4-1-2022

# Exploring the Efficiency of Neural Architecture Search (NAS) Modules

Joshua Dulcich
*Andrews University*, dulcich@andrews.edu

J. N. Andrews Honors Program
Andrews University

HONS 497
Honors Thesis

# Exploring the efficiency of Neural Architecture Search (NAS) modules

Joshua Dulcich
April 1, 2022

Advisor: Rodney Summerscales Ph.D.

Primary Advisor Signature: [illegible]

Department: Computing

# Exploring the efficiency of Neural Architecture Search (NAS) modules

Joshua Dulcich

***Abstract*—Machine learning is obscure and expensive to develop. Neural architecture search (NAS) algorithms automate this process by learning to create premier ML networks, minimizing the bias and necessity of human experts. From this recently emerging field, most research has focused on optimizing a promisingly unique combination of NAS's three segments. Despite regularly acquiring state of the art results, this practice sacrifices computing time and resources for slight increases in accuracy; this also obstructs performance comparison across papers. To resolve this issue, we use NASLib's modular library to test the efficiency per module in a unique subset of combinations. Each NAS algorithm produces an ML image recognition model—tested on the CIFAR-10 dataset—and compared for efficiency, a ratio between compute time and accuracy.**

## I. Introduction

The substantial price tag for practically implementing machine learning not only indicates its value, but the complexity of the process. Currently, the standard method comprises of experts with years of education repetitively guessing and checking their intuition one model at a time. To improve this paradigm, neural architecture search (NAS) automates the development of machine learning (ML) algorithms via primary ML algorithms. Intelligently searching hundreds of thousands of models, this automation of the process has been continuously setting new state of the art (SOTA) results across countless applications. As tech companies have caught wind of these breakthroughs, significant resources have been poured into its development; however, this source of motivation drives most research in two directions. Firstly, to get an initial return on such investments, research is focused on delivering a highly optimized specific implementation for their single use case. This practice discards generalizability that would otherwise allow comparisons between research and growth for those areas that were comparably more promising. Secondly, they focus on discovering the most accurate models they can, rarely prioritizing the computational power or time necessary. As much of the population can't access super-computing power, the ability to use NAS algorithms is extremely limited, despite these huge increases of resources often netting only a slight increase in accuracy. Therefore, there is an explicit lack of research regarding the efficiency of NAS algorithms. This work aims to build on a foundation of efficiency focused NAS research, by testing unique NAS algorithms and exploring the effects of the individual modules.

## II. Background

Due to the complexity of tasks from speech translation to image generation, ML algorithms are easier to understand when depicted as graphs. This can be visualized as a map of cities and roads, generalized as nodes and edges respectively. More specifically, they are directed acyclical graphs

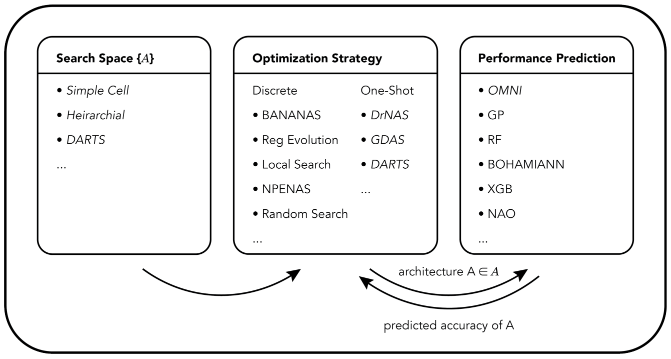


Fig. 1. The structure & flow of neural architecture search

(DAG), where all the roads are one-way, and there are no paths that lead back to anywhere you've been. Each node or edge itself can be characterized by a different mathematical function, altogether creating a unique shape that significantly translates to the accuracy of the model. These architectures are reminiscent of neural pathways, from which the algorithm derives its name. Hence, neural architecture search aims to discover the configuration that leads to the highest accuracy. NAS automates this via three steps, each of which has an ever-growing number of implementations from which to choose. See (Fig. 1). The following section provides a brief overview of the classical methods.

### *A. Search Space*

First, we define the extent of architectures to search. Searching the infinite possible shapes is infeasible. Implementing previous successful spaces is a trade-off that may optimize search but may introduce the limitations of human engineering.

As visually represented in (Fig. 2), defining a search space can be approached from various directions. The different colored nodes in the diagram each represent a different type of neural network layer, e.g., a convolutional or pooling layer. The edges that connect these layers represent the output of one layer being passed as input to the next one. In the most simple search space, depicted on the left of the diagram, a *chain-structured* space is a set of layers in a single line. The parameters that make up this space—the ones which are searched for the optimal configuration—are mainly then number of layers and type of each layer. Hyperparameters, more finely tuned parameters, are usually preset options or left out of the search for architecture. The second diagram from the left in (Fig. 2) depicts a *multi-branch network*. Recent NAS research has demonstrated the usefulness of more complex connections; skip connections that pass a layer's output further down the network, or branched connections that allow the output to be used in multiple places, have revolutionized the accuracy of models. To search these architectures, the variables for the endpoints of each connection have to be considered as well. The third model shown in (Fig. 2) is known as a *cell* or *block*. Essentially, it is a multi-branch network that is repeatedly used in place of a single layer in a *hierarchical* or *macro architecture*, as seen on the right half of (Fig. 2). Stacking these cells allows for more complex architectures while reducing the time to build a whole one. Also, finding useful cells can translate across various problems, regardless of how they're arranged in any particular architecture. Parameters for searching these macro architectures built from cells is straightforward. Cells are constructed as multi-branch networks. Macro architectures are then constructed as networks where layer type is replaced by cell type. Given these search spaces, NAS can create a breadth of useful architecture shapes while minimizing costly computations.

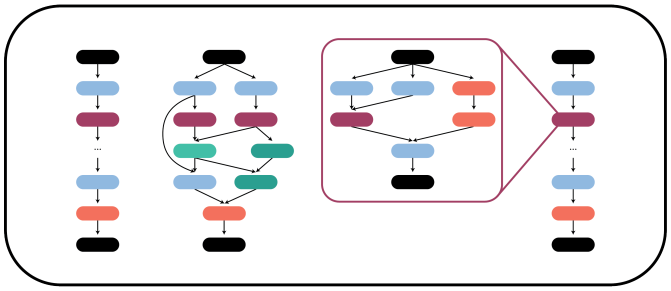


Fig. 2. Various ways to represent a search space—chain, multi-branch, cells & macro architecture

## B. Optimization Strategy

Next, given a graph, NAS determines what modification would improve the current shape—and therefore the model accuracy. It is a balance of exploration vs. exploitation; a familiar puzzle of trying to find a global max reasonably quick without converging early. Strategies fall into two main categories: discrete and one-shot.

Discrete strategies were historically the initial attempt at NAS, where searching would take the form of testing one architecture after the other. Many methods have been developed, a subset of which have been listed in (Fig. 1). *Random search* is often used as a baseline; the premise of randomness implies there exists no correlation between subsequently tested architectures. *Local search* is quite the opposite, focusing on architectures that are slight changes away from the previously tested one. These edits could be as simple as removing one layer, changing a node type, or altering a connection. *Reinforcement learning* is a family of methods that provide the accuracy of each model as a reward, the possible actions as the search space, and where the generation of the model as the action itself. The policy, or rules, by which each method employs to improve the model accuracy varies across implementations. *Neuro-evolutionary* approaches have moved towards genetic algorithms that focus on shape only, as weights are much more efficiently searched via gradient descent. Evolutionary models also have a wide spread of implementation options, like removing the worst performing or the oldest, different types of mutations, and inheritance of parameters. A slightly more complex approach, *bayesian optimization* methods attempt to optimize the accuracy of the model by only using the initial shape and final accuracy as input. Learning from how previous guesses affected the accuracy, the model learns which parameters to tune to maximize accuracy. Tree-based methods are used to effectively search the high-dimensional spaces.

*One-shot*, also known as gradient based methods, have recently taken the spotlight of much NAS research. Using an approach similar to rolling a ball down a hill, gradient descent methods find the local min (or max) of a function, in this case, the models with the highest accuracy scores. (Fig. 3) helps to visualize this process. Starting from the left, a given model has connections without specific operations. To initialize, each connection is given the option of every available type, here represented by color. As the model is trained, certain connections take precedence, similar to the way exercising for a sport will develop the muscles specific to that activity. Then at the end of training, a discrete model is chosen by removing all but the strongest connections. This idea of searching the discrete space using continuous methods has its own volume of implementations, as well as research that has seen it applied to the different steps of NAS.

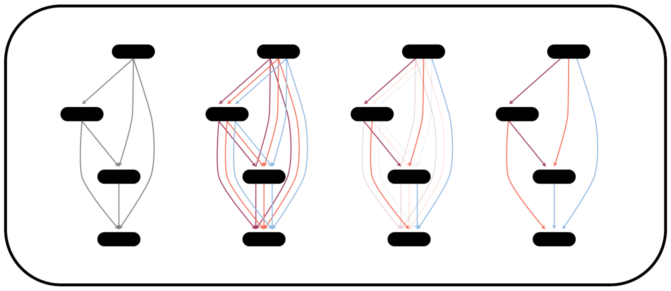


Fig. 3. An example of one-shot emergence

## C. Performance Predictor

To avoid the computational overload of fully training every graph found, we implement a predictor that estimates its model accuracy. This score is returned and guides further exploration of the space. Speed ups are typically implemented by using lower fidelity estimates, learning curve extrapolation, network morphisms, and weight sharing.

The naive way to evaluate each architecture is to fully train it. While a true accuracy score is found, this practice can turn computational hours into years. Several methods have been implemented to optimize the most accurate estimations in the shortest amount of time. *Lower fidelity estimates* is the first category, where a full training regiment is run, except that some parameters have been drastically minimized. This includes, but is not limited to, training for fewer epochs, on a small subset of the data, or downscaled models or data. The next method is *learning curve extrapolation*; poorly performing models are terminated based on how well the models are doing at given intervals, allowing the focus of resources to the more promising models. *Network morphisms* rely on the optimization of past models, as initialization parameters are passed down from parents, a method commonly associated with evolutionary optimization. One-shot models, or *weight sharing*, were briefly introduced in the previous section. Instead of training each architecture found individually, one massive model is fully trained. Then sub-models are pulled out of this massive model, as fully trained; all that is left is to test each for accuracy. One-shot methods in particular have seen some of the largest improvements in terms of performance prediction to date.

## III. Related Work

### A. NASLib

Neural architecture search is currently a focus of deep learning research, but the complex implementation is an obstacle for wider study. NASLib offers a solution by offering "high-level abstractions" for a growing number of algorithms used in each of the three steps of NAS. It also includes benchmarks and evaluation methods which allow an easier implementation of any configuration. This also provides the opportunity to develop your own modules to mix and match with current methods. The inaugural paper demonstrated the robust nature of the library by reproducing several of the current SOTA methods, showing comparable results, as well as implementing a novel approach. [1] [2]

### B. NAS: A Survey

Deep learning has been revolutionizing industry, specifically through the introduction of novel architectures. Manual creation of such machine learning models inefficient, on accounts of all resources. This need has fueled the research of automated neural architecture search methods. This paper reviews the influential works to date, as well as breaking down the components of NAS: the search space, search strategy, and performance estimation strategy. [3]

### C. NAS-Bench-Suite

NAS benchmarks have significantly "lowered the computational overhead" for research by allowing accuracy lookup of certain neural net configurations. While benchmark resources increase with the age of the field, it has recently been shown that they do not always generalize well to specific applications. This work delves into several popular algorithms and tests 25 different search space/dataset combinations. In response, the authors introduce NAS-Bench-Suite to allow more comprehensive, reproducible, and generalizable testing. [4]

### D. Performance Predictors

To avoid the time and compute required to fully train architectures, current NAS methods rely on predicting the performance of models found while searching. While numerous methods have been proposed, there did not exist a comprehensive study and comparison of these methods, prior to this paper. Analyzing 31 different techniques, the authors provide recommendations for how to best use different predictors. A final experiment shows how combining different predictors can achieve better results. [5]

### E. Surrogate Benchmarks

As the issue of large computational overhead is a recurring theme in the field of NAS, tabular benchmarks have provided a way to lookup pre-computed results. However, these tables are restricted to their discrete entries, which are few in comparison to a truly infinite search space. Surrogate benchmarks provide an answer by achieving more function-like performance prediction. Introducing a benchmark of $10^{18}$ architectures, this work is able to expand the search space while simultaneously reducing the cost. [6]

### F. Best NAS Research Practices

The value of NAS algorithms has driven a flurry of recent research. Although this rush of research has produced golden methods, the quality of such research hasn't been standardized in order to promote further research. The authors list several common issues of NAS research, as well as ways to avoid them. This led to their publication of a NAS Research Best Practices Checklist. [7] [8]

### G. Few-Shot NAS

Evaluation of networks is currently a major bottleneck of NAS research. While full vanilla training requires immense overhead, one-shot methods have proven to be considerably more efficient, even bringing original 3000+ day runtimes to just a few hours. One drawback to this method is their temperamental nature due to initialization. As an alternative to training a single super net, few-shot proposes breaking this macro architecture down into sub nets that can be individually one-shot trained, and then combining these results to provide accurate model evaluation. Setting SOTA results across the field, this paper shows a promising new direction in NAS as well as motivates the studies of my own research. [9]

## IV. Methodology

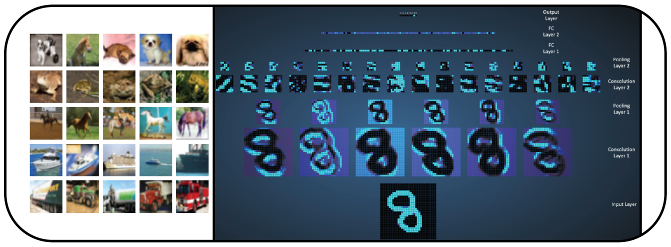


Fig. 4. (Left) Sample of CIFAR-10. (Right) Visualization of a basic image recognition net

To discover efficient modules, we graph the accuracy per compute time tradeoff between tests. NASLib is a library that is actively modularizing several of the popular algorithms used in the different steps of NAS, (Fig. 1) lists a subset of them. It has also compiled every possible major benchmark for NAS, to pioneer cross experiment comparison. Due to the modular nature, it is possible to test over 3,000 unique module combinations. For a feasible scope, a subset of these was derived from leading research: see the italicized algorithms in (Fig. 1), each producing an ML model that is tested on CIFAR-10, a standard image recognition dataset. Note the modules to be examined for efficiency in this experiment comprise a unique combination of a search space and an optimization strategy. Each of the 9 unique combinations first conducts 17,500 searches of the space, producing 175 candidate architectures. The best architecture is then evaluated for 600 epochs (520 evaluations per epoch), saving a trained model every 30 epochs.

The output of each NAS combination produces a final ML model. For standardization, the images in CIFAR-10 are grouped into 10 labeled categories, examples of 5 are shown on the left of (Fig. 4). The right side of (Fig. 4) demonstrates the workings of one of these models, known as a convolutional neural net. The name convolution is derived from the first layer demonstrated, essentially as it learns to recognize individual features of each picture. Alternating with pooling layers, the model downsizes the features in order to generalize its ability to recognize them. To guess, the model assigns a probability to each category, choosing the highest as its answer. Accuracy is recorded as Top-1 and Top-5; grading the model based on whether its top 1 or top 5 guess(es), respectively, contained the correct answer.

Experiments were conducted utilizing Google's Colab Pro+ resources. A typical run uses a Tesla P100-PCIE-16GB GPU, and RAM allocation of 16 GB. For particularly strenuous loads, single runs can be upgraded to Google's TPUs and 52 GB of RAM. Runtime per execution is capped at 24 hours.

## V. Results

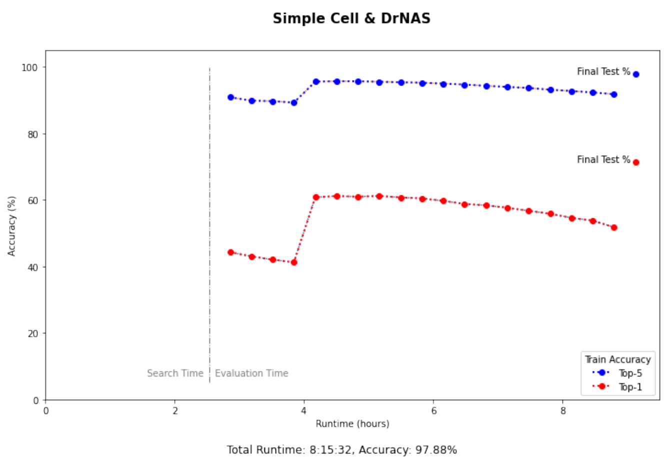


Fig. 5. Select results. See all in Appendix A

Reference (Fig. 5) for visualization of results. Note that (Fig. 5) is is exclusively the Simple Cell Search Space and DrNAS Optimizer run; the complete aftermath can be referenced in Appendix A. Each table graphs model accuracy (%) over runtime (hrs). Due to the varying nature of runtimes, observe that the x-axis isn't relatively equivalent across graphs. The titles denote which algorithms were combined, search space and optimizer respectively. There is a gray divider on the graph that represents the time it took to search the space; its appearance on a graph denotes the completion of finding 100 unique models. To the right of this split, the top-1 and top-5 accuracy of the best model is recorded every 30 epochs. The final model's structure is listed in Appendix B. Finally, the testing score is recorded at the single point for each top-1 and top-5 results. The final runtime and top-5 accuracy are emphasized under each graph. For reference, 10% is the baseline as it represents a random guess from the ten image categories.

Due to the limitations of resources at hand, every combination was not able to be run fully through. However, this does not detract from the results, as the study aims to find the most efficient modules—those that were too expensive are therefore undesirable implementations. However, the relative results gathered do provide context for the efficiency of each module individually. Note that some graph titles are followed by a * or **. Each star represents the necessity of that run to upgrade to the TPU and High-RAM. A single star indicates it was only necessary for the evaluation phase, while a double star means the entire run was high compute. Some of the results had projected runtimes out of the scope of the project, so after a reasonable runtime at consistent pace the final runtimes were extrapolated.

The data itself presented interesting trends, especially when arranging the graphs by module. The first overall theme that was realized, was since the problem of image classification on CIFAR 10 is relatively simple, the extra complexity of the networks discovered left their added usefulness for more complex problems; this also contributed to the fairly quick learn times for each model. The second large find was that GDAS and DARTS optimization both found the same network over the Simple Cell search space. This makes sense, as the same area was searched for both models, alternatively DrNAS optimization found a fairly competitive model as well so it wasn't too far removed either; accuracy difference was less than 1%.

However, it is of importance to note that it took DARTS approximately 35 minutes longer to find the same architecture. This emphasizes that GDAS is quicker than DARTS, and in this case, was more efficient as well. DrNAS was even more efficient, as its accuracy to runtime ratio was higher than either of the other optimizers. Overall, GDAS is definitely the least resource intense optimizer, as it was the only module capable of running on all three search spaces in these conditions. This is followed by DrNAS and DARTS, respective to their order as well as which one is newer.

In terms of search spaces, the fastest module was perhaps foreshadowed by its name, Simple Cell search. This was also the only search space that ran efficiently across all three optimizers. DARTS and Hierarchical spaces were in close proximity; even though 2/3 optimizers were able to search the DARTS space, the 1 optimizer on Hierarchical was able to proceed to evaluation. Runtime was the most clear differential factor between the search spaces—Simple Cell, DARTS, and Hierarchical in order from the fastest, magnitudes apart. Therefore, while Simple Cell and DrNAS was clearly the most efficient combination, it was Simple Cell and GDAS that were capable of running on limited compute regardless of which module they were combined with.

## VI. Discussion

NAS research continues to considerably improve ML, actively setting new SOTA across all applications, in addition to creating accessibility for public use. Specifically, efficient NAS methods have been under-explored as the average human lacks supercomputer access. NASLib allows for benchmarks and simpler compute requirements, as well as making it easier for non-experts to use due to its high level implementation. This novel concept of efficiency focused research across a standardized baseline is accompanied by a rating of this library's user-friendliness.

The ability of this experiment to be extended to the full scale of the library is the most valuable outcome of this research. Unfortunately, due to the scope of the project at hand and the time available, its greatest weakness was the lack of robustness from not being able to come close to the library's full potential.

With the modularity of NASLib, this experiment has the potential to be extended across every combination of module as well as iterated for several more runs per combination. In addition, theorized improvements and novel algorithms can be implemented and run through the same model of testing to compare their efficiency to the methods that already exist. This future possibility would provide a robust understanding of efficient modules, and allow future NAS research to be focused on the most promising areas.

One-shot research has been especially promising lately [9], but further limitations involve sub graphs being restricted to the overarching model, as well as having to store the entire model in memory during processing. This is typically remedied with cell based searches. In general, the field is ripe for more study to improve and prove the robustness of such methods.

NASLib markets itself as an easy to use and modular library to promote the research of NAS. Having received a chance to use it for this work, I now take the time to review the success of these claims. Here I note my in-expertise, especially as this has been my first introduction to the field of NAS. Overall, NASLib certainly put NAS research into my reach; in this it has succeeded. However, there were several aspects which I noted might make it even easier. The biggest flaw was the lack of documentation. Although I did appreciate the use of a file structure to sort the code, I still had to read through the code itself to figure out how things worked, which admittedly took up a majority of my time. Similarly, there was no concise list of usable functions or algorithms, so determining which modules were available to experiment with was a treasure hunt. The examples for running the code were brief but helpful; although the slightly difficult to understand annotations were understandable due to the creators predominantly not speaking English as a first language. The biggest strength of the library itself was the large and active user base, as it was continuously updated fairly frequently.

For future directions, curiosity lends to running the same experiment with greater compute resources. Although the data presented was sufficient to establish trends, these could be confirmed by finding a computer capable of filling in all the graphs. Similarly, the question of module efficiency can be extended to test these modules in combinations with the others in the library as well.

## References

[1] M. Ruchte, A. Zela, J. N. Siems, J. Grabocka, and F. Hutter, "Naslib: A modular and flexible neural architecture search library," 2021, unpublished Manuscript. [Online]. Available: https://openreview.net/forum?id=EohGx2HgNsA

[2] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019.

[3] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *CoRR*, vol. abs/1808.05377, 2019. [Online]. Available: https://arxiv.org/abs/1808.05377

[4] Y. Mehta, C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter, "Nas-bench-suite: NAS evaluation is (now) surprisingly easy," *CoRR*, vol. abs/2201.13396, 2022. [Online]. Available: https://arxiv.org/abs/2201.13396

[5] C. White, A. Zela, B. Ru, Y. Liu, and F. Hutter, "How powerful are performance predictors in neural architecture search?" *CoRR*, vol. abs/2104.01177, 2021. [Online]. Available: https://arxiv.org/abs/2104.01177

[6] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," *CoRR*, vol. abs/2008.09777, 2020. [Online]. Available: https://arxiv.org/abs/2008.09777

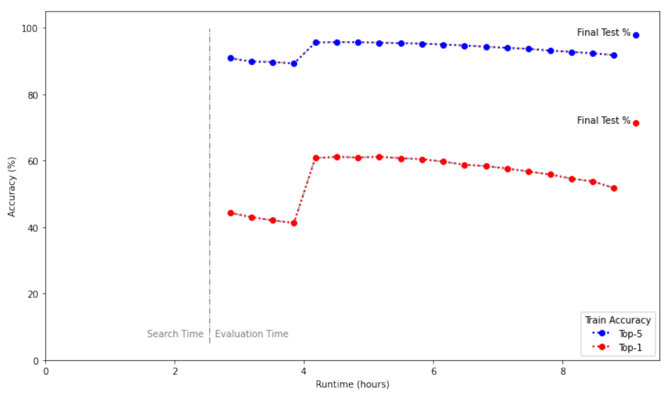
[7] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *CoRR*, vol. abs/1909.02453, 2019. [Online]. Available: http://arxiv.org/abs/1909.02453

[8] ——, "The nas best practices checklist," Available at https://www.automl.org/wp-content/uploads/NAS/NAS_checklist.pdf (2021/01/11).

[9] Y. Zhao, L. Wang, Y. Tian, R. Fonseca, and T. Guo, "Few-shot neural architecture search," *CoRR*, vol. abs/2006.06863, 2020. [Online]. Available: https://arxiv.org/abs/2006.06863

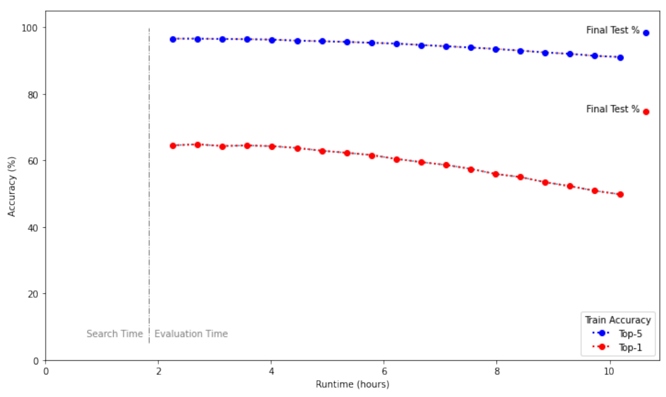
# Appendix A
# Results

**Simple Cell & DrNAS**



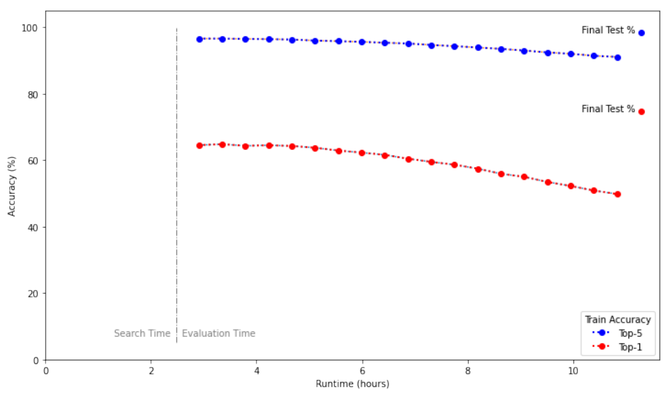
Total Runtime: 8:15:32, Accuracy: 97.88%

**Simple Cell & GDAS**



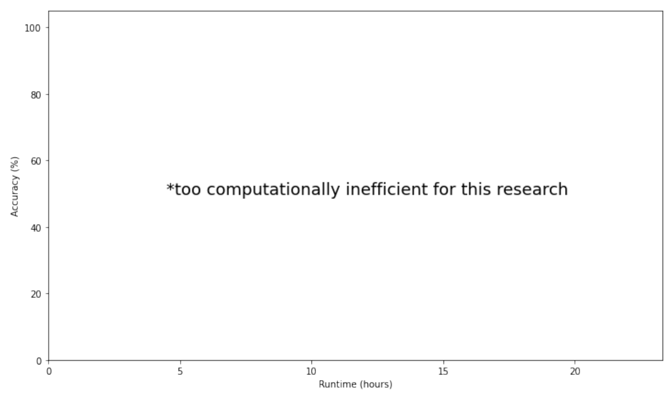
Total Runtime: 9:27:36, Accuracy: 98.47%

**Simple Cell & DARTS**



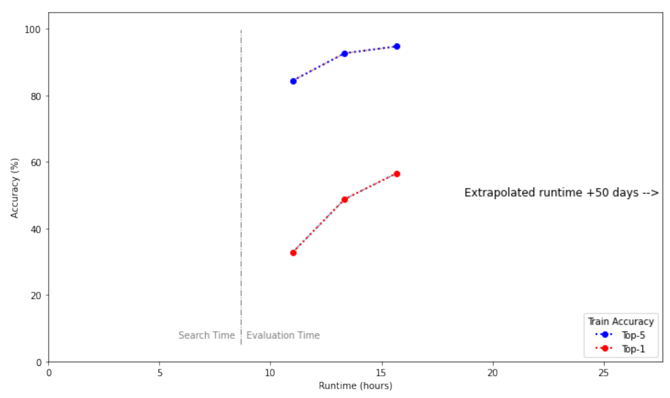
Total Runtime: 10:06:36, Accuracy: 98.47%

**Hierarchical & DrNAS****



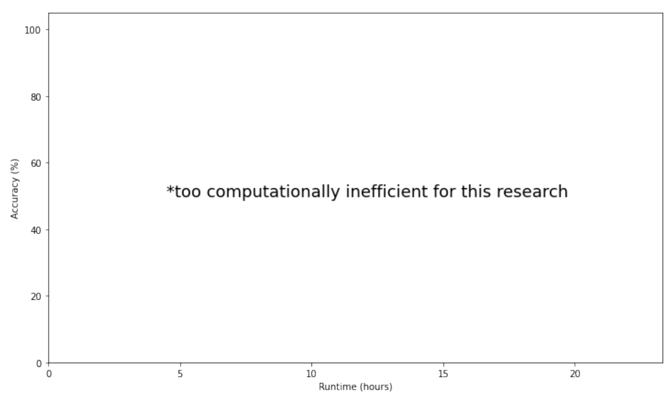
Total Runtime: n/a, Accuracy: n/a

**Hierarchical & GDAS***



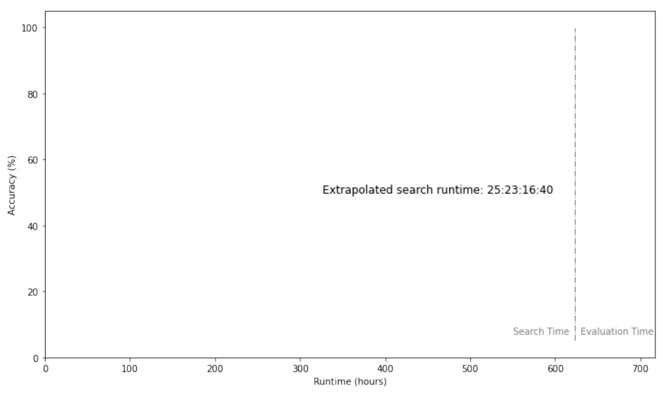
Total Runtime: 51:00:01:20, Accuracy: n/a

**Hierarchical & DARTS****



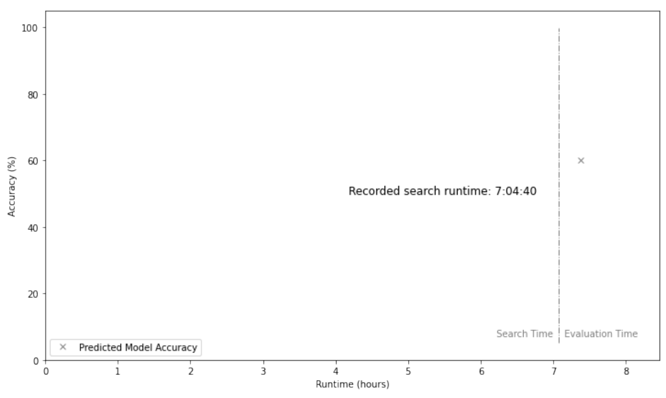
Total Runtime: n/a, Accuracy: n/a

**DARTS & DrNAS****



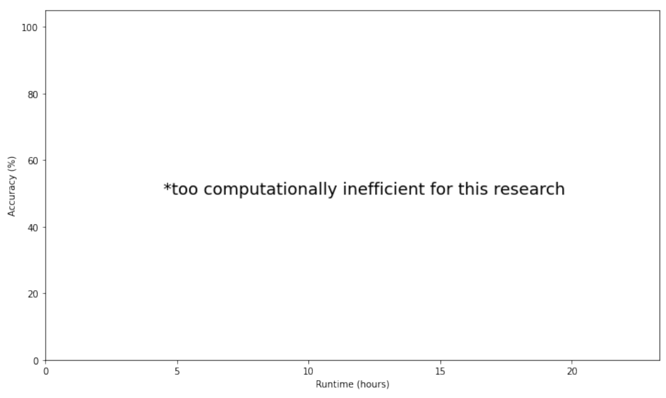
Total Runtime: 25:23:16:40+Eval, Accuracy: n/a

**DARTS & GDAS***



Total Runtime: 7:04:40+Eval, Accuracy: n/a

**DARTS & DARTS****



Total Runtime: n/a, Accuracy: n/a

## APPENDIX B
## GENERATED MODELS

### A. *Simple Cell Search Space & DrNAS Optimizer*

```
[04/06 16:06:06 nl.defaults.trainer]: Final architecture:
Graph normal_cell:
 Graph(
  (normal_cell-edge(1,3)): MaxPool1x1(
    (maxpool): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=
          False)
  )
  (normal_cell-edge(1,4)): DilConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
            dilation=(2, 2),
groups=16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
    )
  )
  (normal_cell-edge(2,3)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
      (4): ReLU()
      (5): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =16, bias=False)
      (6): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (7): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (normal_cell-edge(2,4)): DilConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
            dilation=(2, 2),
groups=16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
    )
  )
  (normal_cell-edge(3,4)): FactorizedReduce()
  (normal_cell-edge(3,5)): Identity()
  (normal_cell-edge(4,5)): Identity()
  (normal_cell-comb_op_at(5)): Concat1x1(
    (conv): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
  )
)
==========
Graph reduction_cell:
 Graph(
  (reduction_cell-edge(1,3)): Zero1x1 (stride=2)
  (reduction_cell-edge(1,4)): DilConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(2, 2),
            dilation=(2, 2),
groups=16, bias=False)
      (2): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (reduction_cell-edge(2,3)): Zero1x1 (stride=2)
  (reduction_cell-edge(2,4)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups
            =16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
      (4): ReLU()
      (5): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =16, bias=False)
      (6): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (reduction_cell-edge(3,4)): DilConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
            dilation=(2, 2),
groups=32, bias=False)
      (2): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
    )
  )
  (reduction_cell-edge(3,5)): Identity()
  (reduction_cell-edge(4,5)): Identity()
  (reduction_cell-comb_op_at(5)): Concat1x1(
    (conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
  )
)
==========
Graph makrograph:
 SimpleCellSearchSpace(
  (makrograph-edge(1,2)): Stem(
    (seq): Sequential(
      (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
            False)
      (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (makrograph-edge(2,3)): Identity()
  (makrograph-edge(2,4)): Identity()
  (makrograph-subgraph_at(3)): Graph normal_cell-0.1788023, scope stage_1, 5 nodes
  (makrograph-edge(3,4)): Identity()
  (makrograph-subgraph_at(4)): Graph reduction_cell-0.2484313, scope stage_2, 5
        nodes
  (makrograph-edge(4,5)): Identity()
  (makrograph-subgraph_at(5)): Graph normal_cell-0.7598774, scope stage_2, 5 nodes
  (makrograph-edge(5,6)): Sequential(
    (op): Sequential(
      (0): AdaptiveAvgPool2d(output_size=1)
      (1): Flatten(start_dim=1, end_dim=-1)
      (2): Linear(in_features=32, out_features=10, bias=True)
) )
) ==========
```

### B. *Simple Cell Search Space with GDAS & DARTS Optimizers*

```
[04/06 16:06:09 nl.defaults.trainer]: Final architecture:
Graph normal_cell:
 Graph(
  (normal_cell-edge(1,3)): MaxPool1x1(
    (maxpool): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=
          False)
  )
  (normal_cell-edge(1,4)): DilConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
            dilation=(2, 2),
groups=16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
    )
  )
  (normal_cell-edge(2,3)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
      (4): ReLU()
      (5): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =16, bias=False)
      (6): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (7): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (normal_cell-edge(2,4)): DilConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
            dilation=(2, 2),
groups=16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
    )
  )
  (normal_cell-edge(3,4)): FactorizedReduce()
  (normal_cell-edge(3,5)): Identity()
  (normal_cell-edge(4,5)): Identity()
  (normal_cell-comb_op_at(5)): Concat1x1(
    (conv): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
  )
)
==========
Graph reduction_cell:
 Graph(
  (reduction_cell-edge(1,3)): Zero1x1 (stride=2)
  (reduction_cell-edge(1,4)): DilConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(2, 2),
            dilation=(2, 2),
groups=16, bias=False)
      (2): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (reduction_cell-edge(2,3)): Zero1x1 (stride=2)
  (reduction_cell-edge(2,4)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups
            =16, bias=False)
      (2): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
      (4): ReLU()
      (5): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =16, bias=False)
      (6): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (reduction_cell-edge(3,4)): DilConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
            dilation=(2, 2),
groups=32, bias=False)
      (2): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
    )
  )
  (reduction_cell-edge(3,5)): Identity()
  (reduction_cell-edge(4,5)): Identity()
  (reduction_cell-comb_op_at(5)): Concat1x1(
    (conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
  )
)
==========
Graph makrograph:
 SimpleCellSearchSpace(
  (makrograph-edge(1,2)): Stem(
    (seq): Sequential(
      (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
            False)
      (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
            track_running_stats=True)
) )
  (makrograph-edge(2,3)): Identity()
  (makrograph-edge(2,4)): Identity()
  (makrograph-subgraph_at(3)): Graph normal_cell-0.1788023, scope stage_1, 5 nodes
  (makrograph-edge(3,4)): Identity()
  (makrograph-subgraph_at(4)): Graph reduction_cell-0.2484313, scope stage_2, 5
        nodes
  (makrograph-edge(4,5)): Identity()
  (makrograph-subgraph_at(5)): Graph normal_cell-0.7598774, scope stage_2, 5 nodes
  (makrograph-edge(5,6)): Sequential(
    (op): Sequential(
      (0): AdaptiveAvgPool2d(output_size=1)
      (1): Flatten(start_dim=1, end_dim=-1)
      (2): Linear(in_features=32, out_features=10, bias=True)
) )
) ==========
```

## C. Hierarchial Search Space & GDAS Optimizer

```
[04/06 06:22:47 nl.defaults.trainer]: Final architecture: Graph cell:
Graph(
(cell-edge(1,2)): Graph motif2-0.9435488, scope stage_1, 4 nodes (cell-edge(1,3)):
      Graph motif0-0.0272178, scope stage_1, 4 nodes (cell-edge(1,4)): Graph motif4
      -0.2315998, scope stage_1, 4 nodes (cell-edge(1,5)): Graph motif0-0.6082666,
      scope stage_1, 4 nodes (cell-edge(2,3)): Graph motif4-0.8348234, scope
      stage_1, 4 nodes (cell-edge(2,4)): Graph motif4-0.3795893, scope stage_1, 4
      nodes (cell-edge(2,5)): Graph motif5-0.3588370, scope stage_1, 4 nodes (cell-
      edge(3,4)): Graph motif3-0.3179757, scope stage_1, 4 nodes (cell-edge(3,5)):
      Graph motif0-0.3752663, scope stage_1, 4 nodes (cell-edge(4,5)): Graph motif3
      -0.5383149, scope stage_1, 4 nodes
)
==========
Graph motif0:
 Graph(
  (motif0-edge(1,2)): Identity()
  (motif0-edge(1,3)): SepConv(
(op): Sequential(
(0): ReLU()
(1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(6): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
) )
  (motif0-edge(1,4)): ConvBNReLU(
    (op): Sequential(
(0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (2): ReLU()
) )
  (motif0-edge(2,3)): ConvBNReLU(
    (op): Sequential(
(0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (2): ReLU()
) )
  (motif0-edge(2,4)): AvgPool1x1(
    (avgpool): AvgPool2d(kernel_size=3, stride=1, padding=1)
  )
  (motif0-edge(3,4)): Zero1x1 (stride=1)
  (motif0-comb_op_at(4)): Concat1x1(
    (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
  )
)
==========
Graph motif2:
 Graph(
  (motif2-edge(1,2)): Identity()
  (motif2-edge(1,3)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (4): ReLU()
      (5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(6): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) )
  )
  (motif2-edge(1,4)): AvgPool1x1(
    (avgpool): AvgPool2d(kernel_size=3, stride=1, padding=1)
  )
(motif2-edge(2,3)): DepthwiseConv( (op): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
(2): ReLU() )
)
(motif2-edge(2,4)): Zero1x1 (stride=1) (motif2-edge(3,4)): DepthwiseConv(
    (op): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (2): ReLU()
) )
  (motif2-comb_op_at(4)): Concat1x1(
    (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
) )
==========
Graph motif3:
 Graph(
  (motif3-edge(1,2)): AvgPool1x1(
    (avgpool): AvgPool2d(kernel_size=3, stride=1, padding=1)
  )
  (motif3-edge(1,3)): Zero1x1 (stride=1)
  (motif3-edge(1,4)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(6): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
) )
  (motif3-edge(2,3)): ConvBNReLU(
    (op): Sequential(
(0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (2): ReLU()
) )
  (motif3-edge(2,4)): Zero1x1 (stride=1)
  (motif3-edge(3,4)): ConvBNReLU(
(op): Sequential(
(0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (2): ReLU()
) )
  (motif3-comb_op_at(4)): Concat1x1(
    (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
)

)
==========
Graph motif4:
 Graph(
  (motif4-edge(1,2)): ConvBNReLU(
(op): Sequential(
(0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (2): ReLU()
) )
  (motif4-edge(1,3)): SepConv(
    (op): Sequential(
(0): ReLU()
      (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(6): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
) )
(motif4-edge(1,4)): DepthwiseConv( (op): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
(2): ReLU() )
```

```
  )
  (motif4-edge(2,3)): MaxPool1x1(
    (maxpool): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=
          False)
  )
  (motif4-edge(2,4)): MaxPool1x1(
    (maxpool): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=
          False)
  )
  (motif4-edge(3,4)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(6): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
) )
  (motif4-comb_op_at(4)): Concat1x1(
    (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
) )
==========
Graph motif5:
 Graph(
  (motif5-edge(1,2)): AvgPool1x1(
    (avgpool): AvgPool2d(kernel_size=3, stride=1, padding=1)
  )
  (motif5-edge(1,3)): MaxPool1x1(
    (maxpool): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=
          False)
)
(motif5-edge(1,4)): Zero1x1 (stride=1) (motif5-edge(2,3)): DepthwiseConv(
    (op): Sequential(

      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
(2): ReLU() )
  )
  (motif5-edge(2,4)): Zero1x1 (stride=1)
  (motif5-edge(3,4)): Zero1x1 (stride=1)
  (motif5-comb_op_at(4)): Concat1x1(
    (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
          =True)
  )
)
==========
Graph makrograph:
 HierarchicalSearchSpace(
  (makrograph-edge(1,2)): Stem(
(seq): Sequential(
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
) )
(makrograph-edge(2,3)): Graph cell-0.4523939, scope stage_1, 5 nodes (makrograph-
      edge(3,4)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
            =64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      ) (4): ReLU()
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(6): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-edge(4,5)): Graph cell-0.7797973, scope stage_1, 5 nodes (makrograph-
      edge(5,6)): SepConv(
    (op): Sequential(
      (0): ReLU()
      (1): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups
            =64,
bias=False)
(2): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      ) (4): ReLU()
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=64,
bias=False)
(6): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(6,7)): Graph cell-0.9726591, scope stage_2, 5 nodes (makrograph-
      edge(7,8)): SepConv(
(op): Sequential(
(0): ReLU()
(1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
      =128,
bias=False)
(2): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
      =128,
bias=False)
(6): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(8,9)): Graph cell-0.9303213, scope stage_2, 5 nodes (makrograph-
      edge(9,10)): SepConv(
    (op): Sequential(
      (0): ReLU()

(1): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups
      =128, bias=False)
(2): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
      =128,
bias=False)
(6): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(10,11)): Graph cell-0.0776364, scope stage_3, 5 nodes (makrograph-
      edge(11,12)): SepConv(
(op): Sequential(
(0): ReLU()
(1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
      =256,
bias=False)
(2): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True) (4): ReLU()
(5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups
      =256,
bias=False)
(6): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
  (makrograph-edge(12,13)): Graph cell-0.6049727, scope stage_3, 5 nodes
  (makrograph-edge(13,14)): Sequential(
    (op): Sequential(
      (0): SepConv(
        (op): Sequential(
          (0): ReLU()
          (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
                groups=256,
bias=False)
          (2): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
                track_running_stats=True)
          (4): ReLU()
          (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
                groups=256,
bias=False)
          (6): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
                track_running_stats=True)
) )
      (1): AdaptiveAvgPool2d(output_size=1)
      (2): Flatten(start_dim=1, end_dim=-1)
      (3): Linear(in_features=256, out_features=10, bias=True)
) )
) ==========
```

## D. DARTS Search Space & GDAS Optimizer

```
[04/06 05:58:57 nl.defaults.trainer]: Final architecture: Graph normal_cell:
 Graph(
  (normal_cell-edge(1,3)): AvgPool(
(avgpool): Sequential(
(0): AvgPool2d(kernel_size=3, stride=1, padding=1)
(1): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
) )
  (normal_cell-edge(1,5)): MaxPool(
    (maxpool): Sequential(
      (0): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=
            False)
(1): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) )
  )
  (normal_cell-edge(2,3)): AvgPool(
(avgpool): Sequential(
(0): AvgPool2d(kernel_size=3, stride=1, padding=1)
(1): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True)
) )
  (normal_cell-edge(2,4)): AvgPool(
    (avgpool): Sequential(
      (0): AvgPool2d(kernel_size=3, stride=1, padding=1)
(1): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
      True) )
  )
  (normal_cell-edge(2,6)): DilConv(
(op): Sequential(
(0): ReLU()
(1): Conv2d(36, 36, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2,
       2),
groups=36, bias=False)
(2): Conv2d(36, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
  (normal_cell-edge(3,4)): Identity()
  (normal_cell-edge(3,7)): Identity()
  (normal_cell-edge(4,5)): AvgPool(
(avgpool): Sequential(
(0): AvgPool2d(kernel_size=3, stride=1, padding=1)
(1): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
        True)
) )
  (normal_cell-edge(4,6)): DilConv(
     (op): Sequential(
(0): ReLU()
(1): Conv2d(36, 36, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2,
         2), groups=36, bias=False)
(2): Conv2d(36, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
       (3): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True,
              track_running_stats=True)
    )
  )
  (normal_cell-edge(4,7)): Identity()
  (normal_cell-edge(5,7)): Identity()
  (normal_cell-edge(6,7)): Identity()
)
==========
Graph reduction_cell:
 Graph(
  (reduction_cell-edge(1,3)): AvgPool(
(avgpool): Sequential(
(0): AvgPool2d(kernel_size=3, stride=2, padding=1)
(1): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
        True)
) )
  (reduction_cell-edge(1,4)): FactorizedReduce(
(relu): ReLU()
(conv_1): Conv2d(72, 36, kernel_size=(1, 1), stride=(2, 2), bias=False)
(conv_2): Conv2d(72, 36, kernel_size=(1, 1), stride=(2, 2), bias=False)
(bn): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
        True)
  )
  (reduction_cell-edge(1,6)): DilConv(
(op): Sequential(
(0): ReLU()
(1): Conv2d(72, 72, kernel_size=(3, 3), stride=(2, 2), padding=(2, 2), dilation=(2,
         2),
groups=72, bias=False)
(2): Conv2d(72, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
  (reduction_cell-edge(2,3)): SepConv(
     (op): Sequential(
(0): ReLU()
       (1): Conv2d(72, 72, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups
             =72,
bias=False)
(2): Conv2d(72, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        ) (4): ReLU()
(5): Conv2d(72, 72, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=72,
bias=False)
(6): Conv2d(72, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
  (reduction_cell-edge(2,4)): SepConv(
     (op): Sequential(
(0): ReLU()
(1): Conv2d(72, 72, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), groups=72,
        bias=False)
(2): Conv2d(72, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        ) (4): ReLU()
(5): Conv2d(72, 72, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), groups=72,
bias=False)
(6): Conv2d(72, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(reduction_cell-edge(2,5)): FactorizedReduce(
(relu): ReLU()
(conv_1): Conv2d(72, 36, kernel_size=(1, 1), stride=(2, 2), bias=False)
(conv_2): Conv2d(72, 36, kernel_size=(1, 1), stride=(2, 2), bias=False)
(bn): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
        True)
  )
  (reduction_cell-edge(3,7)): Identity()
  (reduction_cell-edge(4,5)): DilConv(
(op): Sequential(
(0): ReLU()
(1): Conv2d(72, 72, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2,
         2),
groups=72, bias=False)
(2): Conv2d(72, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
  (reduction_cell-edge(4,7)): Identity()
  (reduction_cell-edge(5,6)): AvgPool(
(avgpool): Sequential(
(0): AvgPool2d(kernel_size=3, stride=1, padding=1)
(1): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
        True)
) )
  (reduction_cell-edge(5,7)): Identity()
  (reduction_cell-edge(6,7)): Identity()
)
==========
Graph makrograph:
 DartsSearchSpace(

(makrograph-edge(1,2)): Stem(
  (seq): Sequential(
    (0): Conv2d(3, 108, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
           False)
(1): BatchNorm2d(108, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
        True) )
)
```

```
(makrograph-edge(2,3)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(108, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(2,4)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(108, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(2,5)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(108, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(3,4)): Identity()
(makrograph-subgraph_at(4)): Graph normal_cell-0.2000754, scope n_stage_1, 7 nodes
        (makrograph-edge(4,5)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(4,6)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-subgraph_at(5)): Graph normal_cell-0.1788023, scope n_stage_1, 7 nodes
        (makrograph-edge(5,6)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(5,7)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-subgraph_at(6)): Graph normal_cell-0.7744736, scope n_stage_1, 7 nodes
        (makrograph-edge(6,7)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(6,8)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )

) )
(makrograph-subgraph_at(7)): Graph normal_cell-0.4690390, scope n_stage_1, 7 nodes
        (makrograph-edge(7,8)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(7,9)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-subgraph_at(8)): Graph normal_cell-0.4026227, scope n_stage_1, 7 nodes
        (makrograph-edge(8,9)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(144, 36, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(36, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(8,10)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(144, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-subgraph_at(9)): Graph normal_cell-0.2693992, scope n_stage_1, 7 nodes
        (makrograph-edge(9,10)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(144, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        )
) )
(makrograph-edge(9,11)): FactorizedReduce(
  (relu): ReLU()
  (conv_1): Conv2d(144, 36, kernel_size=(1, 1), stride=(2, 2), bias=False)
  (conv_2): Conv2d(144, 36, kernel_size=(1, 1), stride=(2, 2), bias=False)
  (bn): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
        True)
)
(makrograph-subgraph_at(10)): Graph reduction_cell-0.2000754, scope r_stage_1, 7
      nodes (makrograph-edge(10,11)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-edge(10,12)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-subgraph_at(11)): Graph normal_cell-0.2484313, scope n_stage_2, 7 nodes
(makrograph-edge(11,12)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-edge(11,13)): ReLUConvBN(
  (op): Sequential(

(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-subgraph_at(12)): Graph normal_cell-0.7598774, scope n_stage_2, 7 nodes
(makrograph-edge(12,13)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-edge(12,14)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-subgraph_at(13)): Graph normal_cell-0.0076198, scope n_stage_2, 7 nodes
(makrograph-edge(13,14)): ReLUConvBN(
  (op): Sequential(
    (0): ReLU()
    (1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
          True)
) )
(makrograph-edge(13,15)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-subgraph_at(14)): Graph normal_cell-0.2037644, scope n_stage_2, 7 nodes
(makrograph-edge(14,15)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-edge(14,16)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      )
) )
(makrograph-subgraph_at(15)): Graph normal_cell-0.9895700, scope n_stage_2, 7 nodes
(makrograph-edge(15,16)): ReLUConvBN(
  (op): Sequential(
    (0): ReLU()
    (1): Conv2d(288, 72, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (2): BatchNorm2d(72, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
          True)
) )
(makrograph-edge(15,17)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(288, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-subgraph_at(16)): Graph normal_cell-0.5343100, scope n_stage_2, 7 nodes
(makrograph-edge(16,17)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(288, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)

) )
(makrograph-edge(16,18)): FactorizedReduce(
(relu): ReLU()
(conv_1): Conv2d(288, 72, kernel_size=(1, 1), stride=(2, 2), bias=False)
(conv_2): Conv2d(288, 72, kernel_size=(1, 1), stride=(2, 2), bias=False)
(bn): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
)
(makrograph-subgraph_at(17)): Graph reduction_cell-0.2511510, scope r_stage_2, 7
      nodes (makrograph-edge(17,18)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(17,19)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-subgraph_at(18)): Graph normal_cell-0.3830676, scope n_stage_3, 7 nodes
(makrograph-edge(18,19)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(18,20)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-subgraph_at(19)): Graph normal_cell-0.6843086, scope n_stage_3, 7 nodes
(makrograph-edge(19,20)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(19,21)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-subgraph_at(20)): Graph normal_cell-0.4620867, scope n_stage_3, 7 nodes
(makrograph-edge(20,21)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-edge(20,22)): ReLUConvBN(
  (op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
(makrograph-subgraph_at(21)): Graph normal_cell-0.9002423, scope n_stage_3, 7 nodes
(makrograph-edge(21,22)): ReLUConvBN(
  (op): Sequential(

(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
  (makrograph-edge(21,23)): ReLUConvBN(
    (op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
  (makrograph-subgraph_at(22)): Graph normal_cell-0.6055091, scope n_stage_3, 7
        nodes
  (makrograph-edge(22,23)): ReLUConvBN(
(op): Sequential(
(0): ReLU()
(1): Conv2d(576, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(2): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True)
) )
  (makrograph-subgraph_at(23)): Graph normal_cell-0.9767119, scope n_stage_3, 7
        nodes
  (makrograph-edge(23,24)): Sequential(
(op): Sequential(
(0): ReLU(inplace=True)
(1): AvgPool2d(kernel_size=5, stride=3, padding=0)
(2): Conv2d(576, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
(3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True) (4): ReLU(inplace=True)
(5): Conv2d(128, 768, kernel_size=(2, 2), stride=(1, 1), bias=False)
(6): BatchNorm2d(768, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      True) (7): ReLU(inplace=True)
(8): Flatten(start_dim=1, end_dim=-1)
(9): Linear(in_features=768, out_features=10, bias=True)
) )
  (makrograph-edge(23,25)): Sequential(
    (op): Sequential(
      (0): AdaptiveAvgPool2d(output_size=1)
      (1): Flatten(start_dim=1, end_dim=-1)
      (2): Linear(in_features=576, out_features=10, bias=True)
) )
) ==========
```