# OSNASLib: One-Shot NAS Library

Sian-Yao Huang and Wei-Ta Chu
National Cheng Kung University, Tainan, Taiwan
{P76084245, wtchu}@gs.ncku.edu.tw

## Abstract

*Incorporating neural architecture search (NAS) into a targeted task is an emerging research branch to further put performance forward. We propose an open-source one-shot NAS frameowrk called OSNASLib to empower users to easily integrate or design one-shot NAS methods for a target task. By well modulating important components in one-shot NAS, users can customize one or more components at will. With the provided interface generator, input/output definitions and the links between components can be initialized easily, and users can focus on algorithm design or functional details implementation. We take image classification and face recognition as the sample tasks, and show performance of baseline methods provided in the library. The source code is available at :* `https://github.com/eric8607242/OSNASLib`.

## 1. Introduction

Designing deep neural architectures to obtain good performance is a common task in current multimedia and computer vision researches. Researchers usually need to jointly consider performance and resource consumption. However, manually tuning and designing neural architectures is very time-consuming. Therefore, researches of neural architecture search (NAS) emerge recently to search good neural architectures for different scenarios automatically.

There are three major components in NAS: search space, search strategy, and performance estimation. Search space is consisted of all candidate neural architectures. From a search space, NAS methods search the best neural architecture for different constraints. A search strategy is thus needed to search the best architecture efficiently from the search space. Random search [8], grid search, evolution algorithm [2] are common search strategies adopted by many NAS methods. To search the architecture yielding the best performance, a performance estimation method is required to estimate performance of any specific architecture. Accuracy predictor, supernet [12, 1], and proxy training architectures are common ways adopted in existing NAS methods.

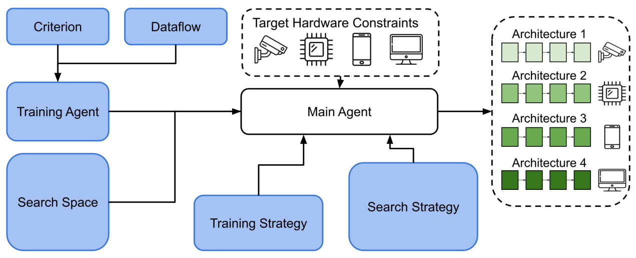

Figure 1: The structure of OSNASLib. The components shown in blue can be customized by users easily.

The earliest NAS methods trained thousands of architectures to evaluate performance of architectures, which required extremely expensive computation and resource. Recent one-shot NAS methods [12, 10, 6] focus on improving search efficiency. They encode the entire search space into an over-parameterized network called a supernet. In each layer of the supernet, candidate blocks of various configurations are included. After training the supernet, all architectures in the search space can be approximated and evaluated by activating different candidate blocks in each layer without additional training. Therefore, one-shot NAS is more efficient because only one neural network (supernet) training is needed.

Although one-shot NAS is efficient, it is still challenging to incorporate NAS methods with different tasks. The challenges include:

- The entry barrier of designing a NAS method is high. Much prior knowledge and implementation skills are needed.
- Most NAS methods were developed for image classification tasks. Various coding styles and training pipelines make extending them to other tasks not feasible.

In this work, we propose OSNASLib that is a general one-shot NAS framework empowering users to integrate one-shot NAS methods into various tasks easily. OSNASLib consists of six major components: criterion, dataflow, training agent, search space, search strategy, and training strategy, as shown illustrated in Fig. 1. For each component, OSNASLib provides several baselines and allows users to customize them for various tasks flexibly. To build a basic one-shot NAS, users only need to set the training agent, dataflow, and criterion without implementing functional details. Furthermore, users can develop a new one-shot NAS method by customizing search space, search strategy, and training strategy. With the provided baselines, OSNASLib allows users to compare their proposed method with baseline NAS methods fairly.

To customize for various tasks, users can implement details of their proposed methods based on provided baselines. OSNASLib clearly defines input/output formats of each component, and the relationship between components. The input/output definitions and how different components should be linked are integrally called an "interface" in this work. OSNASLib provides an interface generator to assist users to generate the interface for any designated component. With the interface, users can focus on designing and implementing functional details of each component. One or more of the six components shown in Fig. 1 can be easily initialized as the user's wish. Other components can be left with default settings. This design makes OSNASLib very flexible and makes users focused.

OSNASLib is suitable for the following users:

- NAS beginners who want to build codebases of basic one-shot NAS methods quickly.
- Researchers who want to integrate one-shot NAS methods into a targeted task to improve performance.
- Researchers who focus on designing NAS methods and want to validate their methods for various tasks and compare with other baseline methods fairly.

## 2. Framework

### 2.1. Training Agent

Integrating NAS methods into deep neural networks for specific tasks has emerged recently. However, for different tasks, different training pipelines are needed to proceed the NAS process. To make OSNASLib more flexible, we decouple the training pipeline from the entire NAS process. With this isolation, OSNASLib allows users to focus on implementing details of the training pipeline like forward/backward parameter updating. The *training agent* illustrated in Fig. 1 embodies the training pipeline. OSNASLib incorporates the training agent with the pre-defined searching pipeline, called *main agent*, to search the best neural architecture for the targeted task automatically.

### 2.2. Criterion

To construct the training agent, users can customize various loss functions for the target task. The component for defining loss functions is called *criterion* in OSNASLib. Separating this component from the training agent is also due to the consideration of flexibility.

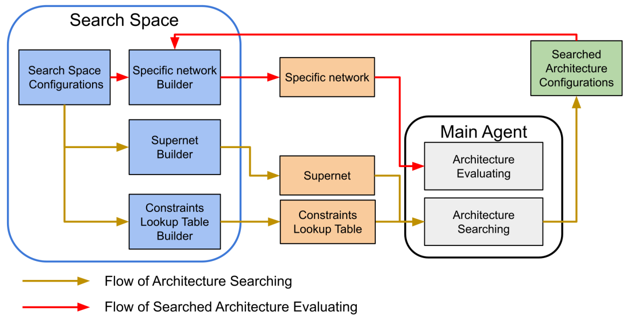
Search Space
Search Space Configurations
Specific network Builder
Supernet Builder
Constraints Lookup Table Builder
Specific network
Supernet
Constraints Lookup Table
Searched Architecture Configurations
Main Agent
Architecture Evaluating
Architecture Searching
Flow of Architecture Searching
Flow of Searched Architecture Evaluating

Figure 2: Structure of the search space. With the provided interface (blue regions), users can define various search spaces by setting different supernet structures and different search space configurations.

### 2.3. Dataflow

For different tasks and different datasets, data pre-process or specific data loading schemes are needed. OSNASLib allows users to customize the dataflow by designing appropriate pre-processing, different data samplers, or different data loaders. With this isolation, users can apply NAS methods on different datasets with the same OSNASLib components easily.

### 2.4. Search Space

In OSNASLib, we allow users to customize the search space for different application scenarios with different hardware resources. Fig. 2 illustrates structure of the search space component in OSNASLib. To define the search space, users should configure a supernet and a hardware constraint lookup table. In the supernet, configurations of the search space like candidate blocks of each layer, and input/output channel sizes of each layer, should be set. Based on the configurations, the main agent constructs the supernet for architecture search. With the provided interface, it is flexible to implement different supernet structures.

To search architectures under various hardware constraints, OSNASLib evaluates the "hardware cost" of the sub-network $a$ in a supernet as

$$Cost(a) = \sum_{l} Cost(a_l), \tag{1}$$

where the term $Cost(a_l)$ is a constant cost of the block in the $l$th layer of $a$. For each search space, OSNASLib constructs a hardware constraint lookup table to store the resource requirement of each candidate block in each layer. The goal of utilizing this lookup table is to evaluate the hardware cost quickly, which has been widely adopted in many one-shot methods [6, 12].

After architecture search, OSNASLib builds a specific network according to the searched architecture configuration. The specific network is then passed to the main agent to be trained from scratch and then evaluated.

## 2.5. Training Strategy

One-shot NAS utilizes a supernet to estimate performance of architectures in the search space. Therefore, how to train a supernet well is very important. A poor supernet may mislead the entire search process and good architectures may potentially not to be found.

The supernet training strategy can also be customized by users in OSNASLib. By implementing functional details based on the generated interfaces, users can focus on development of the supernet training strategy without the distraction of other components.

## 2.6. Search Strategy

Given a supernet $A$, a search strategy aims at searching the best architecture $a$ from the search space under a target hardware constraint $C$. That is,

$$a^* = \underset{a \in A}{\operatorname{argmin}} \mathcal{L}_{val}(\boldsymbol{w}(a)). \tag{2}$$

$$\text{s.t. } Cost(a^*) \leq C, \tag{3}$$

where the term $\boldsymbol{w}$ denotes weights of the supernet, $\mathcal{L}_{val}$ denotes the validation loss, $Cost(a^*)$ denotes the hardware constraint of the architecture $a^*$ calculated by Eq. 1, and $\boldsymbol{w}(a)$ denotes the subset of $\boldsymbol{w}$ corresponding to the sampled architecture $a$.

It is important to design a search strategy that can efficiently search the best sub-network from the supernet. OSNASLib also offers flexibility for users to customize the search strategy. With the provided interfaces, users can implement their search strategies or utilize the default strategies.

## 2.7. Baseline Components

With OSNASLib, users can also compare their proposed NAS method with other methods fairly (e.g., based on the same search hyper-parameters and the same codebase), or evaluate their proposed NAS method on various tasks. This is enabled based on the baseline components provided in OSNASLib. Details of the provided baseline components are shown in Table 1.

```
python3 build_interface.py \\
    -it search_space \\
    --customize-name mynas \\
    --customize-class MyNAS
```

Figure 3: An example of the command for generating an interface for the search space component.

Table 1: The baseline components in OSNASLib.

| Components | Baselines |
|---|---|
| Task | Classification, Face recognition |
| Criterion | Cross entropy loss, Triplet loss [11], Focal loss [9] |
| Dataflow | Cifar10 / 100 [7], Imagenet [3], CASIA-WebFace [13] |
| Search Space | SPOS [4], ProxylessNAS [1], FBNet [12], MobileFaceNet ,SGNAS [6] |
| Search Strategy | Softmax differentiable searcher [10]<br>Gumbel softmax differentiable searcher [12]<br>Evolution searcher [4], Random searcher [8],<br>Architecture generator [6] |
| Training Strategy | Differentiable sampler [10, 12],<br>Uniform sampler [4], Fairness sampler [2] |

```
search_space/
    |- example/
    |       |- __init__.py
    |       |- mynas_lookup_table.py
    |       |- mynas_model.py
    |       |- mynas_supernet.py
    ...
```

Figure 4: The results generated by the interface generator with the file name "mynas". These files are automatically imported into the main agent.

## 3. Interface Generator

OSNASLib empowers users to customize specific components for various tasks. However, importing and cooperating multiple components into the main agent are tedious. To reduce this burden, OSNASLib provides an interface generator to generate interfaces for each component and automatically import necessary files to the main agent. To generate interfaces, only one command is needed. For example, Fig. 3 shows how to use the interface generator to generate the interface for the search space component with the file name "mynas" and class name "MyNAS". The generated results are shown in Fig. 4.

## 4. Image Classification

We first take image classification to showcase the usage of OSNASLib. All experiments were conducted on the CIFAR100 dataset. For architecture search, we randomly sample 80% of images from the training set as the training data, and the rest is kept as the validation data. After architecture search, we train the searched architecture based on the full training data, and test it on the original validation set defined in CIFAR100. All experiments were experimented on one GEFORCE RTX 3090 GPU.

Table 2: Performance of baseline search space components in OSNASLib.

| Search Space | FLOPs (M) | Top-1 Acc (%) | Search time (sec) | Size of search space |
|---|---|---|---|---|
| ProxylessNAS [1] | 28.2±2.9 | 70.8±0.6 | 1680 | $\sim 5.6 \times 10^{17}$ |
| FBNetS [12] | 30.4±1.9 | 70.5±0.7 | 1847 | $\sim 10^{21}$ |
| SPOS [4] (w/o channel search ) | 27.5±1.7 | 68.9±0.8 | 1711 | $\sim 10^{12}$ |
| SGNAS [6] | 29.9±0.3 | 71.8±0.2 | 2145 | $\sim 10^{40}$ |

Table 3: Performance of baseline search strategy components in OSNASLib.

| Search Strategy | FLOPs (M) | Top-1 Acc (%) | Search time (sec) |
|---|---|---|---|
| Evolution Algorithm [4] | 23.7±1.5 | 69.5 ±0.8 | 1728 |
| Random Search [8] | 25.8±0.2 | 70.0±0.2 | 2018 |
| Architecture Generator [6] | 28.2±2.9 | 70.8±0.6 | 1696 |

Table 4: Performance of baseline search space components on face recongition.

| Search Space | FLOPs (M) | Top-1 Acc (%) |
|---|---|---|
| ProxylessNAS [1] | 291.91 | 95.00 |
| FBNetL [12] | 285.73 | 94.68 |
| SPOS [4] (w/o channel search) | 289.64 | 95.18 |
| SGNAS [6] | 291.58 | 95.00 |

### 4.1. Analysis of Search Strategy

Here we compare the baseline search strategies, including random search [8], evolution algorithm [4], and architecture generator [6] on the ProxylessNAS [1] search space. We first train the supernet with the uniform sampling [4] training strategy. After training the supernet, we utilize three search strategies to search the architecture under 27M FLOPs, respectively. This process is run for three times, and the average performance, the average search time, and the average FLOPs of searched architectures are shown in Table 3.

In the experiments, we found that the search strategy architecture generator [6] can achieve the highest top-1 accuracy and the lowest search time. Because the architecture generator is constructed based on the gradient descent algorithm, the FLOPs of the searched architecture are not strictly under the target FLOPs (27M). In addition, the random search [8] strategy achieves satisfactory performance compared to other baseline methods.

### 4.2. Analysis of Search Space

We further compare baseline search spaces, including that in ProxylessNAS [1], SPOS [4], FBNetS [12], and SGNAS [6]. We train the supernets based on different search spaces with the uniform sampling [4] training strategy. After training a supernet, we search the architecture under 27M FLOPs with the architecture generator [6] search strategy. The experiment was run for three times, and the average top-1 accuracy, the average FLOPs, the average search time, and size of the search space are shown in Table 2.

We see that SGNAS can achieve the highest top-1 accuracy with the largest search space. On the other hand, it's interesting to see that, although the search space of FBNetS is larger than ProxylessNAS, ProxylessNAS achieves better top-1 accuracy.

## 5. Face Recognition

We also use OSNASLib to integrate one-shot NAS into a face recognition task, in order to verify that OSNASLib can be flexibly adjusted to different tasks. All experiments here were conducted on the CASIA-WebFace dataset [13]. For architecture search, we randomly sample 80% of images in the dataset as the training data, and the rest is kept as the validation data. After architecture search, we train the searched architecture based on the whole CASIA-WebFace dataset, and test the trained network based on the LFW dataset [5].

Here we compare the baseline search spaces, including that in ProxylessNAS, SPOS, FBNetL, and SGNAS. We train the supernets based on different search spaces with the uniform sampling [4] training strategy. After supernet training, we search good architectures under 300M FLOPs with the evolution searcher [4].

Face recognition performance is shown in Table 4. From this table, we found that SPOS can achieve the highest top-1 accuracy. It is interesting to see that the search spaces yielding the highest top-1 accuracy are different for that for image classification.

## 6. Conclusion

In this work, we propose a general one-shot NAS framework called OSNASLib to allow users to easily and flexibly integrate one-shot NAS methods into various tasks. To make users focus on algorithm design, OSNASLib provides the interface generator to generate interfaces for any designated component. Researchers can fairly evaluate and compare one-shot NAS methods based on the same codebase and same evaluation configurations. To make OSNASLib more robust for various deep learning tasks, we will keep supporting more baseline methods in the future.

## References

[1] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proceedings of International Conference on Learning Representations*, 2019. 1, 3, 4

[2] Xiangxiang Chu, Bo Zhang, and Ruijun Xu. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *arXiv preprint arXiv:1907.01845*, 2019. 1, 3

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2009. 3

[4] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *Proceedings of European Conference on Computer Vision*, 2020. 3, 4

[5] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments, 2007. 4

[6] Sian-Yao Huang and Wei-Ta Chu. Searching by generating: Flexible and efficient one-shot nas with architecture generator. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2021. 1, 2, 3, 4

[7] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009. 3

[8] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2019. 1, 3, 4

[9] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. In *Proceedings of IEEE International Conference on Computer Vision*, 2017. 3

[10] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *Proceedings of International Conference on Learning Representations*, 2019. 1, 3

[11] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2015. 3

[12] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2019. 1, 2, 3, 4

[13] Dong Yi, Zhen Lei, Shengcai Liao, and Stan Z. Li. Learning face representation from scratch, 2014. 3, 4