra12a.html

[16] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1802.01548

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[19] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 3104–3112.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, pp. 8026–8037. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[23] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[24] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[25] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015, pp. 2962–2970. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf

[26] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *CoRR*, vol. abs/1611.01578, 2016. [Online]. Available: http://arxiv.org/abs/1611.01578

[27] R. M. Larsen and T. Shpeisman, "Tensorflow graph optimizations," 2019.

[28] PyTorch, "The topological sorting algorithm for computation graphs in pytorch," https://github.com/pytorch/pytorch/blob/v1.2.0/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h#L26, 2019.

[29] A. MXNet, "The topological sorting algorithm for computation graphs in apache mxnet," https://github.com/apache/incubator-mxnet/blob/4149f8b8752989fce5d80cc13f92d99774988b4f/src/executor/simple_partition_pass.h#L67, 2019.

[30] H. B. Enderton, *Elements of Set Theory*, H. B. Enderton, Ed. San Diego: Academic Press, 1977.

[31] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[32] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[33] R. Tang, W. Wang, Z. Tu, and J. Lin, "An experimental analysis of the power consumption of convolutional neural networks for keyword spotting," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 5479–5483.

[34] PyTorch, "Pytorch conv2d api," https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.

[35] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proceedings of the International Conference on Learning Representations*, 2017.

[36] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 41–53. [Online]. Available: https://doi.org/10.1145/3178487.3178491

[37] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine

for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2934664

[38] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 287–300. [Online]. Available: https://doi.org/10.1145/1966445.1966472

[39] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 25–36. [Online]. Available: https://doi.org/10.1145/2213836.2213840

[40] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. Santa Ana Pueblo, NM: USENIX Association, May 2013. [Online]. Available: https://www.usenix.org/conference/hotos13/session/ousterhout

[41] T. Toda and T. Soh, "Implementing efficient all solutions SAT solvers," *CoRR*, vol. abs/1510.00523, 2015. [Online]. Available: http://arxiv.org/abs/1510.00523

[42] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: https://doi.org/10.1145/324133.324234

[43] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[44] H. Ha and H. Zhang, "Deepperf: Performance prediction for configurable software with deep sparse neural network," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 1095–1106. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00113

[45] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling kb-sized machine learning models to tiny iot devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 79–95. [Online]. Available: https://doi.org/10.1145/3314221.3314597

[46] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 167–177.

[47] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Softw. Engg.*, vol. 23, no. 3, pp. 1826–1867, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9573-6

[48] D. Batory, D. Benavides, and A. Ruiz-Cortes, "Automated analysis of feature models: Challenges ahead," *Commun. ACM*, vol. 49, no. 12, pp. 45–47, Dec. 2006. [Online]. Available: https://doi.org/10.1145/1183236.1183264

[49] J. Sun, H. Zhang, Y. Fang, and L. Wang, "Formal semantics and verification for feature modeling," in *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, June 2005, pp. 303–312.

[50] K. Czarnecki and C. Kim, "Cardinality-based feature modeling and constraints : A progress report," 2005.

[51] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, pp. 615–636, 09 2010.

[52] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 284–294.

[53] H. Ha and H. Zhang, "Performance-influence model for highly configurable software with fourier learning and lasso regression," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 470–480. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00080

[54] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1487–1495. [Online]. Available: https://doi.org/10.1145/3097983.3098043

[55] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca, "Hyperdrive: Exploring hyperparameters with pop scheduling," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3135974.3135994

[56] M. Li, E. Yumer, and D. Ramanan, "Budgeted training: Rethinking deep neural network training under resource constraints," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=HyxLRTVKPH

[57] Z. Lu, L. Chen, C.-K. Chiang, and F. Sha, "Hyper-parameter tuning under a budget constraint," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 5744–5750. [Online]. Available: https://doi.org/10.24963/ijcai.2019/796

[58] J. M. Hernández-Lobato, M. A. Gelbart, R. P. Adams, M. W. Hoffman, and Z. Ghahramani, "A general framework for constrained bayesian optimization using information-based search," *J. Mach. Learn. Res.*, vol. 17, pp. 160:1–160:53, 2016.

[59] A. Gordon, E. Eban, O. Nachum, B. Chen, T. Yang, and E. Choi, "Morphnet: Fast & simple resource-constrained structure learning of deep networks," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1711.06798

[60] J. P. Cambronero, J. Cito, and M. C. Rinard, "Ams: Generating automl search spaces from weak specifications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 763–774. [Online]. Available: https://doi.org/10.1145/3368089.3409700

[61] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 3, Nov. 2012. [Online]. Available: https://doi.org/10.1145/2362389.2362393

[62] J. Hoffmann, A. Das, and S.-C. Weng, "Towards automatic resource bound analysis for ocaml," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 359–373. [Online]. Available: https://doi.org/10.1145/3009837.3009842

[63] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 185–197. [Online]. Available: https://doi.org/10.1145/604131.604148

[64] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann, "Static determination of quantitative resource usage for higher-order programs," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 223–236. [Online]. Available: https://doi.org/10.1145/1706299.1706327