NASLIB: A MODULAR AND FLEXIBLE NEURAL ARCHITECTURE SEARCH LIBRARY

Michael Ruchte[1*], **Arber Zela**[1*], **Julien Siems**[1], **Josif Grabocka**[1], **& Frank Hutter**[1,2]
[1]Department of Computer Science, University of Freiburg
[2]Bosch Center for Artificial Intelligence
{ruchtem, zelaa, siemsj, grabocka, fh}@cs.uni-freiburg.de

ABSTRACT

Neural Architecture Search (NAS) is one of the focal points for the Deep Learning community, but reproducing NAS methods is extremely challenging due to numerous low-level implementation details. To alleviate this problem we introduce *NASLib*, a NAS library built upon PyTorch. This framework offers high-level abstractions for designing and reusing search spaces, interfaces to benchmarks and evaluation pipelines, enabling the implementation and extension of state-of-the-art NAS methods with a few lines of code. The modularized nature of NASLib allows researchers to easily innovate on individual components (e.g., define a new search space while reusing an optimizer and evaluation pipeline, or propose a new optimizer with existing search spaces). As a result, NASLib has the potential to facilitate NAS research by allowing fast advances and evaluations that are by design free of confounding factors. To demonstrate that NASLib is a sound library, we implement and achieve state-of-the-art results with one-shot NAS optimizers (DARTS and GDAS) over the DARTS search space and the popular NAS-Bench-201 benchmark. Last but not least, we showcase how easily novel approaches are coded in NASLib, by training one-shot optimizers on a hierarchical search space. We open source our code at https://github.com/automl/NASLib.

1 INTRODUCTION

The remarkable spread of deep learning in research and applications during the last decade is linked to its ability to learn useful representations in an end-to-end fashion, in contrast to features engineered by humans. Following the successes of more complex hand-designed architectures, such as AlexNet (Krizhevsky et al., 2012), VGG (Simonyan & Zisserman, 2015), Inception (Szegedy et al., 2015), Resnet (He et al., 2016) and MobileNet (Tan et al., 2019), Neural Architecture Search (NAS, see Elsken et al. (2019) for a review) as a sub-field of AutoML (Hutter et al., 2019) is the logical next step to automate the learning of representations by automating the time-consuming manual design of neural architectures.

Despite clear indications of the advantages that NAS could bring to Deep Learning systems (Zoph & Le, 2017; Zoph et al., 2018; Real et al., 2019; Liu et al., 2019b; Pham et al., 2018; Saikia et al., 2019; So et al., 2019; Liu et al., 2019a), research in this field has initially been constrained by the industry-level computational resources it required (Zoph & Le, 2017). A decisive improvement to increasing the efficiency of NAS was provided by the weight-sharing paradigm (Pham et al., 2018; Brock et al., 2018; Liu et al., 2019b) which creates a joint parameterization over all architectures in a search space and allows the discovery of efficient architectures in less than a day.

Apart from the required compute power, an additional burden to conduct NAS research has been the lack of a shared standard code base which can express the diversity of search spaces and search methods using an intuitive interface (Lindauer & Hutter, 2019). The DeepArchitect library (Negrinho et al., 2019) advanced the status-quo with a first NAS library for discrete NAS. However, the framework does not include current weight-sharing (one-shot) NAS methods. So far, the code base of DARTS (Liu et al., 2019b) has been frequently used as a starting point for development

[*]Equal contribution

(e.g. for GDAS (Dong & Yang, 2019), RandomNAS (Li & Talwalkar, 2019), PC-DARTS (Xu et al., 2020), PDARTS (Chen et al., 2019), SNAS (Xie et al., 2019), etc). The main drawback of using this non-standardized, non-modular code base is its implicit search space representation using nested function calls representing the search space components, which severely limits the transferability of implemented NAS optimizers to other search spaces. Furthermore, many methods also use different, incompatible, representations of the search space; for example, some embed the operation choices in the nodes of a directed acyclic graph (DAG) (Zoph et al., 2018; Bender et al., 2018; Pham et al., 2018), while others embed them in the DAG's edges (Liu et al., 2019b; Xu et al., 2020; Dong & Yang, 2019). Furthermore, some use a fixed number of edges in the graph (Liu et al., 2019b) while others can find architectures with e.g. up to a fixed maximum of edges (Ying et al., 2019). A unified search space representation that all of these approaches could act on (and be compared on) would therefore be a major step forwards for NAS research.

In order to resolve the aforementioned burdens raised by the lack of a standardized, flexible and modular NAS framework, we introduce NASLib, a new library developed to make NAS research extendable and reproducible with minimal code engineering efforts, while also allowing apples-to-apples comparisons without confounding factors (Lindauer & Hutter, 2019) by design. Our library combines the flexibility of PyTorch (Paszke et al., 2019) with the graph framework NetworkX (Hagberg et al., 2008) to explicitly create nested graphs that express the search space (e.g., hierarchical and cell search spaces) in an intuitive and modular manner.

Overall, the features of NASLib can be summarized as:

- It is an easy-to-use modular library which provides a flexible and powerful framework both for designing novel NAS methods or search spaces with minimal prototyping time, as well as facilitating the implementation of baseline papers on a common ground, in order to conduct fair experimental comparisons, eliminate redundant engineering and foster reproducibility (Li & Talwalkar, 2019; Lindauer & Hutter, 2019; Yang et al., 2020).
- It streamlines the implementation of prominent state-of-the-art NAS methods, both black-box (discrete optimizers) and weight-sharing ones (one-shot optimizers).
- It provides a unified interface to commonly used cell search spaces, hierarchical search spaces, and additionally tabular NAS benchmarks, such as NAS-Bench-201 which allows designing novel search spaces in a flexible way by means of high-level graph abstractions and manipulations.
- It implements optimizers in a way that is agnostic to the search space, allowing to change the optimizer or the search space with one line of code. This modular structure permits painless combinations of components from different published methods. For example, in Section 4, we showcase the ease with which NASLib allows to use the DARTS optimizer (Liu et al., 2019b) on a hierarchical search space (Liu et al., 2018), which to the best of our knowledge is the first application of a oneshot method to a hierarchical search space.
- It incorporates mature implementations of common search spaces, NAS optimizers and evaluation pipelines. In Section 4 we demonstrate that NAS methods coded in NASLib match the empirical performances of the codes released by the respective papers' authors.

2 Related Work

A whole line of research and engineering has been focused on automating machine/deep learning pipelines and remove the need for the machine learning expert in the loop Hutter et al. (2019). Derivatives of these community efforts include AutoML systems, such as Auto-WEKA (Thornton et al., 2013), auto-sklearn (Feurer et al., 2015), TPOT (Olson et al., 2016), or AutoGluon-Tabular (Erickson et al., 2020), which mainly focus on non-deep learning pipelines that work on tabular data. With the advent of NAS, a special focus has been given to developing tools that can facilitate the deployment of various NAS algorithms for researchers. Popular examples are Auto-Keras (Jin et al., 2019), which uses network morphisms (Wei et al., 2016) to optimize the neural network architecture; Auto-PyTorch Tabular (Zimmer et al., 2020), which uses multi-fidelity meta-learning to jointly optimize the architecture of feed-forward networks on tabular data along with their hyperparameters; the NAS API in AutoGluon (Klein et al., 2020), which only supports multi-fidelity optimizers, such as BOHB (Falkner et al., 2018) or ASHA (Li & Talwalkar, 2019); and

NNI[1], a broad collection of tools related to AutoML. While these libraries aim to simplify the practical everyday use of machine learning, none of them aims to support machine learning experts who develop NAS methods or aim to apply NAS to a new problem domain. Of the above, only NNI includes one-shot NAS methods (Pham et al., 2018; Liu et al., 2019b), but as a collection of methods rather than as a toolbox to facilitate further developments.

The main purpose of NASLib is to unify and simplify NAS research, allowing NAS practitioners to focus on novel ideas rather than spending time on low-level implementation issues. This is motivated by the benefits which standard libraries have delivered in other fields of machine learning, such as *PyTorch Geometric* (Fey & Lenssen, 2019) for Graph Convolutional Neural Networks, or *RLLib* (Liang et al., 2017) for reinforcement learning.

The *DeepArchitect* library (Negrinho et al., 2019) was an important first step towards this direction in the NAS community. It describes the key property that a NAS library should keep its search space definition independent from the optimizer being used. However, DeepArchitect is not compatible with one-shot NAS methods (and not being maintained; the last commit is more than one year ago). NASLib offers more flexibility in the search space objects and extends them further to be compatible with both one-shot and discrete NAS optimizers. Concurrently to this work, we are aware of two other (so far unpublished) libraries that are being developed to facilitate NAS research by providing a collection of different abstractions levels such as, NAS optimizers, search spaces or evaluators: *archai*[2] and *aw_nas*[3]. While we explicitly welcome the development of these related open-source NAS libraries, NASlib is unique in the great flexibility of its search space, which we expect to substantially facilitate NAS research.

3 NASLib: Architecture and Building Blocks

In this section we provide an overview of the NASLib architecture and then go into more detail on the basic building blocks, showcased also with code snippets from the library.

3.1 NASLib Architecture Overview

NASLib was designed to have the search space and optimizers completely disentangled. This allows to easily benchmark various NAS optimizers on different search spaces (including the ones from tabular NAS benchmarks (Ying et al., 2019; Dong & Yang, 2020)) via the same API and in just a few lines of code.

```
from naslib.search_spaces import DartsSearchSpace
from naslib.optimizers import DARTSOptimizer
from naslib.defaults.trainer import Trainer

config = utils.get_config_from_args()

search_space = DartsSearchSpace()
optimizer = DARTSOptimizer(config)

optimizer.adapt_search_space(search_space)
trainer = Trainer(optimizer, config)
trainer.search()
trainer.evaluate()
```

Snippet 1: A minimal example on how one can run both the search and evaluation for DARTS using NASLib. Note that both the search space and the optimizer can be changed in one line of code.

Snippet 1 shows a minimal example on how one can run both the search and evaluation for a NAS optimizer (DARTS (Liu et al., 2019b) in this case) using NASLib. Given a configuration object (line 5), which defines all the hyperparameters and other search or evaluation pipeline settings, we can use that to instantiate a NAS optimizer (line 8). A search space object can be imported from the pre-defined ones existing already in NASLib such as the DARTS cell search space (line 7), or be defined in a custom manner by the user (see Section 3.2). We again emphasize that the optimizers and search spaces are agnostic to one another, in the sense that one can import any optimizer from `naslib.optimizers` and run them on any search space. After the search space and optimizer objects are instantiated, they interact with each other via the `adapt_search_space` method of the optimizer (line 10). The third component of naslib is the `Trainer` (line 11), which is responsible for the search (line 12) and final evaluation (line 13) phase of every NAS algorithm.

[1] `https://github.com/microsoft/nni`
[2] `https://github.com/microsoft/archai`
[3] `https://github.com/walkerning/aw_nas`

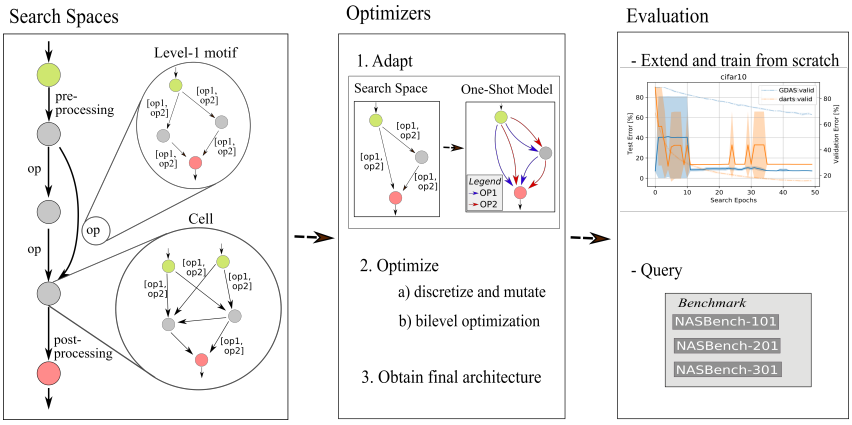

Figure 1: An high-level overview of NASLib which contains 3 main building blocks that interact with each other: (1) Search spaces, (2) Optimizers and (3) Trainers/Evaluators.

An overview of NASLib is given in Figure 1. It shows the main building blocks: search spaces, optimizers and evaluation. The library follows the natural steps of NAS: first define the search space, then the optimizer (which performs architecture search and aims to determine the best architecture), and finally the evaluation of the final architecture. We now discuss each of these buildings blocks.

3.2 NASLIB SEARCH SPACES

The search space representation is of primary importance for NASLib in ensuring that optimizers and search spaces can be combined in a variety of ways. The predominant way of representing NAS search spaces is the directed acyclic graph (DAG). While it can be easily implemented as a computational graph in PyTorch (Paszke et al., 2019), it is the composition of various components (e.g., cells in a macro architecture, graphs nested at multiple levels) in a search space which might require an additional multi-level graph representation, to ensure that optimizers and search spaces remain separate. The importance of this abstract representation is underlined by the rapid recent developments in methods inspired by DARTS (Liu et al., 2019b), which usually integrate their contributions into the DARTS code. NASLib aims to foster the reusability of search methods for different search spaces, thereby eliminating the overhead incurred by the search space construction code.

NASLib's search space builds upon NetworkX (Hagberg et al., 2008), a well-maintained and tested Python package for graph creation and manipulation, where node and edge attributes can be arbitrary Python objects. As a consequence, the complete search space definition is done in one place, using recursive calls of the same object that represents both high-level level abstractions such as the *macro* graph, or low-level ones such as operations in edges and/or nodes of the DAG. Overall, one of the main principles in NASLib is to offer maximum freedom for altering the graph while ensuring that a researcher does not have to worry about the cumbersome details. Below we briefly describe some of the main features of the search space objects in NASLib.

Flexible search space objects. One of the main components of NASLib is the `Graph` class, which can cover a wide range of search spaces using the NetworkX encoding. For constructing the search space object, `Graph` recursively calls itself, by repeatedly placing `Graph` objects on edges or nodes (see Figure 1, left). After the graph object has been instantiated, NASLib takes care of parsing it as a PyTorch module, as well as traversing, copying and updating it.

```
def set_scope(self, scope: str):
    self.scope = scope
    for g in self._get_child_graphs(
     single_instances=False):
        g.scope = scope
    return self
```

Snippet 2: `set_scope` sub-routine that sets the scope of all NASLib graph components.

Scopes. Via the concept of scopes the researcher can treat instances of the same graph differently, e.g., for setting different channels for different parts of the network. A popular example would include the macro cell search space in DARTS. That would involve 3 scopes, since there are 2

reduction cells which reduce the resolution by a factor of 2 throughout the network. The scopes of all cell components are set by iteratively calling the `set_scope` method (see Snippet 2).

Shared and private attributes. NASLib ensures proper handling of nested and copied graphs and supports shared (e.g., architectural weights α) and private attributes (e.g., convolutional weights w).

Hooks. Hooks are provided by the framework to be able to handle edge cases, such as removing edges before discretization or increasing the number of cells per stage for evaluation.

NASLib already comes with the implementation of the following popular search spaces: DARTS (Liu et al., 2019b), Nas-Bench-201 (Dong & Yang, 2020), and the hierarchical search space by Liu et al. (2018). More search spaces will be added soon.

3.2.1 Case Study: the Nas-Bench-201 search space

Snippet 3 shows how the NAS-Bench-201 (Dong & Yang, 2020) cell search space can be defined in NASLib. We label the DAG nodes by indices starting from 1 to 4 and add edges $u, v \in \{1, 2, 3, 4\}$ such that $u < v$. Nodes without an incoming edge are identified as input nodes, and the node with the highest index is identified as output node. Up to this point the operations at the edges and nodes are the ones we defined as default: Identity at the edges and summation as combine operation at the nodes.

Setting the primitive operations requires to handle different settings for each stage of the overall search space, e.g., to set the channels. Therefore, NASLib uses the concept of scopes to differentiate instances of the same graph and select graphs to be optimized (i.e., in cases where we do not want to optimize the macro graph). Additionally, edges can be flagged as final so they will not be altered by the optimizer. For instance, this is used in the DARTS search space for the edges connecting intermediate nodes with the output node.

```
class NasBench201SearchSpace(Graph):
  OPTIMIZER_SCOPE = [
    "stage_1",
    "stage_2",
    "stage_3", ]

  def __init__(self):
    super().__init__()
    # Cell definition
    cell = Graph()
    cell.name = "cell"
    cell.add_nodes_from([1, 2, 3, 4])
    cell.add_edges_densly()

    # Macro graph omitted for brevity

    # Set primitives as ops at the cells
    channels = [16, 32, 64]
    for c, scope in zip(channels,
        self.OPTIMIZER_SCOPE):
      self.update_edges(
        update_func=lambda current_edge_data:
          _set_cell_ops(
            current_edge_data, C=c),
        scope=scope,
        private_edge_data=True
      )
```

Snippet 3: The Nas-Bench-201 cell search space written using the language in NASLib. The macro graph definition is omitted for brevity. The search space is entirely defined as a graph object.

Since the same cell is used at different levels, NASLib offers private (e.g., w) and shared attributes (e.g., α), which allows proper handling of attributes even for possibly deeply nested graphs. This approach is implemented by executing a user-defined function on each edge in the given scope, e.g., setting the primitives in Snippet 3 (in line 21 note `private_edge_data=True` because of the private weights w of the primitive operations). We refer to Snippet 6 in the appendix for the full search space definition. The same logic is used by the optimizers to adapt the search space for their requirements (see Section 3.3).

3.2.2 Case Study: a hierarchical search space

With NASLib, one can also easily construct hierarchical search spaces, such as the one from Liu et al. (2018) (see Snippet 5 in the appendix). We create all motifs and the macro graph as *Graph* instances with their specified topology (Liu et al., 2018). Since NASLib natively handles graphs at edges the same way as primitives, we can build the search space bottom-up: level-1 primitive operations are used on the edges of level-2 motifs, and level-2 motifs are used on the edges of level-3 motifs, etc. In Snippet 5, setting the primitives for level-1 motifs is done last, as they require the number of channels at initialization. By setting copies of one initial motif at edges, NASLib ensures that architectural parameters are shared between the instances. The hierarchical search space can now be optimized by one-shot or discrete optimizers as any other search space.

3.3 NASLIB OPTIMIZERS

A search space graph defines the boundary of the area an optimizer can search, i.e., by the connections between the nodes or the available operations (primitives), such as convolutional layers, identity mappings, etc. The optimizer adapts the search space by its specific logic, performs the search, and determines the final architecture (which is still a `naslib Graph` object) to be evaluated.

This is realized by using the framework functions `update_edges` and `update_nodes`. Both apply a user-defined function on each edge or node for each graph in the scope. For instance, by sampling a shared index of a primitive at a given edge and then later setting the operation on each edge according to the index, the optimizer can discretize the architectural search space. By adding a shared architecture weight and setting the operation on each edge as the weighted sum of the primitives, the optimizer can realize a continuous relaxation of the space (Liu et al., 2019b) (see Snippet 4). Note again that this is completely search space agnostic.

3.3.1 CASE STUDY: THE DARTS OPTIMIZER

One example of altering the search space for optimization purposes is the continuous relaxation done by DARTS (Liu et al., 2019b). Parts of its implementation in NASLib are shown in Snippet 4. We make use of the framework by applying two functions on the graph: `add_alphas` and `update_ops`. The function `add_alphas` adds the architectural weight α at each edge of the search space (given it is in the scope); α is shared between different instances of possible cells or motifs, this is why `private_edge_data=False` in line 21. The function `update_ops` replaces the primitives at the edges with the DARTS-specific continuous mix operation, which completes the preparation of the search space for the architecture search.

Next, the architectural parameters are stored for the architecture optimizer (lines 30-33) and the graph is parsed as a PyTorch network which then allows to access its parameters w. For the update step and determining the final architecture see the more comprehensive Snippet 7 in the appendix.

```
class DARTSOptimizer(MetaOptimizer):
  @staticmethod
  def add_alphas(edge_data):
    alpha = nn.Parameter(1e-3 * randn(
        [len(edge_data.op)],
        requires_grad=True
    ))
    # alpha shared across copies
    edge_data.set('alpha', alpha,
      shared=True)

  def adapt_search_space(self,
      sspace, scope):

    graph = sspace.clone()

    # 1. add alphas
    graph.update_edges(
      self.add_alphas,
      scope=scope,
      private_edge_data=False)

    # 2. replace primitives with mixed_op
    graph.update_edges(
      self.update_ops,
      scope=scope,
      private_edge_data=True)

    # store alphas for optimizer
    self.architectural_weights = [
      a for a in
      graph.get_all_edge_data('alpha')
    ]

    graph.parse()   # convert to pytorch
    # store weights for optimizer
    weights = graph.parameters()
```

Snippet 4: Example showing how the DARTS optimizer adapts the search space to its requirements.

3.4 ADDITIONAL FEATURES

The final architecture can then either be evaluated by using a standardized training pipeline and measuring the final performance or queried via an interface to tabular benchmarks which are both provided in NASLib. This allows for a quick development of new NAS algorithms and comparisons to existing algorithms that are free of confounding factors by design. Also, NASLib provides comfort features, such as checkpointing, data loading and preprocessing currently for three datasets (CIFAR-10, CIFAR-100, SVHN), and configuration via a yaml configuration file and command line arguments. Additionally, NASLib captures searching and evaluation statistics and logs them to dedicated files.

Finally, NASLib also comes with a unified interface to several NAS benchmarks. Currently available benchmarks are NAS-Bench-201 (Dong & Yang, 2020) and NAS-Bench-301 (Siems et al., 2020), which can be used to query anytime results or cheaply train discrete optimizers without having to change a single line of code when switching the search space and its corresponding benchmark API.

4 EMPIRICAL EVALUATION

We evaluate NASLib on several search spaces. First, we reproduce published results for several optimizers on two commonly-used search spaces: the space of NAS-Bench-201 (Dong & Yang, 2020) and the DARTS cell search space (Zoph et al., 2018). Using NAS-Bench-201 allows us to obtain results quickly for discrete optimizers that have high computational demands, and the DARTS search space allows us to either run the DARTS evaluation pipeline or query NAS-Bench-301 (Siems et al., 2020). Additionally, due to its modular design, NASLib allows us to also run DARTS and GDAS on a hierarchical search space (Liu et al., 2018); to the best of our knowledge, this is the first application of any one-shot method to this type of space.

We refer to Appendix A.1 for the experimental setup.

NAS-Bench-201. We first validate our implementation on NAS-Bench-201. We implement four optimizers, DARTS (Liu et al., 2019b), GDAS (Dong & Yang, 2019), Random Search (Bergstra & Bengio, 2012) and Regularized Evolution (Real et al., 2019). Table 1 compares the results of NASLib to the reported results by Dong & Yang (2020) on CIFAR-10. We achieve on par results for GDAS and better results for DARTS and Regularized Evolution.

Table 1: Results for final architecture found by NASLib optimizers on NAS-Bench-201 CIFAR-10. We archive at least on par performance compared to (Dong & Yang, 2019).

Optimizer	**NASLib** test acc	**(Dong & Yang, 2019)** test acc
DARTS	86.26 $\pm$ 0.11	54.30 $\pm$ 0.00
GDAS	93.09 $\pm$ 0.63	93.51 $\pm$ 0.13
RE	94.23 $\pm$ 0.14	93.92 $\pm$ 0.10
RS	94.17 $\pm$ 0.19	93.70 $\pm$ 0.36

Figure 2 shows anytime performance of the two optimizers during the search and the test error of the optimal architecture transferred to CIFAR-100 and ImageNet16x16-200 (Chrabaszcz et al., 2017). The results closely resemble the original ones from the NAS-Bench-201 paper. GDAS outperforms DARTS on average and achieves a test error close to Requlrized Evolution. Figure 3 shows anytime performance of discrete optimizers Regularized Evolution and Random Search.

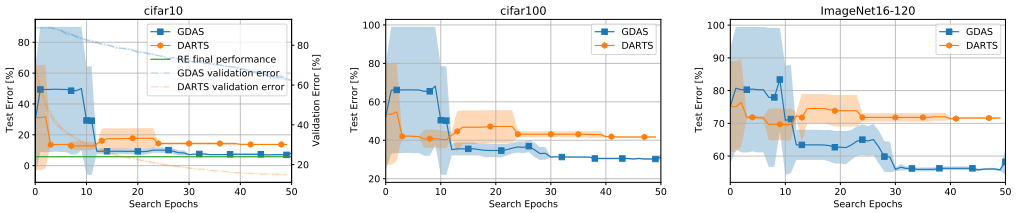

Figure 2: Results for searching with DARTS, GDAS in the NAS-Bench-201 search space on CIFAR10 for 4 random seeds each. The performance of the intermediate architectures as queried from NAS-Bench-201 is plotted on the left axis and the validation error of the one-shot model is shown on the right axis (lighter colors). As NAS-Bench-201 contains the results for architectures on three different datasets we can directly assess how well the architectures found on CIFAR10 (*left* plot) transfer to CIFAR100 (*middle* plot) and ImageNet16-120 (*right* plot). We show the first 50 epochs.

DARTS search space. We also evaluate NASLib on the DARTS search space to alleviate possible artifacts introduced by the benchmarks. Table 2 shows the test error achieved by NASLib compared to the one reported by the respective authors of DARTS and GDAS. We achieve similar performance for both optimizers compared to the original implementations, which validates our implementation of both the optimizers and of the search space (see Figure 5 in appendix for the cell found by

Table 2: Results for final architecture found by NASLib optimizers on DARTS search space for CIFAR-10. We archive on par performance compared to the respective author's implementation (as reported in (Liu et al., 2019b; Dong & Yang, 2019)).

Space	**Optimizer**	**NASLib implementation**			**Author's implementation**		
		Test Error	Params (M)	Search Cost (D)	Test Error	Params	Search Cost
DARTS	DARTS	3.26 $\pm$ 0.14	2.84	1.4	3.00 $\pm$ 0.14	3.3	1.5
	GDAS	3.28 $\pm$ 0.17	3.17	0.8	2.82 $\pm$ 0.13	2.5	0.8
Hierarch.	DARTS	3.31 $\pm$ 0.09	7.94	2.75	-	-	-
	GDAS	3.28 $\pm$ 0.02	10.73	0.29	-	-	-
	Evolution	3.38 $\pm$ 0.14	10.71	-	3.75 $\pm$ 0.12	-	300

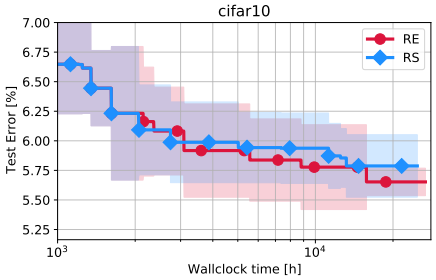

Figure 3: Discrete optimizers regularized evolution and random search on Nas-Bench 201 for CIFAR-10.

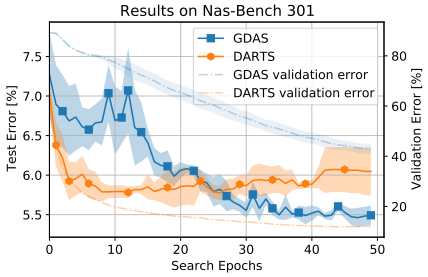

Figure 4: Anytime performance of DARTS and GDAS queried on Nas-Bench 301.

DARTS). Note that the different search cost for GDAS is due to the fact the we report the search cost for 4 runs which are needed to determine the final architecture following (Liu et al., 2019b). In Figure 4, we also show the queried anytime performance of DARTS and GDAS from the recent NAS-Bench-301 surrogate benchmark (Siems et al., 2020) on the DARTS search space.

Hierarchical search space. We also evaluate NASLib on the hierarchical search space to demonstrate its flexibility regarding the search space definition. While Liu et al. (2018) ran evolution for 300 GPU days, due to resource constraints, we could only afford to run the much faster one-shot optimizers on this search space. DARTS achieved a test error of $3.31\% \pm 0.09\%$ and GDAS a test error of $3.28\% \pm 0.02\%$ (single search, 2 seeds for evaluation). This is superior to the architecture found by Liu et al. (2018) trained on our pipeline which achieved $3.38\% \pm 0.14\%$ test error (see appendix for experimental results).

To the best of our knowledge, this is the first time any one-shot model has been run on a hierarchical search space. This reduced the computational demand by a factor of $1000\times$, from 300 to 0.3 GPU days!

5 Conclusions

In this paper we presented NASLib, a modular open source library which offers a unified platform for implementing the state-of-the-art NAS methods. Even though NAS has experienced an explosion of interest by the research community, to date there is no researcher-friendly library that enables the reuse of NAS concepts and source code in a modular fashion.

We explained NASLib's core architecture and its building blocks in depth along with exemplary source code snippets that demonstrate the simplicity of implementing prominent baselines. To verify the performance of the library, we implemented a series of state-of-the-art methods and conducted multiple experiments on three datasets using a popular NAS benchmark search space (NAS-Bench-201). The empirical findings suggest that it is easy to implement recent methods in NASLib and actually achieve competitive results, at the benefit of requiring only a fraction of the usual coding efforts. As a result, we believe NASLib has the potential to be a game-changer in the field of Neural Architecture Search, facilitating rapid prototyping, code sharing, easy applications to different types of applications and search spaces, and reproducible research.

In the future, we will enrich NASLib not only by integrating and evaluating more NAS search spaces and optimizers, but also by adding more features that will help encapsulating further many NAS paradigms. We also plan to integrate analysis tools, e.g. automatically computing correlations between proxy and true performance metrics (Zela et al., 2020; Yu et al., 2020) or curvature information in the architecture space (Zela et al., 2020). We believe that NASLib will become a valuable framework to research community and will become a community project jointly developed by the research community.

REFERENCES

Proceedings of the International Conference on Learning Representations (ICLR'17), 2017. Published online: iclr.cc.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, 2018.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. 13:281–305, 2012.

Andrew Brock, Theo Lim, J.M. Ritchie, and Nick Weston. SMASH: One-shot model architecture search through hypernetworks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rydeCEhs-.

Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1294–1303, 2019.

Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*, 2017.

Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four GPU hours. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR'19)*, pp. 1761–1770, 2019.

Xuanyi Dong and Yi Yang. Nas-bench-102: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HJxyZkBKDr.

J. Dy and A. Krause (eds.). *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, volume 80, 2018. Proceedings of Machine Learning Research.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search. pp. 69–86. Springer, 2019.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

S. Falkner, A. Klein, and F. Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In Dy & Krause (2018), pp. 1437–1446.

M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NeurIPS'15)*, pp. 2962–2970, 2015.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman (eds.), *Proceedings of the 7th Python in Science Conference*, pp. 11 – 15, Pasadena, CA USA, 2008.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

F. Hutter, L. Kotthoff, and J. Vanschoren (eds.). *Automatic Machine Learning: Methods, Systems, Challenges*. Challenges in Machine Learning. Springer, 2019.

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

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs.LG]*, 2014.

Aaron Klein, Louis Tiao, Thibaut Lienart, Cedric Archambeau, and Matthias Seeger. Model-based asynchronous hyperparameter and neural architecture search. *arXiv preprint arXiv:2003.10865*, 2020.

A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger (eds.), *Proceedings of the 26th International Conference on Advances in Neural Information Processing Systems (NeurIPS'12)*, pp. 1097–1105, 2012.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In A. Globerson and R. Silva (eds.), *Proceedings of the 35nd conference on Uncertainty in Artificial Intelligence (UAI'19)*, pp. 129. AUAI Press, 2019.

Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

Marius Lindauer and Frank Hutter. Best practices for scientific research on neural architecture search. *arXiv preprint arXiv:1909.02453*, 2019.

Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L. Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 82–92, 2019a.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJQRKzbA-.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019b. URL https://openreview.net/forum?id=S1eYHoC5FX.

I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *Proceedings of the International Conference on Learning Representations (ICLR'17)* icl (2017). Published online: iclr.cc.

Renato Negrinho, Darshan Patil, Nghia Le, Daniel Ferreira, Matthew Gormley, and Geoffrey Gordon. Towards modular and programmable architecture search. *Neural Information Processing Systems*, 2019.

R. Olson, N. Bartley, R. Urbanowicz, and J. Moore. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In T. Friedrich (ed.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'16)*, pp. 485–492. ACM, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In Dy & Krause (2018), pp. 4092–4101.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pp. 4780–4789, 2019.

T. Saikia, Y. Marrakchi, A. Zela, F. Hutter, and T. Brox. Autodispnet: Improving disparity estimation with automl. In *IEEE International Conference on Computer Vision (ICCV)*, 2019. URL http://lmb.informatik.uni-freiburg.de/Publications/2019/SMB19.

Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. NAS-Bench-301 and the Case for Surrogate Benchmarks for Neural Architecture Search. *arXiv:2008.09777 [cs.LG]*, August 2020.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

David So, Quoc Le, and Chen Liang. The evolved transformer. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 5877–5886, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/so19a.html.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

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

C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In I. Dhillon, Y. Koren, R. Ghani, T. Senator, P. Bradley, R. Parekh, J. He, R. Grossman, and R. Uthurusamy (eds.), *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*, pp. 847–855, 2013.

Tao Wei, C. Wang, Y. Rui, and C. Chen. Network morphism. In *ICML*, 2016.

Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*, 2019. Published online: iclr.cc.

Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJlS634tPr.

Antoine Yang, Pedro M. Esperança, and Fabio M. Carlucci. Nas evaluation is frustratingly hard. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HygrdpVKvr.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 7105–7114, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/ying19a.html.

Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1loF2NFwr.

Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. Understanding and robustifying differentiable architecture search. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1gDNyrKDS.

L. Zimmer, M. Lindauer, and Frank Hutter. Auto-pytorch tabular: Multi-fidelity metalearning for efficient and robust autodl. *ArXiv*, abs/2006.13799, 2020.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR'17)* icl (2017). Published online: iclr.cc.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

A APPENDIX

A.1 EXPERIMENTAL SETUP

NAS-Bench-201. We ran the DARTS and GDAS search for 50 epochs on CIFAR-10 with batch size 64, using NAS-Bench-201's standard split of data points into 50% test and 50% training/validation for the search. Following Dong & Yang (2020), as weight optimizer we use SGD with momentum 0.9, weight decay of 3×10^{-4} and initial learning rate 0.025 with cosine annealing (Loshchilov & Hutter, 2017) to 0.001. As optimizer for the architectural weights, we use Adam Kingma & Ba (2014) with learning rate 3×10^{-4} and weight decay of 0.001. For GDAS we set the initial $\tau = 10$ and reduce linearly to 0.1 and trained it for 250 epochs. For regularized evolution (Real et al., 2019) and Random Search, we set the number of function evaluations to 1000, population size to 100 and sample size 10.

DARTS search space. We also run DARTS and GDAS on this search space using the same hyperparameters as for NAS-Bench-201 which are also the same as used in (Liu et al., 2019b). For training the final model found by the optimizer we used the pipeline from Liu et al. (2019b) with their hyperparameters (Cutout, drop path, auxiliary towers) using the full 50000 images training set. The final architecture found by GDAS includes more parameters which is why we set the batch size for this to 72 instead of 96. A single search took 7.5 hours, evaluation 12 to 51 hours on a single GPU. Following (Liu et al., 2019b; Dong & Yang, 2019) we pick the best out of four searches after low-fidelity evaluation on the validation set and train it from scratch for longer and with more initial channels and stacked cells.

Hierarchical search space. We ran DARTS and GDAS for 10 epochs with batch size 10 on CIFAR-10 using the search space as Liu et al. (2018). Apart from this we used the same hyperparameters as for DARTS. The resulting search took 66 and 7 hours on a single GPU for DARTS and GDAS, respectively, in contrast to the dramatically longer runs for 1.5 days on 200 GPUs in Liu et al. (2018). Different to Liu et al. (2018), we trained the found cells and motifs on 50000 images for 600 epochs using batch size of 32 and SGD with initial learning rate of 0.025 with cosine annealing to 0.001. We use cutout and drop path with probability of 0.2. The evaluation took 23 hours on a single GPU.

A.2 NASLIB CODE SNIPPETS

Here we show more code snippets from NASLib.

```
class HierarchicalSearchSpace(Graph):
  OPTIMIZER_SCOPE = [
    "stage_1",
    "stage_2",
    "stage_3"]

  def __init__(self):
    super().__init__()
    level2_motifs = []  # 6 level-2 motifs
    for j in range(6):
      motif = Graph()
      motif.name = "motif{}".format(j)
      motif.add_nodes_from([1, 2, 3, 4, 5])
      motif.add_edges_densly()
      level2_motifs.append(motif)

    cell = Graph()       # 1 level-3 motif
    cell.name = "cell"
    cell.add_nodes_from([1, 2, 3, 4, 5, 6])
    cell.add_edges_densly()

    cells = []
    channels = [16, 32, 64]
    for c, s in zip(channels,
                    self.OPTIMIZER_SCOPE):
      cell_i = cell.copy()
      # place level 2 motifs as op
      cell_i.update_edges(
        update_func=lambda current_edge_data:
            _set_motifs(current_edge_data,
                motifs=level2_motifs, c=c),
        private_edge_data=True
      )
      cell_i.set_scope(s, recursively=True)
      # place level 1 motifs, i.e. primitives
      cell_i.update_edges(
        update_func=lambda current_edge_data:
            _set_cell_ops(current_edge_data,
                c, stride=1),
        scope=s,
        private_edge_data=True
      )
      cells.append(cell_i)

    # Macro graph omitted for brevity
```

Snippet 5: The hierarchical search space written using the language in NASLib. The macro graph definition is omitted for brevity. The search space is entirely defined as instances of graphs.

```python
import torch.nn as nn
from naslib.search_spaces.core import primitives as ops
from naslib.search_spaces.core.graph import Graph, EdgeData
from naslib.search_spaces.core.primitives import AbstractPrimitive
from .primitives import ResNetBasicblock

class NasBench201SeachSpace(Graph):

    OPTIMIZER_SCOPE = [
        "stage_1",
        "stage_2",
        "stage_3",
    ]

    QUERYABLE = True

    def __init__(self):
        super().__init__()

        # Cell definition
        cell = Graph()
        cell.name = "cell"
        cell.add_node(1)             # Input node
        cell.add_node_from([2, 3])   # Intermediate nodes
        cell.add_node(4)             # Output node
        cell.add_edges_densly()      # Edges

        # Macro graph definition
        self.name = "makrograph"

        total_num_nodes = 20
        self.add_nodes_from(range(1, total_num_nodes+1))
        self.add_edges_from([(i, i+1) for i in range(1, total_num_nodes)])

        # operations at the edges
        channels = [16, 32, 64]

        self.edges[1, 2].set('op', ops.Stem(channels[0]))    # preprocessing
        # stage 1
        for i in range(2, 7):
            self.edges[i, i+1].set('op', cell.copy().set_scope('stage_1'))
        # stage 2
        self.edges[7, 8].set('op', ResNetBasicblock(channels[0], channels[1], stride=2))
        for i in range(8, 13):
            self.edges[i, i+1].set('op', cell.copy().set_scope('stage_2'))
        # stage 3
        self.edges[13, 14].set('op', ResNetBasicblock(channels[1], channels[2], stride=2))
        for i in range(14, 19):
            self.edges[i, i+1].set('op', cell.copy().set_scope('stage_3'))
        # post-processing
        self.edges[19, 20].set('op', ops.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(channels[-1], self.num_classes)
        ))

        # set the ops at the cells (channel dependent)
        for c, scope in zip(channels, self.OPTIMIZER_SCOPE):
            self.update_edges(
                update_func=lambda current_edge_data: _set_cell_ops(current_edge_data, C=c),
                scope=scope,
                private_edge_data=True
            )


    def query(self, metric=None, dataset=None, path=None):
        # query logic here


def _set_cell_ops(current_edge_data, C):
    current_edge_data.set('op', [
        ops.Identity(),
        ops.Zero(stride=1),
        ops.ReLUConvBN(C, C, kernel_size=3),
        ops.ReLUConvBN(C, C, kernel_size=1),
        ops.AvgPool1x1(kernel_size=3, stride=1),
    ])
```

Snippet 6: The complete Nasbench 201 cell search space written using the language in NASLib. The query implementation is omitted for brevity. The search space is entirely defined as graph object.

```
class DARTSOptimizer(MetaOptimizer):

    @staticmethod
    def add_alphas(current_edge_data):
        len_primitives = len(current_edge_data.op)
        alpha = Parameter(1e-3 * torch.randn(size=[len_primitives], requires_grad=True))
        current_edge_data.set('alpha', alpha, shared=True)

    @staticmethod
    def update_ops(current_edge_data):
        primitives = current_edge_data.op
        current_edge_data.set('op', MixedOp(primitives))

    def adapt_search_space(self, search_space, scope=None):
        graph = search_space.clone()    # We are going to modify the search space

        if not scope:
            scope = graph.OPTIMIZER_SCOPE   # use the search space default one

        # 1. add alphas
        graph.update_edges(self.add_alphas, scope, private_edge_data=False)
        # 2. replace primitives with mixed_op
        graph.update_edges(self.update_ops, scope, private_edge_data=True)

        for alpha in graph.get_all_edge_data('alpha'):
            self.architectural_weights.append(alpha)

        graph.parse()

        # Init optimizers
        self.arch_optimizer = self.arch_optimizer(self.architectural_weights.parameters(),
            lr=self.config.arch_learning_rate, betas=(0.5, 0.999),
            weight_decay=self.config.arch_weight_decay)
        self.op_optimizer = self.op_optimizer(graph.parameters(),
            lr=self.config.learning_rate, momentum=self.config.momentum,
            weight_decay=self.config.weight_decay)

        graph.train()
        self.graph = graph
        self.scope = scope

    def step(self, data_train, data_val):
        input_train, target_train = data_train
        input_val, target_val = data_val

        # Update architecture weights
        self.arch_optimizer.zero_grad()
        logits_val = self.graph(input_val)
        val_loss = self.loss(logits_val, target_val)
        val_loss.backward()
        clip_grad_norm_(self.architectural_weights.parameters(), self.grad_clip)
        self.arch_optimizer.step()

        # Update op weights
        self.op_optimizer.zero_grad()
        logits_train = self.graph(input_train)
        train_loss = self.loss(logits_train, target_train)
        train_loss.backward()
        clip_grad_norm_(self.graph.parameters(), self.grad_clip)
        self.op_optimizer.step()

        return logits_train, logits_val, train_loss, val_loss

    def get_final_architecture(self):
        graph = self.graph.clone().unparse()
        graph.prepare_discretization()     # e.g. darts sspace: only 2 in-edges with max alpha

        def discretize_ops(current_edge_data):
            if current_edge_data.has('alpha'):
                primitives = current_edge_data.op.get_embedded_ops()
                alpha = current_edge_data.alpha.detach().cpu()
                current_edge_data.set('op', primitives[np.argmax(alpha)])

        graph.update_edges(discretize_ops, self.scope, private_edge_data=True)
        graph.prepare_evaluation()
        graph.parse()
        return graph
```

Snippet 7: The DARTS optimizer as implemented in NASLib. Non-important aspects are ommited for brevity.

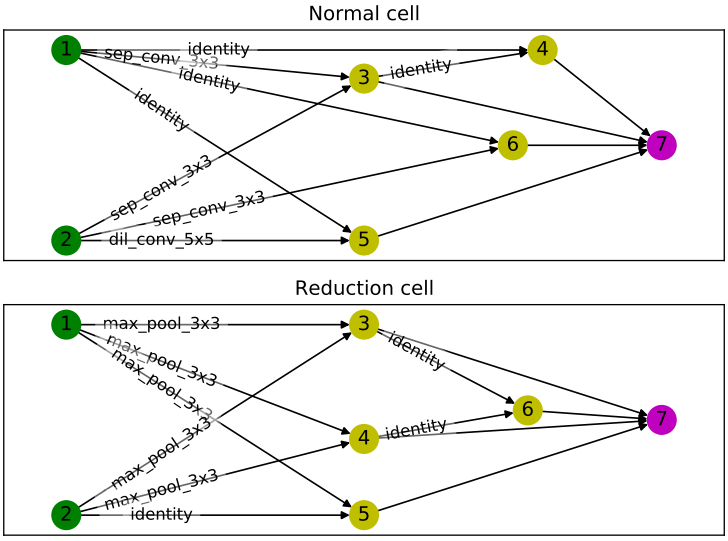

Figure 5: Normal and reduction cell found by DARTS on CIFAR-10.