ModularNAS: Towards Modularized and Reusable Neural Architecture Search

Yunfeng Lin [1] Guilin Li [2] Xing Zhang [2] Weinan Zhang [1] Bo Chen [2] Ruiming Tang [2] Zhenguo Li [2]
Jiashi Feng [3] Yong Yu [1]

Abstract

Automated neural architecture search (NAS) methods have been demonstrated as a powerful tool to facilitate neural architecture design. However, the broad applicability of NAS has been restrained due to the difficulty of designing task-specific search spaces and the necessity and verbosity to implement every NAS component from scratch when switching to another search space. In this work, we propose ModularNAS, a framework that implements essential components of NAS in a modularized and unified manner. It enables automatic search space generation for customized use cases while reusing predefined search strategies, with little extra work needed for each case. We conduct extensive experiments to verify the improved model performance on various tasks by reusing supported NAS components over customized search spaces. We have also shown that targeting existing architectures, ModularNAS can find superior ones concerning accuracy and deployment efficiency, such as latency and FLOPS. The source code of our framework can be found at `https://github.com/huawei-noah/vega/tree/master/vega/algorithms/nas/modnas`.

1 Introduction

Deep learning models often require extensive manual design to achieve high accuracy and efficiency for specific tasks. Recent progress in neural architecture search (NAS) has discovered architectures with superior performance over handcrafted ones in various tasks (Cai et al., 2019; Yang et al., 2019; Liu et al., 2019a; 2020).

Significant efforts have been made to improve the efficiency of NAS. Early approaches (Zoph & Le, 2016; Tan et al., 2018) train each sampled architecture independently from scratch, which is computationally prohibitive when searching over large datasets. Recent methods (Pham et al., 2018; Liu et al., 2019b; Li et al., 2019; Yang et al., 2019) adopt weight sharing strategies which are much more feasible computationally and achieve competitive accuracy. Other efficient methods include network morphism (Chen et al., 2016; Yang et al., 2018; Cai et al., 2018a) and performance prediction (Baker et al., 2017; 2018).

[1]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China [2]Huawei Noah's Ark Lab, Shenzhen, China [3]Department of Electrical and Computer Engineering, National University of Singapore, Singapore. Correspondence to: Jiashi Feng <elefjia@nus.edu.sg>, Weinan Zhang <wnzhang@sjtu.edu.cn>, Yong Yu <yyu@apex.sjtu.edu.cn>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

Despite the effectiveness of the existing NAS methods, the current implementations of these methods are often case-specific. Early NAS methods can be formulated as hyperparameter optimization at the cost of efficiency. However, efficient NAS methods, including weight sharing and network morphism, require access to runtime network weights and graph topology, thus relying on architecture-specific code to work. Consequently, their implementations intertwine with the specific search space, resulting in considerable engineering work for adapting a method to new search space. This is highly *non-automatic*, and the broad application of NAS in user-defined tasks has been heavily restrained due to the verbosity of designing architecture-specific search space and implementing the NAS workflow from scratch.

Therefore, the situation calls for a code framework that provides a common interface to NAS methods with different strategies and formulations, such that their implementations become search space agnostic under the framework. Moreover, the framework should provide an easy way to define the search space for different use cases.

To provide a fundamental basis in designing a common interface for various NAS methods, we propose a new formulation of the architecture search process that breaks down an extended range of well-recognized NAS methods into combinations of the fundamental components, with emphasis on the transformation of architecture states (e.g., weights of evaluated networks), which usually takes form in weight sharing (Wu et al., 2019; Guo et al., 2019) and network morphism (Cai et al., 2018a; Yu & Huang, 2019). Based on such a formulation, we build **ModularNAS**, an automated neural architecture search framework that implements the state-of-the-art efficient NAS methods as a combination of reusable components (e.g., search algorithm, network transformation, evaluation strategy) with unified interfaces, enabling easy integration with different search strategies with simplicity and flexibility. Our framework provides easy-to-use utilities that convert fixed architecture given by the user to customized search space in runtime, facilitating the design and use of efficient NAS methods for user-defined tasks in just a few lines of code.

Our contributions can be summarized as follows:

- We design a user-friendly NAS framework called ModularNAS, based on our proposal for a unified formulation of NAS. This framework supports a wide range of popular NAS methods with different paradigms (such as black-box optimization, weight sharing, and network morphism).
- We improve the implementation reusability of essential NAS components by decoupling the search spaces, optimization algorithms, network transformations, and evaluation strategies while unifying the interactions between these components through an event mechanism, enabling easy integration of different NAS methods.
- We automate the generation of task-specific search space by dynamically replacing modules of a given network with modularized search space components such as different neural operators or dynamic network width and depth.
- We conduct comprehensive experiments to demonstrate that ModularNAS can be efficiently utilized to carry out architecture search for various deep learning applications, including image classification, speech recognition and recommender systems, improving the performance of given architectures in each task without re-implementing the components used.

2 Methodology

2.1 Classical NAS Formulation

Early architecture search methods can be formulated as a hyperparameter optimization problem with a large search space (Zoph & Le, 2016). A controller samples a child network (sub-net) in each step, which is trained from scratch and evaluated to obtain the performance as the controller's reward signal, without adopting any efficient strategy in architecture evaluation. Such methods are very time-consuming, often requiring hundreds of GPU hours to converge.

Table 1. Overview of efficient strategies in NAS

Efficient strategy	References
None	(Zoph & Le, 2016) (Real et al., 2018) (Tan et al., 2018)
One-shot (discrete)	(Pham et al., 2018) (Guo et al., 2019) (Yang et al., 2019)
Network morphism (path)	(Cai et al., 2018a) (Cai et al., 2018b)
One-shot (softmax sum) One-shot (gumbel sum) One-shot (binarized)	(Liu et al., 2019b) (Xie et al., 2019) (Cai et al., 2019)
Network morphism (elastic)	(Yu & Huang, 2019) (Guo et al., 2019) (Cai et al., 2020)

Several weight sharing methods are proposed to speedup architecture evaluation, where a one-shot network (supernet) that contains all candidate paths is trained. A child network is obtained by selecting the corresponding paths in the one-shot model, controlled by extra parameters that are either discrete (Pham et al., 2018; Guo et al., 2019) or differentiable (Liu et al., 2019b; Cai et al., 2019).

In other efficient NAS methods, a child network inherits its weights from the parent model, through network transformations that retain the weights of the parent model, while modifying its architecture. Typical transformations include adding/removing graph nodes (Cai et al., 2018a;b) and changing the depth/width of the network (Chen et al., 2016; Yang et al., 2018; Cai et al., 2020). Table 1 summarizes various efficient strategies and related works.

2.2 A Unified Formulation of NAS

We observe that most efficient NAS methods use the intermediate architecture states (e.g., network weights, graph topology) in a search process. For example, weight sharing methods reuse the weights of common nodes in the one-shot model. If we leave out the parts of a NAS method that require access to the architecture states, the remaining parts become architecture-independent and can be used in arbitrary search spaces. Therefore, we can divide NAS methods into separate functional components.

Now we present a new formulation of architecture search procedure that unifies most well-recognized NAS methods, including weight sharing (Liu et al., 2019b) and network morphism (Chen et al., 2016; Yang et al., 2018; Cai et al., 2020). To illustrate the idea, we first define the following notations representing the essential elements in a NAS process, as displayed in Table 2.

We divide the variables involved in a NAS process into three fundamental types: the *architecture parameter* α represents the encoding of all possible architectures in the search space;

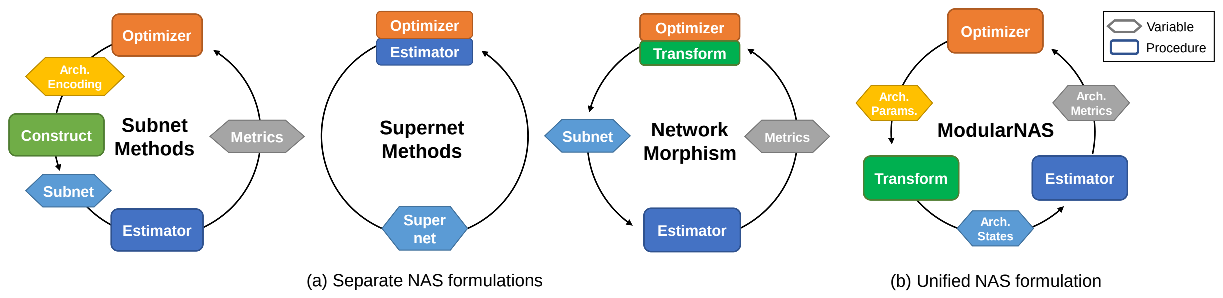

Figure 1. Comparison of NAS formulations. Rectangles denote components, and hexagons denote variables.

Table 2. Notations for the unified formulation of NAS.

Notation	Description
$\mathcal{A}$	the architecture parameter space
$\mathcal{V}$	the architecture state space (e.g., model weights and graph topology)
$\mathcal{R}$	the architecture metrics space (e.g., model accuracy and latency)
$\alpha \in \mathcal{A}$	a set of parameter values that encode an architecture
$r \in \mathcal{R}$	the metrics values of an architecture
$v \in \mathcal{V}$	intermediate architecture states in a NAS process
$\omega : \mathcal{V} \mapsto \mathcal{V}$	state optimizer: update the arch. states (e.g., train the model)
$\eta : \mathcal{V} \mapsto \mathcal{R}$	state evaluator: evaluate the arch. states (e.g, validate the model)
$\Omega : \mathcal{A} \times \mathcal{R} \mapsto \mathcal{A}$	parameter optimizer: update the architecture parameters
$\delta : \mathcal{V} \times \mathcal{A} \mapsto \mathcal{V}$	state transformer: alter the arch. states according to the updated α
$\alpha_0 \in \mathcal{A}$	the initial architecture parameter
$v_0 \in \mathcal{V}$	the initial architecture state
$\mathcal{H} \subset \mathcal{V}$	the set of final architectures, defines the stopping criterion

v stands for the *states* generated in a search process for architecture updating and evaluating, such as network weights and topology; the *metrics* r indicate the estimated performance scores of the candidate architecture.

We then decompose the operators of a NAS process into four components that function separately: the *search optimizer* Ω optimizes the architecture parameters; the *state optimizer* ω updates the architecture states; the *state evaluator* η evaluates the architecture metrics by using the updated states; the *state transformer* function δ modifies the architecture states as controlled by the parameters. The unified formulation of the architecture search process is summarized in Algorithm 1.

A critical insight in our formulation is that architecture states serve as the intermediate information between search optimization and architecture evaluation, which is used in

Algorithm 1 Unified Architecture Search Process

```
Input: initial architecture parameters and states $\alpha_0, v_0$
Output: best parameter searched $\alpha^*$
  $\alpha, v \leftarrow \alpha_0, v_0$
  while $v \notin \mathcal{H}$ do
    $v \leftarrow \omega(v)$                ▷ Update arch. states
    $r \leftarrow \eta(v)$                  ▷ Evaluate arch. metrics
    RECORD$(\alpha, r)$                     ▷ Save visited arch. and metrics
    $\alpha \leftarrow \Omega(\alpha, r)$   ▷ Optimizer samples new params.
    $v \leftarrow \delta(v, \alpha)$        ▷ Transform arch. based on $\alpha$
  return $\alpha^*$ with the best metrics $r^*$
```

Figure 2. Pseudo-code for the unified architecture process. In each loop step, a new set of *architecture parameters* α is sampled by *optimizer* Ω, used by *state transformer* δ to modify the *architecture state* v and obtain the desired architecture. The transformed state is then updated (trained) and evaluated by ω and η. The evaluated *metrics* r is used to update the optimizer and sample the next set of parameters.

architecture transformation for mapping the sampled parameters to the corresponding architectures for evaluation.

Despite its simplicity, the unified formulation can cover a full spectrum of search algorithms, including the subnet, supernet, and morphism based algorithms, through different instantiations and combinations of essential elements, as illustrated in Figure 1. In the following part, we elaborate on the realization of several typical types of search algorithms in our framework. The formulations of more NAS methods under our framework are presented in Appendix A.

Subnet based search. Subnet based methods view the architecture search as an optimization problem over a discrete parameter space and retrain each candidate architecture from scratch before evaluation (Zoph & Le, 2016; Tan et al., 2018; Real et al., 2018; Wang et al., 2018). Therefore, the *state transformer* function δ constructs the child architecture according to the architecture parameters regardless of previous architectures, i.e., $\delta(v, \alpha) = \text{CONSTRUCT}(\alpha)$. The state optimizer ω fully trains the architecture and the state evaluator η evaluates the metrics used by the optimization function Ω to update the architecture parameters.

Supernet based search. For methods that share weights between child networks with common nodes in a supernet that comprise all paths (Liu et al., 2019b; Bender et al., 2018; Cai et al., 2019; Xie et al., 2019; Wu et al., 2019; Liu et al., 2018; Li et al., 2019; Guo et al., 2019; Pham et al., 2018), the δ function selects the corresponding paths specified by α. The ω and η trains and evaluates the supernet, respectively. The Ω updates the parameters using the gradients or reward signals derived from the metrics.

Network morphism. We identify some proposed search processes as somewhere between the subnet based search and supernet based search, which do not train each subnet from scratch or train a supernet but partially reuse the weights of the current architecture through function-preserving network transformations (Chen et al., 2016; Cai et al., 2018a;b; Yang et al., 2018; Stamoulis et al., 2019; Guo et al., 2019; Yu & Huang, 2019; Cai et al., 2020). In this case, the δ function transforms the network according to the architecture parameters, typically by altering the computational graph topology and node attributes.

3 MODULARNAS: FRAMEWORK DESIGN AND DYNAMICS

Following the unified formulation, we design and implement ModularNAS with simplicity, modularity, and reusability in mind. Specifically, we introduce a new programming paradigm for defining the architecture search space, enabling the automatic generation and reuse of the search space. To support complex NAS methods such as network morphism, we implement the architecture transformation functionality that binds with the generated search space and is automatically invoked in the search routine.

3.1 Design Overview

We first declare the components and interfaces of the proposed framework in alignment with the unified NAS formulation. Then we specify the search routine in the framework that invokes these components using standard interfaces, which enables seamless switching between NAS methods and search spaces. The search routine of the framework is illustrated in Figure 3.

Parameter space. The *Parameter space* is the set of all architecture parameters, which corresponds to $\mathcal{A}$ in the above formulation. An *architecture parameter* is a named dimension of values, which corresponds to α.

Optimizer. The *Optimizer* defines the search algorithm that tries to find the optimal set of architecture parameters. It uses the architecture metrics provided by the *Estimator* to update its states and sample new parameters, as described in Ω.

Estimator. Analogous to ω and η, the *Estimator* defines the estimation strategy for updating the architecture and evaluating its metrics, such as training the model specified by the parameters and measuring its performance. Besides, the *Estimator* also schedules the search process, controls the interaction with the *Optimizer*, and records information such as metrics of explored architectures.

Construct routine. The *Construct* routine declares new architecture parameters and defines their mappings to the candidate architectures during the generation of search space. It provides several ways to generate the parameter space $\mathcal{A}$ and architecture state space $\mathcal{V}$, either from scratch, predefined backbone architectures, or based on previous search results.

Transform routine. The *Transform* routine modifies the architecture according to the parameter values and its current states via method-specific handler functions. It corresponds to δ in the formulation.

The *Optimizer* draws a new set of parameter values from the *Parameter Space* according to its algorithm in each step. The *Transform* routine alters the current architecture to the one encoded by the updated values. The *Estimator* evaluates the architecture with specific performance metrics. The *Optimizer* uses the evaluation metrics to guide its direction.

We refer the reader to Appendix C for details on the interfaces of the framework components, and Appendix B for the standard framework routines.

3.2 Automatic Search Space Generation

ModularNAS supports automatically generating the search space from customized architecture by replacing the *Stub modules* (inserted as placeholders) with actual modules specified in the candidate set. By using Stub modules as the anchor points in the architecture, the *Construct routine* instantiates and combines the *Architecture candidates* with the user-defined architecture to assemble the desired search space.

Stub module. The Stub module is a placeholder module in the user-defined architecture. It saves the hyperparameters and arguments required to build the original network module in its position, such as the shape of input and output data, and strides. The information is used in the Construct routine to determine the correct form of candidate modules.

For architecture with undecided modules to be searched, instead of binding the candidate modules with the architecture immediately, we replace the original modules with Stub modules, which will be converted to the desired set of candidate modules automatically through Construct and

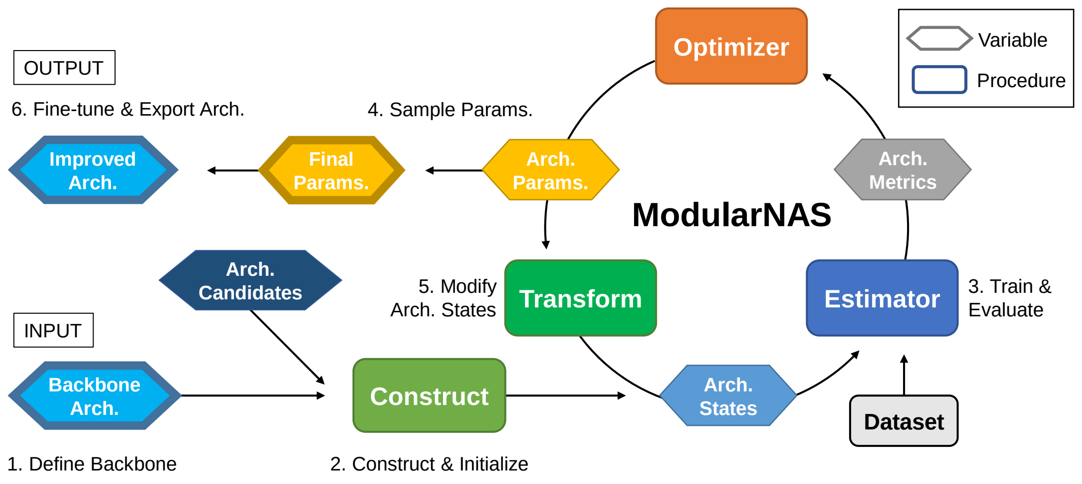

Figure 3. Illustration of previous NAS methods and the unified search routine in ModularNAS. In each cycle of the unified routine, the Optimizer samples parameter values which are used in the Transform routine to alter the architecture states based on previous states. The Estimator computes the architecture performance metrics which are used by the Optimizer to decide future search steps. The best parameters and the corresponding architecture are exported when the search ends.

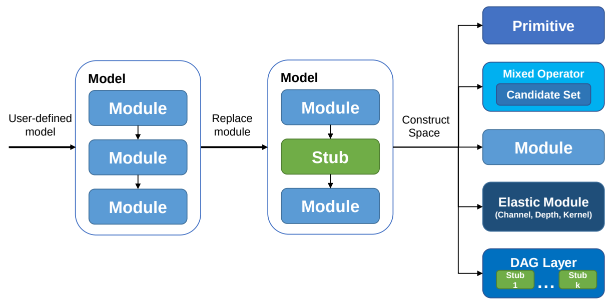

Figure 4. Examples of search space generation using the Stub module and architecture candidates. The original module in the architecture (Module) is replaced with a Stub module. It is then converted to one of the possible candidate modules: Primitives, Mixed Operators, the original module, Elastic Modules with dynamic transformations, and Layers with possibly more Stub modules.

Transform routine. By introducing the Stub module, we semantically separate the search space specifications relevant to the macro architecture, and those determined only by the search methods and architecture candidates.

Architecture candidates. In architecture search setups, the candidate set often consists of basic network operators and repeating building blocks, and may contain overlapping elements between search scenarios. To reduce the redundant definition of these candidates and reuse them across use cases, we define some frequently used network modules as standard architecture candidates, including basic neural operators like convolution and network building blocks like the directed acyclic graph (DAG) (Liu et al., 2019b). An architecture candidate can be a neural operator, a mixed operator, a layer with more Stub modules, a group of modules with dynamic width and depth, or the original network module, as shown in Figure 4. Detailed definitions and descriptions of the architecture candidates can be found in Appendix D.

Default Construct routine. For many search methods, a default Construct routine is used to generate the search space. It iterates the Stub modules defined earlier in the macro architecture, substitutes each with candidate modules, and assigns an architecture parameter representing the choice of candidate architectures, with additional handler functions that control architecture transformation if required. The resulting architecture parameter space is an abstraction of the architecture search space, i.e., the set of all possible architectures. The mapping between the parameter space and architecture space is appropriately defined and carried out in the Transform routine. Details on the default Construct routine can be found in Appendix B.

3.3 Architecture Transformation

To begin the search process, the *Optimizer* samples a new set of the architecture parameters from the generated search space in the previous subsection. The architecture transformation takes place upon parameter updates, which alters the current network's weights or topology. While early NAS methods simply define this behavior as the complete reconstruction of the architecture, new methods propose more complex transformations such as weight sharing, weight inheritance, and network morphism.

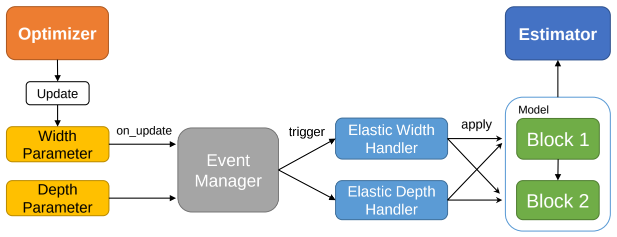

Figure 5. Example of transformation routine for the OFA (Cai et al., 2020) network morphism strategy. The Optimizer samples new parameters specifying the width and depth of the network. The Event Manager receives parameter update event and invokes the corresponding handler functions to transform the network blocks to target widths and depths.

Transform handler. We notice that transformations are often local to the mutable parts of the architecture. For example, the weight inheritance technique applies to each shared computation graph node of the child architectures separately. Therefore, we implement the Transform functionality as invocations of algorithm-specific functions called handlers that hook onto each architecture parameter.

Transform routine. The Transform routine works by triggering handler functions defined in the Construct step with an event-based mechanism. When the value of a parameter is updated, a parameter update event is emitted, triggering the handler functions that watch this event, which carry out the architecture transformations. Figure 5 shows an instance of Transform routine implementing the network transformations specified in OFA (Cai et al., 2020), where updating architecture parameters representing the network depth and width triggers the handlers that transform the current network to the desired dimensions. Details on the Transform routine are presented in Appendix B.

Formally, let p_i be an architecture parameter with name n_i, and H_k is the handler function that listens on the condition c_k that is triggered by the update event u_i of parameter p_i. When the value of p_i changes from v_i to v'_i, then u_i is dispatched, invoking H_k with argument p_i and v'_i. Apart from watching for a single update, the handler can also be triggered by updates of multiple parameters. This enables global transformation of the architecture states, such as re-initializing the network weights which is the default setting in some methods. For example, a handler that listens on any of several parameters updates is triggered by the condition $c = (u_i \vee u_j \vee u_k)$, and the condition that triggers upon all updates is $c = (u_i \wedge u_j \wedge u_k)$.

3.4 Architecture estimation

Once the network is obtained through transformation, the *Estimator* may call a user-defined Trainer to train and evaluate the network according to the evaluation strategy. The *Estimator* then computes the *Metrics* that reflect the architecture's desired property, such as performance, latency, and energy consumption. The values of the *Metrics* may further be used in a *Criterion*, which computes the loss w.r.t. the architecture parameters if they are differentiable. Details on the *Metrics* and *Criterion* modules can be found in Appendix E and Appendix F, respectively.

3.5 Additional features

We implement additional functionalities using standard framework interfaces to improve the flexibility and scalability of ModularNAS.

Hyperparameter tuning. We implement the hyperparameter tuning process using the same interface as in the architecture search process. Hyperparameters can be configurations of framework routines or arguments of arbitrary external programs. The Estimator in a tuning process evaluates the parameters by running the target process and takes its output as metrics.

Process Pipeline. We allow chaining search, training, and hyperparameter tuning processes to form an pipeline. Each process in the pipeline may depend on the outputs of other processes. A scheduler executes the processes in topological order of their dependencies. Pipelines may be nested, with some sub-processes being pipelines themselves.

Distributed search support. We implement distributed support for discrete NAS methods, where the main Estimator distributes the architecture evaluation tasks to multiple remote Estimators and collects the results for the Optimizer.

Hardware-aware NAS support. To measure the hardware performance such as latency and energy consumption, we implement a pipeline of hardware performance measurement as a chain of architecture Metrics. A typical example of pipeline is illustrated in Figure 6.

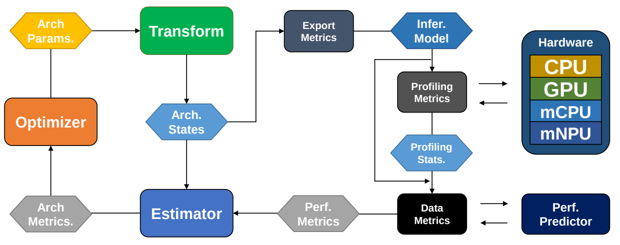

Figure 6. Illustration of hardware performance measurement pipeline. The model specified by the current architecture states is exported for inference on the target hardware to measure its computing properties (Tan et al., 2018). Alternatively, a performance predictor (Wu et al., 2019) may be used to approximate the hardware metrics. The measured data is returned to the Estimator as Metrics results.

4 Example search space

To help illustrate the proposed framework's usability and modularity, we show an example code of search setup that utilizes the framework implementation components. The backbone architecture is defined using PyTorch (Paszke et al., 2019). Suppose we want to find the optimal kernel size of a convolution layer in a ConvNet from the range of $\{3, 5, 7\}$. First, we replace the convolution with a Stub module (called Slot in the framework API), while keeping all the initial arguments.

```
1 # op = Conv2d(chn_in, chn_out, stride)
2 op = Slot(chn_in, chn_out, stride)
3 def handler(v):
4   op.set_entity(Conv2d(chn_in, chn_out, stride,
      kernel_size=v))
5 p = Categorical([3, 5, 7], on_update=handler)
```

Figure 7. Example code for simple search space definition.

In our framework, there are two basic ways to support this setup. The simplest way is to replace the Stub module with convolutions of different kernel sizes on each parameter update, as shown in line 4 of the example code in Figure 7.

```
from modnas.registry.arch_space import build
# op = Conv2d(chn_in, chn_out, stride)
op = Slot(chn_in, chn_out, stride)
def handler(v):
  op.set_entity(build(v))
p = Categorical(['NC3', 'NC5', 'NC7'], on_update=
    handler)
```

Figure 8. Example code for predefined candidate set.

Another solution is to use the predefined candidate sets in the framework. The framework contains out-of-the-box candidate architectures such as operator primitives, layers, and several mixed operators described in several NAS literature. The example code is presented in Figure 8. We use identifiers to refer to the registered architecture candidates, where "NC3" stands for regular convolution with 3x3 kernel.

When using mixed operators, the transform handler is no longer required as the mixed operators select the correct candidates in the model's forward pass. The explicit declaration of architecture parameter is also omitted since it is already defined within each mixed operator. This makes the search space definition even more straightforward, with only the declaration of Stub modules present, as shown in Figure 9. The default Construct behavior converts the Stub modules to mixed operators with specified candidates.

Once the search space is defined, a search procedure can be set up from a YAML configuration file containing the settings for all NAS components used in the search. A wrapper function initializes each component from its configuration

```
# op = Conv2d(chn_in, chn_out, stride)
op = Slot(chn_in, chn_out, stride)
"""
YAML config:
mixed_op:
  primitives: ['NC3', 'NC5', 'NC7']
"""
```

Figure 9. Example code for using the mixed operator.

and starts the search routine. Switching between different components is done by simply changing the configuration. Figure 10 shows a typical example of a search configuration.

```
data:
  type: ImageNet
estim:
  train:
    type: DefaultEstim
    epochs: 120
  search:
    type: SubNetEstim
    epochs: 1000
trainer:
  default:
    type: ImageClsTrainer
optim:
  type: RandomSearchOptim
mixed_op:
  primitives: ['NC3', 'NC5', 'NC7']
model:
  type: MobileNetV2
  args:
    num_classes: 1000
```

Figure 10. Example YAML configuration file for a search procedure, where the random search is applied on the MobileNetV2 (Sandler et al., 2018) search space using SPOS (Guo et al., 2019) estimation strategy (training hyperparameters are omitted for brevity). The evolution algorithm can be used instead by changing the value "RandomSearchOptim" to "EvolutionOptim".

5 Evaluations

This section evaluates the modularized NAS implementations, hardware efficiency support, and framework usability in ModularNAS through comprehensive experiments.

5.1 Experiments on Modularized NAS Methods

To showcase the ability to reuse the implementations of architecture search space, search algorithm, and evaluation strategies using ModularNAS, we present a series of experiments on combinations of several search spaces, search algorithms, estimation strategies, and datasets. An extended list of combinations and results can be found in Appendix H.

Image classification task. We select several image classification datasets such as CIFAR-10 (Krizhevsky, 2009), CIFAR-100 (Krizhevsky, 2009), and ImageNet (Deng et al., 2009) as benchmark datasets.

Table 3. Search results on the image classification task.

Dataset	Arch. Space	Top-1 (%)	Search Method
C10	Cell[1]	97.20	DARTS (Liu et al., 2019b)
		97.03	Proxyless-G (Cai et al., 2019)
		96.80	SNAS (Xie et al., 2019)
		97.45	**StacNAS** (Li et al., 2019)
C100	Cell	79.38	DARTS
		80.79	DARTS (one-level)
		79.96	Proxyless-G
		80.29	Proxyless-GU[2]
		80.76	SNAS
		81.36	SNAS (bi-level)
		83.25	**StacNAS**
	MbV2	72.64	Original
		71.91	DARTS
		72.02	DARTS (one-level)
		72.26	Proxyless-G
		72.45	Proxyless-GU
		72.32	CARS (Yang et al., 2019)+Random
		73.24	**SPOS (Guo et al., 2019)+Random**
ImgNet	MbV2	72.49	Original
	MbV2-L[3]	74.25	Baseline
		75.61	Proxyless-G
		75.62	Proxyless-GU
		75.16	DARTS
		75.75	**DARTS(one-level)**
		74.93	SPOS (Guo et al., 2019) + Random
		75.00	SPOS + RE (Real et al., 2018)
		73.58	Random[4]
	Res-50	76.97	Original
		76.78	DARTS (one-level)
		77.09	DARTS
		76.74	Proxyless-G
		76.71	Proxyless-GU

[1] The search space follows DARTS (Liu et al., 2019b) CIFAR-10 settings.
[2] Uniformly sampling candidate paths in the model forward pass.
[3] The search space follows ProxylessNAS (Cai et al., 2019) ImageNet settings.
[4] Averaged results from 8 architectures sampled from the search space.

Voice recognition task. We use a customized voice recognition dataset containing extracted voiceprint features of human audio samples recorded in different scenarios. The task is to classify each voiceprint sample into three categories based on the speaker's characteristics.

Recommendation task. We choose Avazu[1], a Click-Through Rate (CTR) prediction challenge on Kaggle, as the recommendation task.

NAS Benchmarks. We evaluate several discrete search algorithms on NAS benchmark datasets such as NAS-Bench101 (Ying et al., 2019), which contains all architectures' evaluation metrics in a fixed search space.

[1]https://www.kaggle.com/c/avazu-ctr-prediction

Table 4. Search results on the voice recognition task.

Arch. Space	Top-1 (%)	Search Method
Res-34	96.58	Original
Res-40	96.90	
Pym-34 (Han et al., 2017)	96.94	
Res-34	**97.11**	**DARTS**
	97.05	Proxyless-G
	96.99	SPOS+RE
	96.87	CARS+RE

Table 5. Search results on the recommendation task.

Arch. Space	AUC	Search Method
DeepFM (Guo et al., 2017)	0.7833	Original
DCN (Wang et al., 2017)	0.7823	
Multi-branch	0.7832	Random[1]
	0.7862	**Subnet (Zoph & Le, 2016) + Random**
	0.7836	DARTS
	0.7823	Proxyless-G

[1] Averaged results of 6 architectures sampled from the search space.

Search space. For the image classification task, apart from the cell-based search space frequently used in various NAS literature (Zoph & Le, 2016; Liu et al., 2019b; Xie et al., 2019), we also use the ResNet50 (He et al., 2016) and MobileNetV2 (Sandler et al., 2018) as the macro architecture to generate the search spaces. For MobileNetV2 architecture, we follow ProxylessNAS (Cai et al., 2019) and search for kernel size and expansion ratio of the convolutions in each residual block. For the ResNet search space, the candidate set is defined as convolutions with kernel size ranging from $\{3, 5, 7\}$. For the voice recognition task, ResNet-34 is used as the macro architecture, and variants of ResNet basic blocks are chosen as candidates. We use a multi-branch network for the recommendation task, and search for feature interaction layers, including inner product (Qu et al., 2019), outer product, and MLPs. Detailed descriptions of

Table 6. Search results on the NASBench101 dataset.

Top-1 (%)	Search Method
94.47	NRE (Real et al., 2017)
94.47	RE (Real et al., 2018)
94.23	XGBoost (Chen & Guestrin, 2016) + SA (Pincus, 1970)
94.18	MLP (Pedregosa et al., 2011) + SA
93.95	Linear (Pedregosa et al., 2011) + SA
94.11	XGBoost + Random Sampling
93.85	Random

the search spaces can be found in Appendix G.

Search algorithms. We select several well-recognized gradient-based and discrete search algorithms for reproducing and benchmarking. RE refers to Regularized Evolution (Real et al., 2018), and NRE refers to normal evolution (Real et al., 2017). SA stands for Simulated Annealing (Pincus, 1970) for finding the optimum in the surrogate models, which can be Gradient Boosting Decision Tree (XGBoost) (Chen & Guestrin, 2016), Multi-layer perceptron (MLP) (Pedregosa et al., 2011) and Linear regression (Pedregosa et al., 2011), respectively. RS refers to random search. Additional descriptions of supported discrete search algorithms can be located in Appendix G.3.

Evaluation strategies. A subset of training data (50000 samples for ImageNet, 20% for CIFAR and other tasks) is held out as the validation set for search algorithms that employ bi-level optimization. A default one-shot training strategy is used for gradient-based algorithms where network weights and architecture parameters are trained alternatively between data batches. For discrete algorithms, different architecture evaluation strategies are used. In SPOS (Guo et al., 2019), child architectures are sampled uniformly and trained to share the weights of common modules. In CARS (Yang et al., 2019), the training and searching of child architectures alternate in each step. Details on more supported evaluation strategies are presented in Appendix G.4.

Results. We summarize the search experiments results in Table 3, Table 4, Table 5, and Table 6, showing the performance of optimized architectures with the search strategies used. For each task and search space, the evaluated search strategies find architectures that surpass the original or randomly selected ones. Several insights are: (a) on search algorithms: Gradient-based algorithms using one-level optimization perform better than their bi-level counterparts. Genetic algorithms outperform other discrete algorithms by a large margin. (b) on evaluation methods: The one-shot training does well in estimating architecture performance using the least training time, while SPOS allows running multiple searches with one-time training. (c) on search spaces: Large convolution kernels contribute to performance boost in MobileNetV2 search space while the ResNet architectures are not sensitive to the kernel size.

Our framework enables efficient and consistent analysis on novel combinations of optimization strategies which might be neglected in previous works. For example, by replacing the sampling strategy in ProxylessNAS with uniform sampling, ProxylessNAS-GU achieves higher accuracy in CIFAR100 experiments compared to ProxylessNAS by 0.2%-0.3% in multiple search spaces. On the other hand, our framework can be used to find the best performing strategy given search space and task. For example, the ResNet-50 search space favors DARTS as search algorithm.

5.2 Experiments on Pareto Optimal Search

Apart from searching for one architecture with the best accuracy, we also explored Pareto optimal architecture search with multi-objective discrete search algorithms, such as genetic algorithms or even random search. We show the feasibility of finding the best architecture for multiple deployment scenarios on any given search space by using multiple non-dominating Metrics as objectives for the Optimizer.

Experiment Details. We use the SPOS evaluation method to estimate the MobileNetV2 architecture performance on the ImageNet dataset where 10% of the training data is held out as the validation set. The latency Metrics are used for each deployment platform, which measures and predicts the latency on the target device. 5000 architectures are sampled for each combination of Metrics.

Results. We compare the curves of the Pareto fronts in each setting and their accuracy/efficiency trade-off on the target hardware platform, as presented in Figure 11. The conclusion can be derived that the FLOPs metrics help find more efficient architectures on multiple platforms with comparable performance. On the other hand, the hardware-specific latency objective contributes to further improvements to the Pareto fronts of the corresponding scenario.

5.3 Evaluation on Framework Usability

We evaluate the usability of several NAS frameworks in terms of performing search on user-defined tasks.

Experiment settings. The voice recognition task in previous evaluations is selected as the target task, which includes a customized search space based on ResNet-34 (He et al., 2016) and user-defined data loading, training, and evaluating procedures based on PyTorch (Paszke et al., 2019). We use the interfaces provided in several NAS frameworks to implement the same NAS setups, including search space, search strategy and procedure, and compare the typical source lines of code (in Python) needed to run NAS procedures using each framework. We select three search setups representing some commonly used NAS methods: Setup A adopts random search strategy while trains each architecture from scratch (Zoph & Le, 2016). Setup B trains a supernet using ProxylessNAS (Cai et al., 2019) method. Setup C applies random search with network transformations modifying the network depth and width (Cai et al., 2020).

Results. The evaluation results are summarized in Table 7. DeepArchitect (Negrinho et al., 2019) implements subnet-based search via a search space description language but fails to support supernet and other efficient search methods. NNI (Microsoft, 2019) requires manually binding the architecture candidates with the network mutables, and use different interfaces in NAS methods that requires adaptation. The ModularNAS framework reduces the amount of work

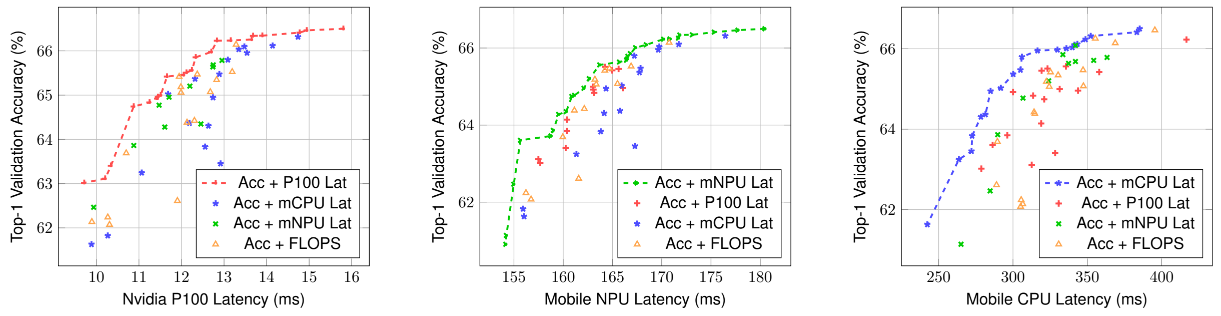

Figure 11. Results on Pareto optimal search using multiple Metrics as objectives. Compared to a single architecture with the best accuracy, we find a curve of architectures that improves the trade-off between performance and efficiency in different deployment scenarios.

Table 7. Evaluation results on framework usability.

#LOC / Framework	Search Space[1]	Setup A[2]	Setup B[3]	Setup C[4]
(None)	N/A	150	200	> 500
DeepArchitect	50	20	N/A	N/A
NNI	30	15	15	200
ModularNAS	< 10	0	0	25

[1] Search space implementation using framework interface.
[2] Random search with each subnet trained independently (Zoph & Le, 2016).
[3] Train a super network with the ProxylessNAS method (Cai et al., 2019).
[4] Search for network depth and width as in OFA (Cai et al., 2020).

required to define a search space on a customized macro architecture using automatic search space generation. No adaptation is needed to switch to different setups since NAS components are modularized and interchangeable.

6 Related Work

Neural Architecture Search. As pointed out in (Elsken et al., 2019), there are three distinct components of the NAS methods: the search space design, the search algorithm, and the evaluation strategy. Different realizations of these three components compose the current NAS literature. For example, using the cell-based search space, the aging evolution and retraining of candidate architecture give the AmoebaNet (Real et al., 2018). As mentioned above, most of the current NAS methods utilize speedup techniques like weight sharing for computational efficiency. Although efficient in execution, these strategies do not have a shared code base. Thus, they are often specific to their use cases, hindering their applications on customized cases, motivating us to develop a NAS framework that overcomes this difficulty.

NAS Frameworks. Several works propose frameworks that try to generalize architecture search implementations to employ them on customized search space. DeepArchitect (Negrinho et al., 2019) describes the search space as a dependency graph of hyperparameters. However, it only serves as a general search space programming language and is entirely agnostic about the details on architecture evaluation and transformation. NNI (Microsoft, 2019) proposes to define mutable regions in an architecture. While it supports efficient evaluation strategies like weight sharing, it achieves that in an ad-hoc and inconsistent way, with architecture parameters treated as weights in some cases (Liu et al., 2019b) and hyperparameters (Pham et al., 2018) in others. Moreover, it does not reuse architecture candidates and still fails to provide unified interfaces for evaluation, resulting in code redundancy and reimplementation.

7 Conclusion

We present ModularNAS, an architecture search framework that enables modularization and reuse of NAS solutions, reducing the amount of work for building NAS pipeline on customized tasks to few lines of code. With a well-defined unified formulation of existing NAS methods and automatic search space generation, different combinations of optimizing and evaluation strategies applied to customized architectures are made tractable using our framework. The flexibility and reusability of ModularNAS is verified through extensive experiments with multiple tasks and strategies.

For future works, we plan to (1) incorporate more search spaces used in other deep learning tasks, (2) integrate more variants of efficient NAS methods, (3) support deployment for more search scenarios such as federated learning, where architecture evaluation and optimization can be done on edge devices and data centers separately.

Acknowledgement

Weinan Zhang is supported by "New Generation of AI 2030" Major Project (2018AAA0100900) and National Natural Science Foundation of China (62076161, 61772333, 61632017). This work is sponsored by Huawei Innovation Research Program.

References

Ashok, A., Rhinehart, N., Beainy, F. N., and Kitani, K. M. N2n learning: Network to network compression via policy gradient reinforcement learning. *ArXiv*, abs/1709.06030, 2018.

Bai, J., Lu, F., Zhang, K., et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. *ArXiv*, abs/1611.02167, 2017.

Baker, B., Gupta, O., Raskar, R., and Naik, N. Accelerating neural architecture search using performance prediction. *arXiv: Learning*, 2018.

Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. V. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Efficient architecture search by network transformation. In *AAAI*, 2018a.

Cai, H., Yang, J., Zhang, W., Han, S., and Yu, Y. Path-level network transformation for efficient architecture search. In *ICML*, 2018b.

Cai, H., Zhu, L., and Han, S. Proxylessnas: Direct neural architecture search on target task and hardware. *ArXiv*, abs/1812.00332, 2019.

Cai, H., Gan, C., and Han, S. Once for all: Train one network and specialize it for efficient deployment. *ArXiv*, abs/1908.09791, 2020.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

Chen, T., Goodfellow, I. J., and Shlens, J. Net2net: Accelerating learning via knowledge transfer. *CoRR*, abs/1511.05641, 2016.

Chollet, F. Xception: Deep learning with depthwise separable convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1800–1807, 2017.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Li, F.-F. Imagenet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

Devries, T. and Taylor, G. W. Improved regularization of convolutional neural networks with cutout. *ArXiv*, abs/1708.04552, 2017.

Dong, X. and Yang, Y. Nas-bench-201: Extending the scope of reproducible neural architecture search. *ArXiv*, abs/2001.00326, 2020.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20:55:1–55:21, 2019.

Guo, H., Tang, R., Ye, Y., Li, Z., and He, X. Deepfm: A factorization-machine based neural network for ctr prediction. *ArXiv*, abs/1703.04247, 2017.

Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. Single path one-shot neural architecture search with uniform sampling. *ArXiv*, abs/1904.00420, 2019.

Han, D., Kim, J., and Kim, J. Deep pyramidal residual networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6307–6315, 2017.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2016.

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

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, 2018.

Head, T., MechCoder, Louppe, G., Shcherbatyi, I., fcharras, Vinícius, Z., cmmalone, Schröder, C., nel215, Campos, N., Young, T., Cereda, S., Fan, T., rene rex, Shi, K. K., Schwabedal, J., carlosdanielcsantos, Hvass-Labs, Pak, M., SoManyUsernamesTaken, Callaway, F., Estève, L., Besson, L., Cherti, M., Pfannschmidt, K., Linzberger, F., Cauet, C., Gut, A., Mueller, A., and Fabisch, A. scikit-optimize/scikit-optimize: v0.5.2, March 2018. URL https://doi.org/10.5281/zenodo.1207017.

Hinton, G. E., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015.

Jin, H., Song, Q., and Hu, X. Auto-keras: An efficient neural architecture search system. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

Krizhevsky, A. Learning multiple layers of features from tiny images. 2009.

Larsson, G., Maire, M., and Shakhnarovich, G. Fractalnet: Ultra-deep neural networks without residuals. *ArXiv*, abs/1605.07648, 2017.

Li, G., Zhang, X., Wang, Z., Li, Z., and Zhang, T. Stacnas: Towards stable and consistent optimization for differentiable neural architecture search. *ArXiv*, abs/1909.11926, 2019.

Lian, J., Zhou, X., Zhang, F., Chen, Z., Xie, X., and Sun, G. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

Liu, B., Zhu, C., Li, G., Zhang, W., Lai, J., Tang, R., He, X., Li, Z., and Yu, Y. Autofis: Automatic feature interaction selection in factorization models for click-through rate prediction. *arXiv preprint arXiv:2003.11235*, 2020.

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A. L., Huang, J., and Murphy, K. Progressive neural architecture search. In *ECCV*, 2018.

Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A. L., and Fei-Fei, L. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 82–92, 2019a.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *ArXiv*, abs/1806.09055, 2019b.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. In *ICLR*, 2017.

Luo, R., Tian, F., Qin, T., and Liu, T.-Y. Neural architecture optimization. In *NeurIPS*, 2018.

Microsoft. Nni: Neural network intelligence. `https://github.com/microsoft/nni`, 2019.

Negrinho, R. and Gordon, G. J. Deeparchitect: Automatically designing and training deep architectures. *ArXiv*, abs/1704.08792, 2017.

Negrinho, R., Patil, D., Le, N., Ferreira, D., Gormley, M. R., and Gordon, G. J. Towards modular and programmable architecture search. *ArXiv*, abs/1909.13404, 2019.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Louppe, G., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

Pincus, M. Letter to the editor - a monte carlo method for the approximate solution of certain types of constrained optimization problems. *Oper. Res.*, 18:1225–1228, 1970.

Qu, Y., Fang, B., Zhang, W., Tang, R., Niu, M., Guo, H., Yu, Y., and He, X. Product-based neural networks for user response prediction over multi-field categorical data. *ACM Transactions on Information Systems (TOIS)*, 37:1 – 35, 2019.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. Large-scale evolution of image classifiers. In *ICML*, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. In *AAAI*, 2018.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

Stamoulis, D., Ding, R., Wang, D., Lymberopoulos, D., Priyantha, B., Liu, J., and Marculescu, D. Single-path nas: Device-aware efficient convnet design. *ArXiv*, abs/1905.04159, 2019.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2815–2823, 2018.

Wang, L., Zhao, Y., and Jinnai, Y. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *ArXiv*, abs/1903.11059, 2018.

Wang, R., Fu, B., Fu, G., and Wang, M. Deep & cross network for ad click predictions. In *ADKDD'17*, 2017.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10726–10734, 2019.

Xie, S., Zheng, H., Liu, C., and Lin, L. Snas: Stochastic neural architecture search. *ArXiv*, abs/1812.09926, 2019.

Yang, T.-J., Howard, A. G., Chen, B., Zhang, X., Go, A., Sze, V., and Adam, H. Netadapt: Platform-aware neural network adaptation for mobile applications. In *ECCV*, 2018.

Yang, Z., Wang, Y., Chen, X., Shi, B., Xu, C., Xu, C., Tian, Q., and Xu, C. Cars: Continuous evolution for efficient neural architecture search. *ArXiv*, abs/1909.04977, 2019.

Ying, C., Klein, A., Real, E., Christiansen, E. L., Murphy, K., and Hutter, F. Nas-bench-101: Towards reproducible neural architecture search. *ArXiv*, abs/1902.09635, 2019.

Yu, J. and Huang, T. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv: Computer Vision and Pattern Recognition*, 2019.

Zhang, H., Cissé, M., Dauphin, Y., and Lopez-Paz, D. mixup: Beyond empirical risk minimization. *ArXiv*, abs/1710.09412, 2018.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *ArXiv*, abs/1611.01578, 2016.

A UNIFIED FORMULATION EXPLAINED

A.1 Relation to RL

We note that the unified NAS process resembles the agent-environment loop in Reinforcement Learning literature. The optimization function Ω acts as the agent and the architecture space as the environment. The transition function δ defines the state transition of the environment. The evaluation function η computes the architecture metrics in R, which serve as the reward of the agent. The environment can be stateless if no intermediate reward is available for the agent.

A.2 Formulation of NAS methods

We express several types of NAS and other architecture refining methods using the unified NAS formulation.

Predictor based. For methods that speed up the evaluation of architectures through the use of architecture performance prediction or extrapolation(Baker et al., 2018; Luo et al., 2018), or a lookup table of all evaluated architectures(Ying et al., 2019; Dong & Yang, 2020), the architecture states no longer represent the actual network model. Instead, we assign v as the internal representation of current architecture in the predictor or the LUT, and η as the predictor function that returns performance of architecture by calling the predictor or looking up the records.

Model compression & pruning. The NAS process is closely related to the model compression and pruning process. Several methods propose iterative modification and retraining of the target network model(He et al., 2018; Han et al., 2016; Ashok et al., 2018; Yang et al., 2018), which naturally fit our formulation. In such cases, $\mathcal{A}$ stands for the action space of operations, the transformation step δ carries out the specific operation such as pruning or compression, and the update step ω and evaluate step η fine-tunes the modified architecture states and evaluates the rewards for the operation, respectively.

Search space transition. In some search space designs(Jin et al., 2019; Negrinho & Gordon, 2017), some architecture parameters are dependent on other parameters, and the search space is spontaneously transformed according to assigned parameters before reaching the terminal state. In this case, the search space transformation is defined through transition functions that map a search space to smaller ones.

In this work, we do not explicitly define the transition of search space. We define the transition of architecture states as triggered by parameter updates in the terminal search space only, which does not affect the universality of our formulation since the transformation of architectures only depends on the values of final architecture parameters rather than their dimensions. To support sequential decisions in search algorithms, we provide an interface to iterate through unresolved architecture parameters in the current search space until all parameters have been assigned values.

B ROUTINES

We present the detailed description of standard routines used in the ModularNAS framework, namely the Search routine, Construct routine, and the Transform routine. We explain the Estimator step and Optimizer step in the next section.

Search routine. A complete search run begins with the generation of search space via Construct routine and initialization of the architecture states and parameters. Then the search process is carried out through interactions of the Estimator step, Optimizer step, and Transform routine in each cycle.

Construct routine. We define the default behavior of the Construct routine as converting each Stub module in the backbone into an actual module specified by the candidate set. The corresponding architecture parameters are added to the Parameter Space. Alternatively, a customized Construct routine may be used without using the Stub modules.

Transform routine. The Transform routine carries out architecture transformation through handler functions. It creates an update event for each updated architecture parameter. Then it polls the handler function triggers to find matching triggering conditions. Finally, the corresponding handler functions are invoked with the updated parameters, and the Transform routine ends. Note that conditions that cover more update events have higher priorities over ones with less triggering events. Thus global transformations will override the local ones if triggered at the same time.

C PROGRAMMING INTERFACE

We describe the interfaces and behaviors for each modular component in the ModularNAS framework. We refer the reader to `https://modularnas.readthedocs.io/` for detailed documentation.

C.1 Parameter Space

The architecture parameter space provides access to the architecture parameters, such as searching by name, modifying values, insertion, deletion, and iteration.

An architecture parameter has some basic properties: its name as the unique identifier in the parameter space, and its data type that determines the type of its value.

Categorical parameter. A Categorical parameter contains unique identifier strings as parameter values, often used in discrete algorithms.

Algorithm 2 Search routine in ModularNAS

```
Input: Backbone $B$, Stubs $S$, Candidate sets $C$
Output: best architecture parameter $P^*$
1: $V, A, H, T \leftarrow$ CONSTRUCT($B, S, C$)
2: $v \leftarrow$ INITIALIZEARCHSTATES($V$)
3: $p \leftarrow$ INITIALIZEARCHPARAMS($A$)
4: while not EXITCRITERION($n$) do
5:     $n, r \leftarrow$ ESTIMATOR.STEP($n$)
6:     RECORD($p, r$)
7:     $p \leftarrow$ OPTIMIZER.STEP($p, r, A$)
8:     $v \leftarrow$ TRANSFORM($v, p, H, T$)
   return parameter $p^*$ with best metrics $r^*$
```

Algorithm 3 Construct routine

```
Input: Backbone $B$, Stubs $S$, Candidates $C$
Output: $V, A, H, T$
  $V, A, H, T \leftarrow \emptyset$
  for $s_i$ in $S$ do
      $m_i, V_i \leftarrow$ DEFINEMODULE($s_i, B, C_i$)
      $A_i, h_i, t_i \leftarrow$ DEFINEPARAM($s_i, B, C_i$)
      $V \leftarrow V \times V_i$
      $A \leftarrow A \times A_i$
      $H \leftarrow H \cup \{h_i\}$
      $T \leftarrow T \cup \{t_i\}$
      SUBSTITUTE($s_i, m_i$)
  return $V, A, H, T$
```

Algorithm 4 Transform routine

```
Input: $v, \alpha, H, T$
Output: new architecture states $v'$
  $v = \{v_j\}$
  $\alpha = \{\alpha_i\}$
  $H = \{h_k\}$
  $T = \{t_k\}$
  $U \leftarrow \{u_i = $ True$\}$ for $\alpha_i$ in $\alpha$
  $v' \leftarrow v$
  for $h_k$ in PRIORITYORDER($H$) do
      if $t_k(U)$ is True then
          $v' \leftarrow h_k(\alpha, v')$
  return $v'$
```

Figure 12. Standard routines in ModularNAS framework. Top: unified Search routine for NAS process. Left: Construct routine for generating search space. Right: Transform routine for modifying architecture states according to the updated architecture parameter values.

Tensor parameter. A Tensor parameter contains a tensor with a given shape and data type that can be differentiable, often used in gradient-based algorithms.

Numerical parameter. A Numerical parameter contains a number in float point or integer type as its value, used in Bayesian optimizations and hyperparameter tuning space.

C.2 Optimizer

The Optimizer provides two interfaces: NEXT proposes the next set of parameter values to be evaluated, and STEP updates the Optimizer states according to the last parameter values and the Metrics results in the Estimator. See Figure 13 for examples of search algorithms implemented as Optimizer modules.

Discrete algorithms. We show an example of Evolution(Real et al., 2017) search algorithm to represent search algorithms that work on discrete parameters. At initialization, the population is filled with random architecture parameter values with no metrics. The NEXT function returns the next set of parameter values that have not been evaluated. In STEP function, the evaluated parameters are added to the population queue. Once all the individuals are evaluated, the algorithm produces the next generation through a sequence of genetic operations such as Survival, Selection, Crossover, and Mutation.

Gradient-based algorithms. In gradient-based algorithms, the Estimator provides the results of the Criterion modules, which are backward propagated to obtain the gradient of the architecture parameters. The tensor optimizer specified in the gradient-based Optimizer will update the architecture parameters according to the gradients.

C.3 Estimator

Once the architecture is obtained through transformation, the Estimator updates its states and uses Metrics components to compute the desired property of the architecture, such as performance, latency, and energy consumption. The estimation strategy decides how the Estimator update the

Algorithm 5 Evolution Optimizer

```
1: $P \leftarrow \emptyset$
2: function INITIALIZE($N$)
3:     $P \leftarrow$ INITIALIZEPOPULATION($N$)
4: function NEXT
5:     return first $\alpha$ for $(\alpha, \emptyset)$ in $P$
6: function STEP($\alpha, r$)
7:     Add $(\alpha, r)$ to $P$
8:     if all $\alpha_i$ in $P$ have metrics $r_i$ then
9:         while $\|P\| < N$ do
10:            $P \leftarrow$ SURVIVAL($P$)
11:            $P \leftarrow$ SELECTION($P$)
12:            $P \leftarrow$ CROSSOVER($P$)
13:            $P \leftarrow$ MUTATION($P$)
```

Algorithm 6 DARTS Optimizer

```
1: function INITIALIZE($n$)
2:     Initialize tensor optimizer
3: function NEXT
4:     for $\alpha_i$ in current $\alpha$ do
5:         copy $\alpha_i$ to graph node
6:     return $\alpha$
7: function STEP($\alpha, r$)
8:     Virtual step
9:     Compute loss $L$
10:    Compute grad $g$ of $\alpha$
11:    Compute hessian $H$
12:    $g_i \leftarrow g_i - \xi H_i$
13:    Tensor optimizer step
```

Figure 13. Example of Optimizer implementations. Left: Optimizer that employs an Evolution(Real et al., 2017) algorithm. Right: Optimizer that implements DARTS(Liu et al., 2019b) algorithm, where a virtual training step and second-order approximation is applied.

architecture states. A Criterion module may use the Metrics results to compute the gradient of the architecture parameters in gradient-based methods. Details on the Metrics and Criterion modules can be found in Appendix E and Appendix F, respectively.

D Architecture Candidates

The backbone models and candidate architectures are decoupled and registered as shared components which can be reused in different experiment settings, including some frequently used neural operators such as convolutions, and building blocks such as residual blocks. The correct configuration for each candidate module, such as input and output data shape, can be retrieved from the stub modules defined in the backbone architecture, which contain the same configuration used to construct the original modules.

Primitive. We define the primitive to be any architecture module that does not include stub modules or candidate modules and can be determined by the input and output data sizes. Given the primitive type, the Construct function will choose the correct primitive to construct using the information stored in the stub module.

Mixed Operator. A mixed operator contains a list of candidate operations. The output of a mixed operator is computed using the output of some number of the candidate operations. We implement the mixed operator as a network module that can be converted from Stub provided with a list of primitives.

Layer. A layer is a primitive that can have stubs as its submodules. The framework supports chaining several constructor functions that hierarchically convert each Stub to a layer of primitives. Some predefined layers include Direct Acyclic Graph(Liu et al., 2019b) layer, Tree layer(Cai et al., 2019), and Multi-branch(Szegedy et al., 2015) layer.

Elastic Spatial & Sequential Group. The Elastic Spatial Group is a group of Transform handlers that support dynamically changing the width (channels) of architecture modules during runtime. On the other hand, the Elastic Sequential Group changes the depth of the network model by skipping some nodes in the computation graph. The target width and depth can be determined by architecture parameters, making them accessible to the Optimizer.

An Elastic Spatial Group modifies the number of channels of a single hidden layer (feature map). It manages two kinds of layers: fan-out layers, to which the hidden layer is the output, and fan-in layers, to which the hidden layer is the input. When setting some channels of the hidden layer as active ones, the group temporarily reduces the dimensions of the weights in the fan-out and fan-in layers, so that they behave like the same layer with inactive channels reduced. Given the target number of channels, the index of the active channels can be determined in several ways. These include using extra parameters as attention, L1-norm of fan-in layer weights, or affine parameters in middle normalization layers.

An Elastic Sequential Group takes several groups of network modules and temporarily replaces some with identity operations, resulting in reduced network depth.

E Metrics

A Metric module computes a property of the architecture during Estimator evaluation. Metrics provide an interface named COMPUTE, which evaluates the properties of a given target. Metrics can be nested, where outer Metrics collect the results of the inner Metrics on multiple modules or objects as inputs.

E.1 Traversal Metrics

One of the essential metrics is the traversal metrics, which iterate over all modules in the architecture and collects the properties of each module by calling the internal metrics.

E.2 Computational Metrics

We implement a profiling utility that measures several computational properties of network networks, including FLOPs, memory access, parameters count, at the graph node level. We then import these functionalities in ModularNAS as internal Metrics that operate on individual modules in a network model.

E.3 Hardware Metrics

Some properties of the architecture are related to the hardware where it is deployed, such as latency and energy consumption. To measure such properties, we implement a pipeline of hardware performance measurement via a chain of Metrics.

Export Metrics. The export Metrics converts and saves the current architecture in the form supported by the target hardware. It also makes use of commonly used model formats like ONNX(Bai et al., 2019).

Profiling Metrics. The profiling Metrics runs the converted architecture on the target device(Tan et al., 2018) and returns its runtime performance statistics.

Data Metrics. The data Metrics collect the performance data to provide the correct results for different inquiries. It can also use data sources like lookup tables and hardware performance predictors(Cai et al., 2019; Wu et al., 2019) or simulators.

F Criterion

The Criterion module computes the loss function of the network model using the input data, model output, and the architecture states, such as the results of some Metrics. During training, the loss can be computed through a sequence of multiple criterion components, with each Criterion taking the input data pair, the model output, and the previous Criterion output as its input. Criterion modules may also be nested.

Task-specific Criterion. We register task-specific loss functions as primary Criterion modules, such as cross-entropy loss and mean-squared error loss. This type of Criterion comes as the first in a Criterion sequence. Thus they only require the input data and model output.

Aggregate Criterion. The aggregate criterion takes a value from a Metrics and put it together with the loss value of the previous Criterion. Methods of aggregation include addition, multiplication, and multiplication by the logarithmic.

Special Criterion. Some criterion manipulates the inputs before computing the loss value. For example, we implement the MixUp(Zhang et al., 2018) as a Criterion that mixes one data pair with another. We also implement the Knowledge Distillation(Hinton et al., 2015) as a Criterion that uses the logits of a teacher model as ground truth.

G Details on experiment settings

G.1 Search Space Settings

Cell-based. Following DARTS(Liu et al., 2019b), the cell-based search space consists of 8 stacked convolutional cells, each containing seven nodes and 14 edges. Reduction cells are located at the 1/3 and 2/3 of the network depth. The architecture parameters represent the choice for each edge in normal and reduce cells, which sums to a total of 28 architecture parameters in the search space. The initial number of channels in supernet is 16 and is increased to 32 in final architecture. The number of cells is also increased to 20. The set of candidate operations is kept the same as in previous works, where operators are arranged in the ReLU-Conv-BN order.

MobileNetV2 backbone. For MobileNetV2(Sandler et al., 2018) backbone, we follow ProxylessNAS(Cai et al., 2019) settings and replace the convolutions of every residual block as Stub modules to search between convolutions of different kernel size (3, 5, 7) and expansion ratio (1, 3, 6). We insert zero operation in the candidate set to represent a reduction in network depth. We exclude the first residual block from the search space.

MobileNetV2-Elastic. We provide another search space based on MobileNetV2(Sandler et al., 2018) backbone with elastic width and depth. We use the Elastic Spatial Group transformations on the output feature map of the depthwise convolution in every residual block and search between expansion ratios (1, 3, 6). We use the Elastic Sequential Group transformation on each bottleneck stage and search for different repetitions of blocks (1, 2, 3, 4). We exclude the first residual block from the search space. We fix the kernel size at 3x3 for all depth-wise convolutions.

ResNet backbone. For ResNet(He et al., 2016) backbone, we replace all the 3x3 convolutions in each residual block as Stub modules, and we define the candidate set as convolutions with kernel size ranging from 3, 5, and 7. We include depthwise-separable convolutions(Chollet, 2017) in an extended candidate set.

CTR model. We design a Xception(Chollet, 2017) like multi-branch feature interaction network as the base model for CTR prediction task. For intermediate layers, we select implicit feature interactions as candidates which produce output with the same shape as the input. For examples, inner product (IP), Hadamard product (HP), and cross product (OP) with linear resize layers, MLP, Cross layer in DCN(Wang et al., 2017), and CIN layer in xDeepFM(Lian et al., 2018). The branch outputs are summed to obtain the prediction.

G.2 Training Settings

Cell-based on CIFAR-10/100. For the supernet in cell-based search space, we followed training settings in DARTS(Liu et al., 2019b). The weight parameters are optimized using SGD, with initial learning rate 0.025 with cosine annealing schedule(Loshchilov & Hutter, 2017), momentum 0.9, weight decay 3e-5, and gradient clipping 0.5. The batch size is set to 64. For the final architecture training, we set the training batch size to 96 with Cutout(Devries & Taylor, 2017) of size 16, and train for 600 epochs using the auxiliary loss of ratio 0.4, and DropPath(Larsson et al., 2017) of rate 0.2.

MobileNetV2 on ImageNet. For the MobileNetV2 backbone, we set the base batch size to 256, and use SGD optimizer with learning rate 0.05 (annealed to zero with cosine schedule), Nesterov momentum 0.9, and weight decay 4e-5. We train the final architecture for 150 epochs with label smoothing. We linearly scale the batch size and learning rate to 8 Tesla V100 GPUs in parallel.

CTR model on Avazu. For the CTR prediction model, we follow common training settings and use Adam(Kingma & Ba, 2015) optimizer with learning rate set to 0.001 and decaying exponentially at the rate of 0.85. The model is trained for one data epoch.

G.3 Search Algorithm Settings

DARTS. For DARTS(Liu et al., 2019b) search algorithm, we use Adam(Kingma & Ba, 2015) as optimizer for tensor architecture parameters, with learning rate 3e-4, momentum $(0.5, 0.999)$ and weight decay 0.001. We disable the unrolling step and second-order approximation in one-level optimization cases. We set the search epochs to 50 for bi-level settings, and 80 for one-level settings.

ProxylessNAS. For ProxylessNAS(Cai et al., 2019) algorithm, we use Adam as parameter optimizer with learning rate 0.006, momentum $(0, 0.999)$ and zero weight decay. We sample two candidate paths for each choice block in the search step and rescale the architecture parameters after the update.

SNAS. For SNAS(Xie et al., 2019) algorithm, we use the same optimizer settings as in DARTS, and set the initial annealing temperature to 5.0, with exponential annealing rate 7.5e-5 per mini-batch data, or 4.5e-2 per epoch.

Genetic Algorithms. We implement an evolution algorithm(Real et al., 2017) with typical settings. We set the population size to 100, with the number of elimination set to 1, the number of selection set to 10, mutation probability set to 0.01. We set two parents and one offspring for each crossover operation. We also implement Regularized Evolution(Real et al., 2018) that eliminates the oldest individuals.

SMBO methods. We implemented SMBO algorithms where an Acquire method finds the global maximum in the performance prediction model trained from history data in search process. We use Simulated Annealing(Pincus, 1970) and random sampling as acquire methods. For prediction models, we use GBDT methods in XGBoost(Chen & Guestrin, 2016) and linear models like MLP and Linear regression in Scikit-learn library(Pedregosa et al., 2011).

Bayesian optimization. We use Bayesian optimizations in hyperparameter tuning process. Specifically, we use the Gaussian Process estimator provided in the Scikit-optimize library(Head et al., 2018).

G.4 Evaluation Strategy Settings

SPOS strategy. The single-path one-shot (SPOS) strategy(Guo et al., 2019) trains each child architecture sampled from a uniform distribution in the training process. In the search process, each child architecture inherits weights from shared nodes and is evaluated without retraining. We perform batch normalization sanitization for each architecture, i.e., we use training data to recalculate the moving average and variance of BN layers to restore the distributions of intermediate feature maps. We set the training epochs to 1000 for ImageNet experiments, and 800 for CIFAR10/100 experiments.

CARS strategy. The training process in CARS(Yang et al., 2019) alternates with the architecture search process, where architectures sampled in each search step are trained for several epochs before evaluation. We use the same Estimator component to implement the CARS and SPOS strategy, except the number of training epochs for sampled architectures is zero in the CARS strategy. We apply uniform warm-up training before the search stage.

PS strategy. The progressive shrinking (PS) strategy proposed in Once for All(Cai et al., 2020) arranges architectures according to the model size, such as width and depth. Then the sampling and training are carried out in descending order, from larger networks to smaller ones nested within. We follow OFA settings and reorder the convolution parameters in channel dimension according to the L1 norm in each training stage, while we disable the knowledge distillation(Hinton et al., 2015) in our experiments. We employ BN sanitization for each child network during the search process. This strategy is only used in architecture space featured with elastic transformations such as dynamic width and depth.

H Additional experiment results

H.1 Evaluation on Search Time Penalty

Our framework achieves modularity of the search space through procedural search space generation and event-triggered network transformation in Construct and Transform steps, which results in extra network structures and function calls and may increase search time.

We measure the training and inference time of networks generated from Constructors and found no significant difference between networks specified by fixed code of the same functionality. As a result, the search time remain mostly the same regardless of using the framework or not. On the other hand, the evaluation step specified by the user-defined trainer consumes most of the time in a search procedure. Thus, no significant search time penalty is introduced by framework modularity.

H.2 Experiments on Additional Datasets

We migrate the MobileNetV2 backbone experiments to more image classification datasets to verify the general effectiveness of the search algorithms on different tasks.

Dataset details. Tiny-ImageNet-200 is a downsampled subset of the ILSVRC2012 dataset(Deng et al., 2009), which contains 200 selected image classes, with 500 training samples and 50 validation samples for each class. The images are downsampled to 64x64. Alternatively, We randomly select 100 classes with 800 original training images each to form the ImageNet-Sub-100 dataset. The images are cropped to 224x224 in the search process.

Experiment details. We use the MobileNetV2 search space settings in ProxylessNAS(Cai et al., 2019). We use the same Optimizer and Estimator settings as in ImageNet experiments. For SPOS(Guo et al., 2019) strategy, we sampled 5k architectures for each search algorithm except the last random search run, with 20k architectures sampled.

Results. We summarize the final validation results of found architectures in Table 8. We further evaluate the effectiveness of architecture estimation through proxy dataset by select the best architectures found on each dataset and train them on the full ILSVRC2012 dataset. The results are reported in Table 9. Compared to the downsampled Tiny-ImageNet-200 dataset, the ImageNet-Sub-100 dataset with full resolution and fewer classes serves as a better proxy for finding architectures with higher performance.

H.3 Experiments on Elastic Groups

We validate the implementation of Elastic Groups by using them to build a search space that contains architectures of different width and depth. We then use various combinations of Optimizers and Estimators to search for the best architecture. See MobileNetV2-Elastic in Appendix G.1 for the search space definitions.

Results. We summarize the results in Table 10. We conclude that search algorithms prefer larger architectures with PS evaluation strategy than with SPOS strategy, which shows the effectiveness of PS in preserving the accuracies of larger subnets. However, this search space contains architectures mostly smaller than the original models. Without the use of knowledge distillation(Hinton et al., 2015), child architectures are always inferior to the baseline model.

H.4 Visualization of Gradient-based algorithms

We visualize the variation of differentiable architecture parameters in each training epoch using several supported gradient-based algorithms: DARTS(Liu et al., 2019b), SNAS(Xie et al., 2019), and ProxylessNAS-G(Cai et al., 2019).

Experiment settings. We use the same Cell-based search space settings as in DARTS(Liu et al., 2019b). For validation data size, we hold out 50% of training data for DARTS and SNAS(bi-level) algorithms, 20% for ProxylessNAS-G, and none for one-level cases.

Results. We plot the last one of the 28 tensor parameters, which encode the choice of candidates in normal and reduction cells. As shown in Figure 14, different search algorithms exhibits different update patterns of architecture parameters. Specifically, the Proxyless-G algorithm introduces randomness compared to DARTS and SNAS algorithms due to binarized paths in training and searching steps. Besides, search algorithms with bi-level and one-level settings show diverse update trajectories.

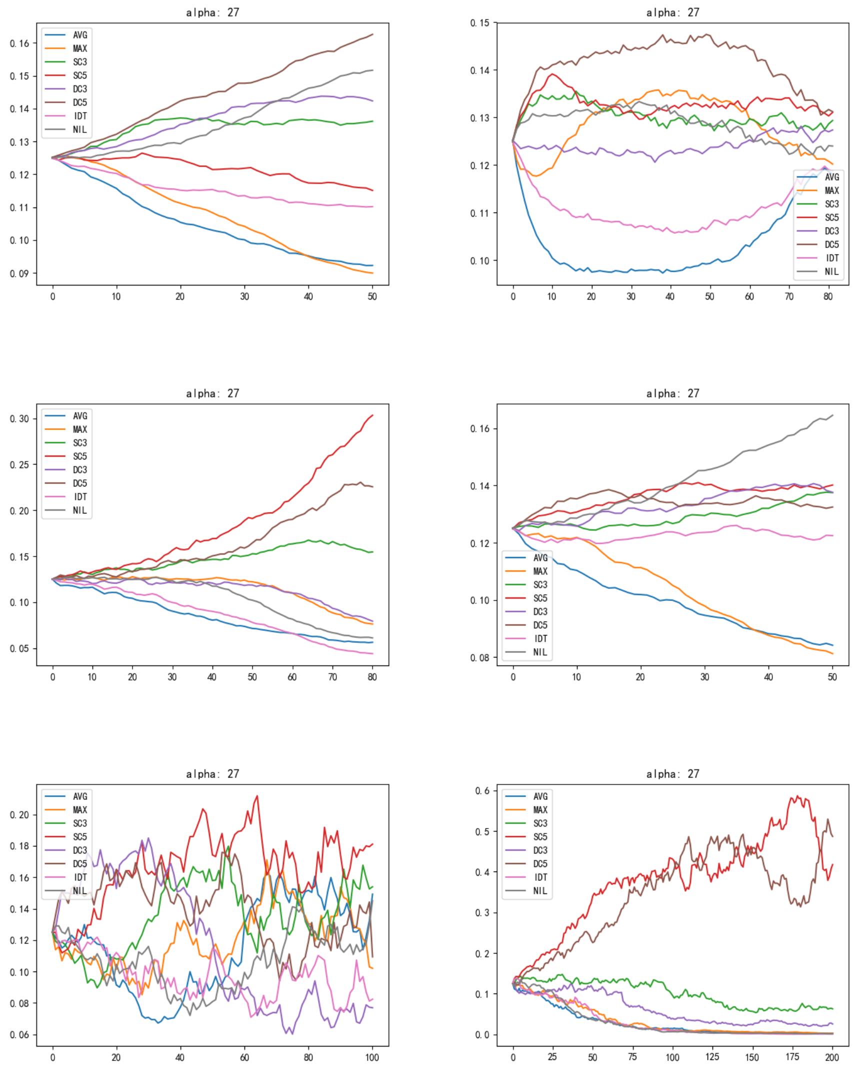

Figure 14. Visualization of architecture parameter update trajectories for each candidate path in the last choice block of the cell-based search space. Top: DARTS(Liu et al., 2019b), DARTS (one-level). Middle: SNAS(Xie et al., 2019). SNAS (bi-level). Bottom: ProxylessNAS-G(Cai et al., 2019), ProxylessNAS-GU (uniformly sampling paths in forward pass).

Table 8. MobileNetV2 backbone results on additional datasets

Dataset	Top-1 (%)	Params (M)	Search Method	Search cost (GPU hrs)
Tiny-ImageNet-200	63.01	3.66	Baseline	
	63.77	3.668	SPOS+Random	100+25
	63.86	3.673	SPOS+RE	100+25
	63.32	3.489	SPOS+NRE	100+25
	64.48	3.287	SPOS+Random (20k)	100+100
ImageNet-Sub-100	80.88	3.532	Baseline	
	80.22	3.716	DARTS (one-level)	200
	81.18	3.172	DARTS	150
	81.52	3.801	Proxyless-G	50
	81.80	3.723	Proxyless-GU	50

Table 9. Transfered search results on ILSVRC2012 dataset

Dataset	Top-1 (%)	Params (M)	Search Method	Search cost (GPU hrs)
Tiny-ImgNet-200	**74.73**	4.311	SPOS+Random (20k)	100+100
ImageNet-Sub-100	**75.38**	4.876	Proxyless-GU	50

Table 10. Elastic MobileNetV2 results on CIFAR100

Arch. Space	Top-1 (%)	Params (M)	Search Method	Time (hours)
MbV2-E	73.46	3.532	Baseline	
	70.47	1.982	SPOS+Random	24+8
	71.51	1.531	SPOS+RE	24+8
	71.39	2.104	SPOS+MLP	24+8
	71.28	2.515	PS+Random	36+8
	72.07	2.503	PS+RE	36+8
	71.17	2.742	PS+MLP	36+8