A Genetic Programming Approach To Zero-Shot Neural Architecture Ranking

Anonymous Author(s)
Affiliation
Address
email

Abstract

Neural networks are becoming increasingly ubiquitous in a wide range of use cases. A primary hurdle in deploying neural networks in many scenarios is the tedious and difficult neural network architectural design process, which was reliant on expert knowledge and iterative design. Neural Architecture Search (NAS) reduces the human effort required for design, but still has considerable resource requirements and is extremely slow. To address the inefficiencies of conventional NAS, Zero-Shot NAS is a new paradigm, which introduces zero shot neural architecture scoring metrics (NASMs) to identify good neural network designs without training them. While applying Zero Shot NASMs is cheap and requires no training resources, we identify that there is a lack of NASMs that generalize well across neural architecture design spaces. In this paper, we present a program representation for NASMs and automate its search with genetic programming. We discover effective NASMs for Image Classification as well as Automatic Speech Recognition. We believe that our work indicates a new direction for NASM design and can greatly benefit from recent advances in program synthesis.

1 Introduction

As the diversity of hardware platforms and end use-cases of neural networks increase, practical deployment of neural networks is becoming increasingly difficult. Maximizing accuracy of neural network architectures is no longer the sole concern, with significant emphasis being placed on the deployment efficiency and the carbon footprint of discovering efficient architectures. Further, hardware performance metrics are often non-differentiable in nature as these costs can vary arbitrarily depending on the the memory access pattern, cache hierarchy and other several factors. [1, 2] Given the irregular deployment costs of neural networks in hardware aware settings, the efficiency of differentiable neural architecture design is significantly lower than in accuracy aware settings. The sample efficiency of non-differentiable methods also suffer due to the irregular nature of the loss landscape of hardware performance.

Zero-shot Neural Architecture Search aims to alleviate the training costs in the Neural Architecture Search process by 'scoring' neural networks at initialization. Ideally, a higher score should correspond to a higher test accuracy of the neural network after training. We refer to algorithms that can score neural network architectures without training them as *Zero Shot Neural Architecture Scoring Metrics* (NASMs). There exist several zero-shot NASMs such as NASWOT, ZenNAS, AngleNAS, TENAS and SynFlow [3, 4, 5, 6, 7]. Design of these NASMs are driven by human intuition or are theoretically inspired by discovering metrics that quantify the *trainability* and *expressivity* of neural networks. [4] empirically studies metrics such as `synflow`, which was introduced as a techinque to pruning neural network weights at initialization based on a saliency metric [8]. NASWOT [3] treats output of each layer of a neural network as a binary indicator (zero if value is negative, one if value is positive) and

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

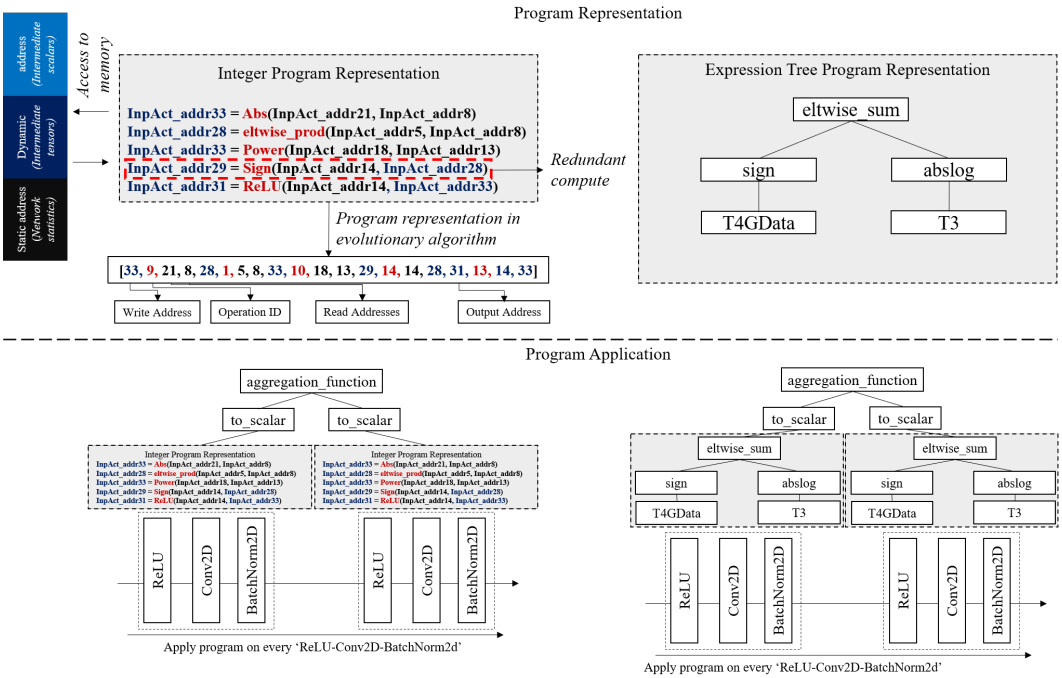


Figure 1: (Left) Integer program representation is unconstrained and introduces redundant operations, leading to program bloating. It also has limited dynamic scalar and tensor memory addresses. (Right) An expression tree structure introduces fewer redundancies and more interpretable programs.

uses the hamming distance between two binary codes induced by an untrained network at two inputs as a measure of dissimilarity. This is driven by the intuition that the more similar two inputs are, the more challenging it should be for the network to learn how to separate them. Such methods of scoring neural network architectures can work well for a small sub-set of neural architecture design spaces, but in practice often fail to transfer across NN design spaces. These methods may also impose structural restrictions on the neural networks that limit the architecture design space. For instance, ZenNAS [6] truncates the neural network representation by removing residual connections. This limits the ability to distinguish differing connectivity patterns of the neural network architecture.

In this paper, we identify a domain specific language (DSL) which can effectively represent existing SoTA NASMs. We also identify an appropriate choice of input-output example and formulate the discovery of programs (NASMs) with the DSL as a program synthesis task driven by evolutionary search. This method allows us to discover NASMs that impose minimal restrictions on the neural network representation and task. Our contributions are as follows:

- Utilizing an expression tree structure to represent NASMs in a manner that is sufficiently expressive to capture the task of neural architecture ranking across neural architecture design spaces and tasks.
- Demonstration of our genetic programming driven methodology (EZNAS - **E**volutionarily Generated **Z**ero Shot **N**eural **A**rchitecture **S**coring Metric) to automate zero-shot NASM design for Image Classification and Automatic Speech Recognition.
- Outline of future work to motivate research in more robust input-output example representation, DSL design and program search strategies.

2 Evolutionary Framework

Domain Specific Language (DSL). DSLs are computer languages that are suitable for a specialized domain and are more restrictive than full-features programming languages [9]. An appropriate DSL

construction is crucial to discovering NASMs efficiently. We explored two alternative program representations for evolutionary search of NASMs.

Our initial choice of DSL was inspired by AutoML-Zero [10] and posed minimal structural restrictions on the program. The program representation and application is depicted in Figure 1 (Left). A total of 22 tensors were generated at every `ReLU-Conv2D-BatchNorm2D` (*RCB tensors*) instance of the neural network (weights, activations and gradients). Each NASM was applied to every RCB instance and the score was generated by averaging the score assigned by the NASM at every RCB instance of the neural network. We had 22 static memory addresses to store the RCB tensors (referenced with integers `0-21`), 80 dynamic memory addresses to store intermediate tensors generated by the program and 20 dynamic memory addresses to store intermediate scalars. The fitness of each individual program is calculated by measuring the Kendall Tau rank correlation between the scores generated of 80 neural network architectures, and their test accuracy. The test accuracy is readily available to us as we use public NAS data-sets (NASBench-201 [11] and NDS [12]).

As seen in Figure 1 (Left), each NASM was sequentially represented as a set of integers, with every four integers indicating an expression with a result, operation ID and two operands. We initialize *valid* random integer arrays and convert them to programs to evaluate and fetch the fitness. *Valid* integer arrays are the arrays that can be converted to programs and generate a float score. We allow `Mate, InsertOP, RemoveOP, MutateOP & MutateOPArg` as variation functions described further in the appendix. While we discover weak NASMs with this formulation, we observe that there are too many redundancies in the programs discovered. Program length bloating as well as operations that do not contribute to the final output made the run time intractable. In EZNAS, each individual NASM has to be evaluated on several gigabytes of input-output examples with no approximations to generate precise fitness values.

To reduce the computational complexity of search, we necessitate an *expression tree* structure on the NASM program to capture the executional ordering of the program. As depicted in Figure 1 (Right), the program output appears at a root node, and the child (terminal) nodes are the *arguments* of the expression tree. These *arguments* are the 22 RCB tensors. The advantage of this program representation is that there is only a single root node with dense connectivity form the root to terminal nodes, leading to lesser redundancies. As the expression tree describes the executional ordering of the mathematical operations available to us, it is crucial to provide a varied set of mathematical operations. We provide 34 unique operations in our program search space, including basic mathematical operations such as Addition, Product, Log as well as operations such as Cosine Similarity, KL Divergence and Hamming Distance. We provide the full list of mathematical operations available in the supplementary material.

Search Technique. Our search algorithm discovers programs by modifying the expression tree representation. At each generation, fitness score is generated for every expression tree (NASM) in the population. A selection and variation algorithm is then used to generate new offspring. We utilize the VarOr implementation from *Distributed Evolutionary Algorithms in Python* (DEAP) [13] framework for variation of individual programs. We generate `n` *valid* offspring programs at each generation. $n/2$ offspring are generated as a result of three operations; crossover, mutation or reproduction. To encourage diversity, we also randomly generate $n/2$ valid individuals at every generation. Invalid programs that either fail to execute or give `inf, nan` outputs are replaced by new individuals. We give an algorithmic description of our search method in the Appendix.

Image Classification						Automatic Speech Recognition	
NASM	Amoeba	DARTS	ENAS	PNAS	NASNet	NASM	ASR TIMIT
`EZNAS-A`	**0.42**	**0.55**	**0.51**	**0.48**	**0.43**	`EZNAS-ASR`	0.33
`NASWOT`	0.22	0.47	0.37	0.38	0.3	`SNIP`	0.01
`grad_norm`	-0.1	0.28	-0.02	-0.01	-0.08	`grad_norm`	0.07
`syn_flow`	-0.06	0.37	0.02	0.03	-0.03	`syn_flow`	**0.4**

Table 1: (Left) Kendall Tau rank correlation for the image classification task on NDS [12] Neural Architecture Design Space. (Right) Spearman rank correlation for the automatic speech recognition task on the NASBench-ASR [14] dataset.

3 Results

We choose the most consistent NASM from all our evolutionary runs on the image classification task (`EZNAS-A`) and the automatic speech recognition task (`EZNAS-ASR`) and compare with existing NASMs from recent works. We can see that we are able to obtain competitive Spearman rank correlation coefficient on the NASBench-ASR automatic speech recognition task. Our `EZNAS-A` NASM is able to outperform competitive NASMs consistently and able to generalize across the architecture design spaces, as indicated by the test in Table 2 (Left).

4 Discussion

In this paper, we present EZNAS, a novel genetic programming driven approach to discover Zero Shot *Neural Architecture Scoring Metrics* (NASMs). EZNAS automates the discovery of relevant features of a neural network with minimal human intervention, and outperforms most existing human-designed NASMs in both generalizability and Kendall Tau Ranking Correlation. We believe that this is a first step towards not only efficient neural architecture search but also discovering interpretable programs that indicate the attributes of neural network that contribute to accuracy. In the rest of this section, we discuss the limitations of our current approach and outline future work that may enable us to discover better NASMs.

Program Design. To simplify the search problem, we take a mean (`aggregation_function` in Figure 1) of the scores generated across all `ReLU-Conv2D-BatchNorm2D` instances. This is a major limitation, as it hopes to capture layer connectivity patterns implicitly in the network attribute tensors (22 tensors). Extending our evolutionary search to take into account the global connectivity pattern of the neural networks or learning weighted averaging techniques for the individual layer scores along with the scoring program could help us discover better NASMs. Research in graph neural networks along with evolutionary program synthesis may allow us to obtain higher performance NASMs. Further, we impose structural restrictions on our program by necessitating a fixed structure (`ReLU-Conv2D-BatchNorm2D`). We have to truncate instances of `ReLU-Conv2D-Conv2D-BatchNorm2D` from the NDS space by ignoring the second convolution. As more diverse architectures are introduced in the field, we must reformulate collection of network statistics in a more robust manner. It is important to note that NASWOT measures the similarity in the binary codes generated from the ReLU units for two different inputs. Our implemented program design limits us to only using a single/batch of inputs and does not compare different input samples directly. Thus, we would not be able to discover the NASWOT NASM metric with our current formulation. Fortunately, integrating such functionality into the program design space is trivial and is an interesting line of future work to generate more complex metrics.

Network Statistics. Due to the computational resources required to generate network statistics, we have to pre-compute them and use the generated data as an *evolutionary task dataset*. We only use network statistics with input batch-size of 1 for evolution. A single sample statistic of a neural network is insufficient to describe the architecture. Unfortunately, this would cause a linear increase in memory requirement as well as increase in run-time of the evolutionary search. If we were to generate network statistics of all neural networks in the NASBench-201 and NDS spaces (excluding ImageNet), it would be 7 terabytes of tensors with a batch size of just 1. Computing network statistics at run-time can be made more efficient by using lower precision numerical formats, or exploring proxy tasks on smaller datasets (down-sampled image datasets, smaller neural network design architecture etc.) to discover metrics that can influence accuracy.

Evolutionary Search. Recent advances in the field of Neuro-Symbolic Program Synthesis [15] to learn mappings from input-output examples (neural network statistics to neural architecture scores) can motivate improvements in our evolutionary search. Further, exploring architecture connectivity encodings [16] for neural architecture search and discovering programs to rank neural architecture encodings may enable discovery of effective connectivity patterns to enable a deeper understanding of important features in neural architecture design beyond those contained in activations maps and weights.

References

[1] Akhauri, Y., A. Niranjan, J. P. Munoz, et al. Rhnas: Realizable hardware and neural architecture search, 2021.

[2] Abdelfattah, M. S., Łukasz Dudziak, T. Chau, et al. Best of both worlds: Automl codesign of a cnn and its hardware accelerator, 2020.

[3] Mellor, J., J. Turner, A. Storkey, et al. Neural architecture search without training, 2021.

[4] Abdelfattah, M. S., A. Mehrotra, Łukasz Dudziak, et al. Zero-cost proxies for lightweight nas, 2021.

[5] Chen, W., X. Gong, Z. Wang. Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective, 2021.

[6] Lin, M., P. Wang, Z. Sun, et al. Zen-nas: A zero-shot nas for high-performance deep image recognition, 2021.

[7] Hu, Y., Y. Liang, Z. Guo, et al. Angle-based search space shrinking for neural architecture search, 2020.

[8] Tanaka, H., D. Kunin, D. L. K. Yamins, et al. Pruning neural networks without any data by iteratively conserving synaptic flow, 2020.

[9] Balog, M., A. L. Gaunt, M. Brockschmidt, et al. Deepcoder: Learning to write programs, 2017.

[10] Real, E., C. Liang, D. R. So, et al. Automl-zero: Evolving machine learning algorithms from scratch, 2020.

[11] Dong, X., Y. Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search, 2020.

[12] Radosavovic, I., J. Johnson, S. Xie, et al. On network design spaces for visual recognition, 2019.

[13] Fortin, F.-A., F.-M. De Rainville, M.-A. Gardner, et al. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[14] Mehrotra, A., A. G. C. P. Ramos, S. Bhattacharya, et al. {NAS}-bench-{asr}: Reproducible neural architecture search for speech recognition. In *International Conference on Learning Representations (ICLR)*. 2021.

[15] Parisotto, E., A. Mohamed, R. Singh, et al. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations (ICLR 2017)*. 2017.

[16] White, C., W. Neiswanger, S. Nolen, et al. A study on encodings for neural architecture search, 2021.

Appendix
A Genetic Programming Approach To Zero-Shot Neural Architecture Ranking

Anonymous Author(s)
Affiliation
Address
email

1 Details of Evolutionary Framework

EZNAS uses evolutionary search to discover programs that can score neural network architectures in proportion to their accuracy. There are several components that are instrumental in facilitating efficient search for such a program in the prohibitively large program space. In this section, we go over the different components of our framework in detail.

1.1 Program Representation

In our evolutionary search, each individual program has to be evaluated on tens of gigabytes of *network statistics* with no approximations to generate precise NASM score-accuracy KTR fitness. To make such search computationally tractable, it is crucial to not introduce redundant operations in the program. Our initial attempt described in the Appendix 2.3 resembled that of AutoML-Zero [1] where NASMs were represented as a sequence of instructions and a memory space to store intermediate tensors. This led to discovery of NASMs with many redundant computations and an intractably large run-time. To reduce the complexity of search, we necessitate a *expression tree* structure on the NASM program to capture the executional ordering of the program. The program output appears at a root node, and the child (terminal) nodes are the *arguments* of the expression tree. These *arguments* are the *network statistics*. The advantage of this program representation is that there is only a single root node with dense connectivity from the root to the terminal nodes, leading to lesser redundancies.

1.2 Neural Network Statistics Generation

To search for genetic programs that can rank neural network architectures effectively, we need to generate a dataset of *'network statistics'*. As depicted in Figure 1, for any sampled neural network from the NDS or NASBench-201 spaces, we identify every `ReLU-Conv2D-BatchNorm2D` instance after randomly initializing the neural network. We register the input and output activation (`T1, T2`) and gradient (`T1G, T2G`) to the ReLU unit, weight and weight gradient of the Conv2D layer (`T3, T3G`) and the output and output gradient of BatchNorm2D (`T4, T4G`). We register these values for a single randomly sampled training data item (a single image $\mathcal{D}$), a random noise input ($\mathcal{N}$) with mean 0 and variance 1 and a training sample perturbed by random noise ($\mathcal{D} + \sqrt{0.01}\mathcal{N}$). In the Appendix 2.8, we also present an alternate formulation which identifies every `Conv2D-BatchNorm2D-ReLU` instance for network statistics generation to demonstrate that the evolutionary framework is not restricted to a `ReLU-Conv2D-BatchNorm2D` structure.

We register 24 different tensors for each `ReLU-Conv2D-BatchNorm2D` instance, however since the weight tensor is the same for the three types of input ($\mathcal{D}$, $\mathcal{N}$, $\mathcal{D} + \sqrt{0.01}\mathcal{N}$), we have 22 unique tensors. For evolutionary search, we generate an *evolution task dataset* by randomly sampling 80 neural networks from each available search space (NASBench-201 and NDS) and dataset (CIFAR-10,

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

CIFAR-100, ImageNet-16-120). At the time of evolution, we only have access to *evolution task dataset*. At the end of the search, we use a *test task dataset* to identify the best programs. The *test task dataset* has 1000 neural networks statistics for each dataset on the NASBench-201 space and 200 neural network statistics for each design space on NDS. We generate a smaller network statistics dataset on NDS due to RAM constraints. Since we are generating statistics for each layer on over 4640 ($= 1000 \times 3 + 200 \times 5 + 80 \times 8$) neural networks, the dataset size is approximately 400 GB for a batch size of 1.

1.3 Mathematical Operations

As an expression tree describes the execution order of the mathematical operations available to us, it is crucial to provide a varied set of operations to process the neural network statistics effectively. We provide 34 unique operations in our program search space. We include basic mathematical operations such as Addition, Product, Log as well as some advanced operations such as Cosine Similarity, KL Divergence, Hamming Distance. We provide the full list of mathematical operations at the end.

1.4 Program Application

Majority of the neural network architectures available in NASBench-201 and NDS have over 100 instances of `ReLU-Conv2D-BatchNorm2D`. This may mean that an expression tree would have as many as 2200 (22×100) possible arguments (terminal nodes), each of which can be used multiple times. This would result in a computationally intractable expression tree. To simplify the search problem, we generate a single expression tree with 22 possible inputs. This expression tree is then applied on every `ReLU-Conv2D-BatchNorm2D` instance and the output is aggregated across all instances using an `aggregation_function`. It is not necessary that the root node of an expression tree would give a scalar value, so we add a `to_scalar` operation above the root node of the expression tree. This serves as the 'score' of a single layer of the the sampled neural network architecture. We demonstrate how an expression tree is applied to a neural network architecture to generate a score in Figure 3. In `EZNAS-A`, the `aggregation_function` and `to_scalar` are both `mean`. In Appendix 2.8, we explore L2-Norm as a `to_scalar` function as well and find that we are able to discover effective NASMs.

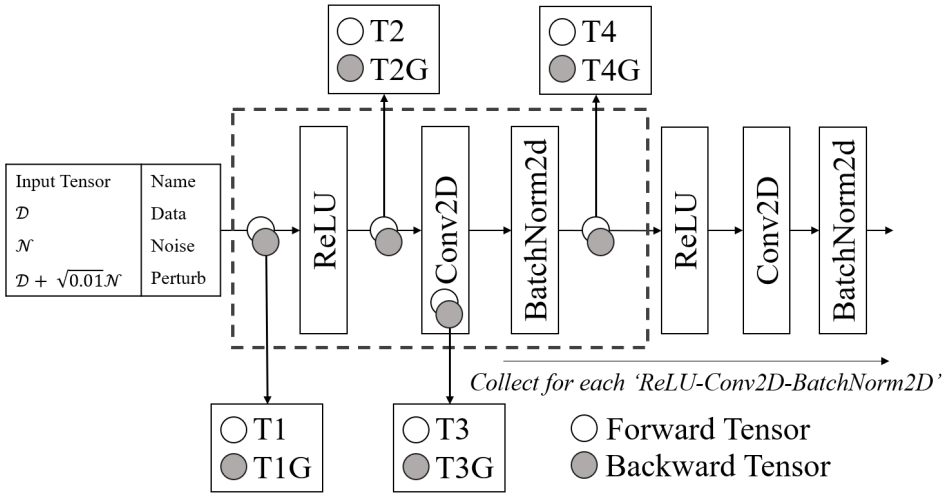


Figure 1: Collecting network statistics for the NASBench-201 and NDS search spaces. The $\mathcal{D}$ input tensor represents a single sample image from the dataset such as CIFAR-10, CIFAR-100 serving as input to the neural network. $\mathcal{N}$ represents a randomly initialized noise tensor. We refer to $\mathcal{D} + \sqrt{0.01}\mathcal{N}$ as 'Perturb'. It represents an input to the neural network which is a data-sample perturbed by noise.

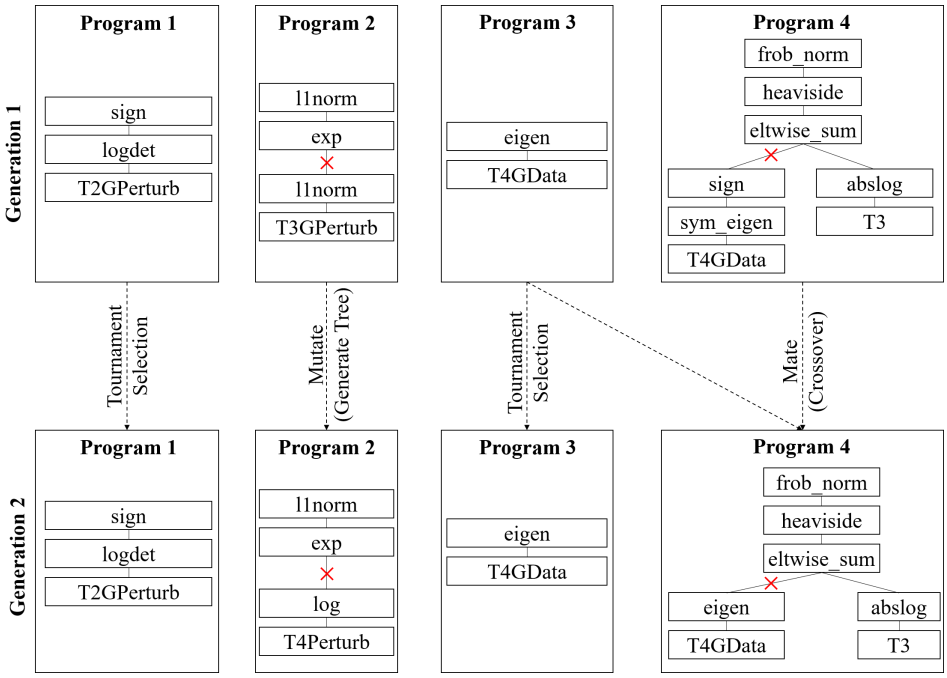


Figure 2: Variation of expression trees every generation. The x signs denote point of cross-over or mutation.

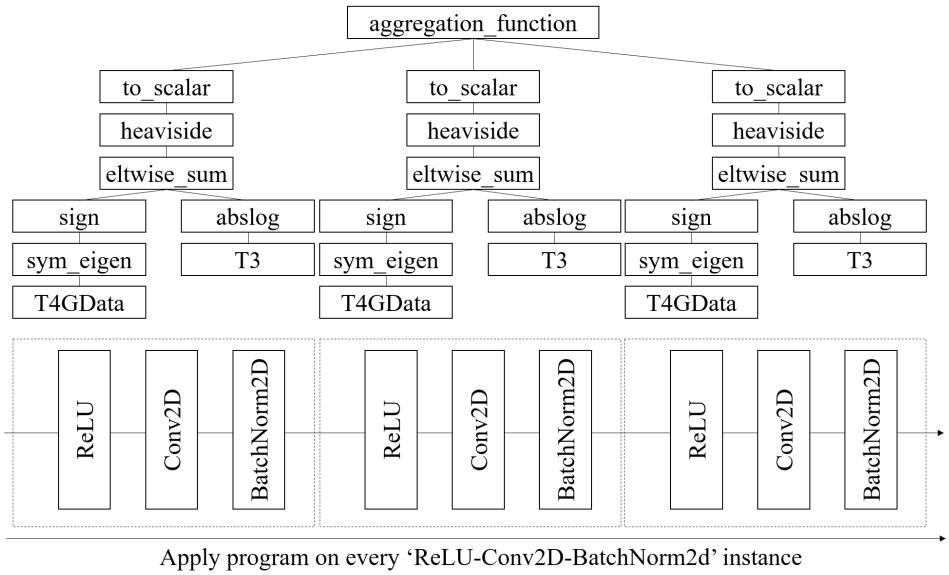


Figure 3: A program is evaluated by applying the NASM program followed by a `to_scalar` (such as L2-Norm, Mean) function on every `ReLU-Conv2D-BatchNorm2d` instance in the sampled neural network architecture. This is followed by an `aggregation_function` to give a final score to the sampled neural network architecture.

1.5 Evolutionary Algorithm

Our search algorithm discovers programs by modifying the expression tree representation. A fitness score is generated for every expression tree in the population. A selection and variation algorithm is then used to generate new offspring.

1.5.1 Population Initialization

We initialize a population of `n` programs. We do not impose any restrictions on the operations the nodes can use in the expression tree. Due to this, the number of valid expression trees several is orders of magnitude lesser than the total number of expression trees that can be generated with our mathematical operators and network statistics. To increase search efficiency, we would like to ensure we sample and evolve only valid expression trees. To enable this, we ensure that all individuals in the population are valid programs by testing program execution on a small sub-set of network statistics data. All programs that produce outputs with inf, nan or fail to execute are replaced by new randomly initialized valid programs.

1.5.2 Fitness Objective

As the goal of zero shot NAS is to be able to rank neural network architectures well, we utilize the Kendall Tau rank correlation coefficient as the fitness objective. The search objective is to maximize the Kendall Tau rank correlation coefficient between the scores generated by a program and the test accuracy of the neural networks.

1.5.3 Variation Algorithms

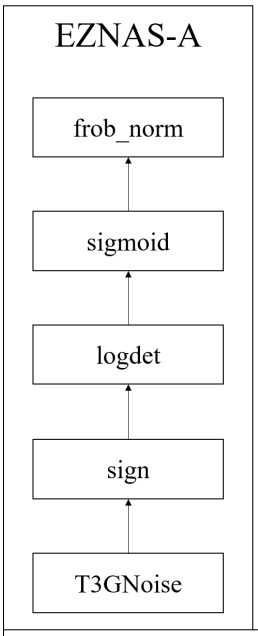
In our tests, we utilize the VarOr implementation from *Distributed Evolutionary Algorithms in Python* (DEAP) [2] framework for variation of individual programs. We generate `n` (hyper-parameter) offspring programs at each generation. `n/2` offspring are generated as a result of three operations; crossover, mutation or reproduction. These variations are depicted in Figure 2. For crossover, two individual programs are randomly selected from the population and mated. Our mating function randomly selects a crossover point from each individual and exchanges the sub-trees with the selected point as root between each individual. The first child is appended to the offspring population. For mutation, we randomly select a point in the individual program, and replace the sub-tree at that point by a randomly generated expression tree. We repeat the variation algorithm on the population till `n/2` valid individuals are generated. To encourage diversity, we also randomly generate `n/2` valid individuals. We have placed static limits on the height of all expression trees at 10.

1.6 Program Evaluation Methodology

At each generation, the fitness of the entire population is invalidated and recalculated. Calculating the fitness of each program on the entire dataset (which can be approximately 1 TB) is computationally infeasible. Further, we may want to find generalized programs that give high fitness on many different architecture design spaces and datasets. This means we have to evaluate each individual on 7 TB of data. Reducing the computation by evaluating the fitness of the population on a single small fixed sub-set of neural networks from the search space causes discovered programs to trivially over-fit to the sub-set statistics in our tests. To address over-fitting of programs to small datasets of network statistics while minimizing compute resources required for evaluating on the entire dataset of network statistics, we use our *evolution task dataset*. The evolution task dataset contains statistics of 80 neural networks on each search space. We evaluate individuals by randomly choosing `s` search spaces, and sampling 20 neural networks from each of the chosen search spaces. We take the minimum fitness achieved by the individual program on the `s` spaces. We consider `s` as a hyper-parameter. In our tests, this is consistently kept at 4.

1.7 Program Testing Methodology

At the end of the evolutionary search, our primary goal is to test whether the programs discovered are able to provide high fitness on previously unseen neural network architecture statistics. We test the fittest program from our final population as well as the two fittest programs encountered through-out the evolutionary search. At test time, we take the program and find the score-accuracy KTR over the entire *test task dataset*.




```
def EZNAS_A(ic,oc,k):
    T3GNoise = torch.randn(ic, oc, k, k)
    tens = torch.sign(T3GNoise)
    tens = torch.logdet(tens)
    tens[tens!=tens] = 0
    tens = torch.sigmoid(tens)
    tens = torch.norm(tens, p='fro')
# Testing
c_var = [4, 8, 16, 32, 64, 128]
k_var = [1, 3, 5, 7]
# E -> repeat 5000 times and return average
# Vary output channels
E([NASM(ic=8, oc=x, k=3).item() for x in c_var])
>>> [2.29,  3.26,  4.60,  6.50,  9.25, 13.06]
# Vary input channels
E([NASM(ic=x, oc=8, k=3).item() for x in c_var])
>>> [2.30,  3.24,  4.61,  6.55,  9.23, 13.07]
# Vary kernel size
E([NASM(ic=8, oc=8, k=x).item() for x in k_var])
>>> [4.00 , 3.25, 3.53, 4.04]
```

Score monotonically increasing non-linearly with channel count.
Kernel size of 1 and 7 gives higher score than 3 or 5

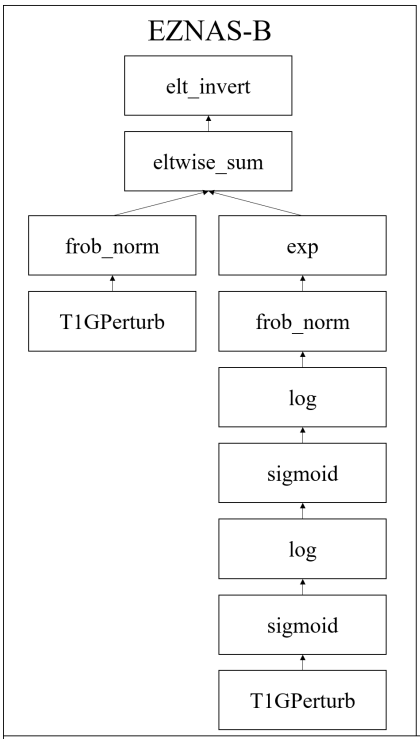
Figure 4: Analysis of our best program (`EZNAS-A`) on the Image Classification task. `to_scalar` not required as the output is scalar already.

2 Examination Of Our Best Programs

We analyze two NASMs discovered by EZNAS to give a deeper understanding of the nature of programs our search finds. The first program we analyze is `EZNAS-A`. This evolution task dataset for discovering this program was NDS DARTS. The input to the NASM in Figure 4 is the `T3GNoise` (Weight Gradient with Random Noise Input). To understand how the NASM score varies with the weight gradient size, we generate random tensors of varying sizes and average the NASM output 5000 times. We find that the score increases as the number of channels or depth increases, we also observe that kernel sizes of 1 and 7 give higher scores than 3 and 5 with the lowest score being assigned to kernel of size 3. It is interesting to see that the expectation value of `EZNAS-A` NASM in Figure 4 translates to a weighted form of parameter counting, with a non-linear monotonically increasing scaling of score with the number of input/output channels and a locally parabolic relationship between the score and the kernel size with the minimum score (among integers) at kernel size of 3.

The second program we analyze (`EZNAS-B`) is depicted in Figure 5. This evolution task dataset for discovering this program was NDS ENAS and its score-accuracy KTR on the *test task dataset* is provided in Figure 13. We analyze this NASM in a manner similar to Figure 4. We provide random tensors in place of `T1GPerturb` (Difference in pre-activation gradients for a random noisy input and a data-sample input). This is an approximation to understand how the NASM responds to change in activation gradient size. We find that the activation map size is exponentially more influential to the score when compared to the number of channels.

With the random initialization approximation done in our analysis, we find that our NASM `EZNAS-A` is counting the number of parameters in a 'weighted fashion'. `EZNAS-B` generates a score by giving higher weightage to feature map size over number of channels as a proxy for accuracy. It is interesting to note that when compared to the methods from recent papers in zero shot NASMs, this form of weighted parameter counting works more consistently and better. This is further supported by our finding that FLOPs and parameter count are competitive NASMs which work more consistently as opposed to `synflow` or `grad_norm` which work on the NASBench-201 space but fail on the NDS space.




```
def EZNAS_B(c, k):
    T1GPerturb = torch.randn(1, c, k, k)
    tens = torch.sigmoid(T1GPerturb)
    tens[tens<=0] = 1
    tens = torch.log(tens)
    tens = torch.sigmoid(tens)
    tens[tens<=0] = 1
    tens = torch.log(tens)
    tens = torch.norm(tens, p='fro')
    tens = torch.exp(tens)
    tens += torch.norm(pagp, p='fro')
    tens = to_scalar(1/tens)
# Testing
c_var = [4, 8, 16, 32, 64, 128]
k_var = [4, 8, 16, 32]
# E -> repeat 5000 times and return average
# Vary channels
E([NASM(c=x, k=4).item() for x in c_var])
>>> [4.6e-5, 7.1e-7, 1.98e-9,
     4.4e-13, 3.2e-18, 1.7e-25]
# Vary activation size
E([NASM(c=2, k=x).item() for x in k_var])
>>> [4.7e-5, 1.9e-9, 3.2e-18, 9.3e-36]
```

Activation map size is exponentially more important than the number of channels.

Figure 5: Analysis of our second best program (`EZNAS-B`) on the Image Classification task.

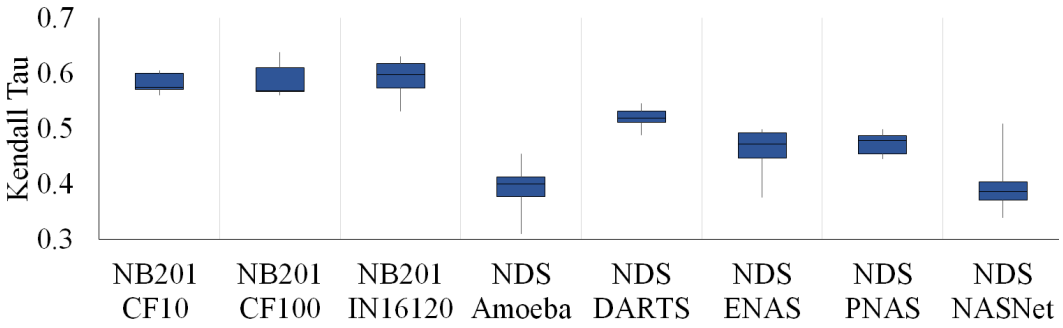


Figure 6: Effect of seed on our score-accuracy KTR of `EZNAS-A` with a batch size of 1. CIFAR and ImageNet abbreviated as CF and IN respectively. This test was done on each design space over 7 seeds for 400 Neural Networks.

2.1 NASBench-201 and NDS

For image classification, we utilize the NASBench-201 [3] and NDS [4] NAS search spaces for our evolutionary search as well as testing. NASBench-201 consists of 15,625 neural networks trained on the CIFAR-10, CIFAR-100 and ImageNet-16-120 datasets. Neural Networks in Network Design Spaces (NDS) uses the DARTS [5] skeleton. The networks are comprised of cells sampled from each of AmoebaNet [6], DARTS [5], ENAS [7], NASNet [8] and PNAS [9]. There exists approximately 5000 neural network architectures

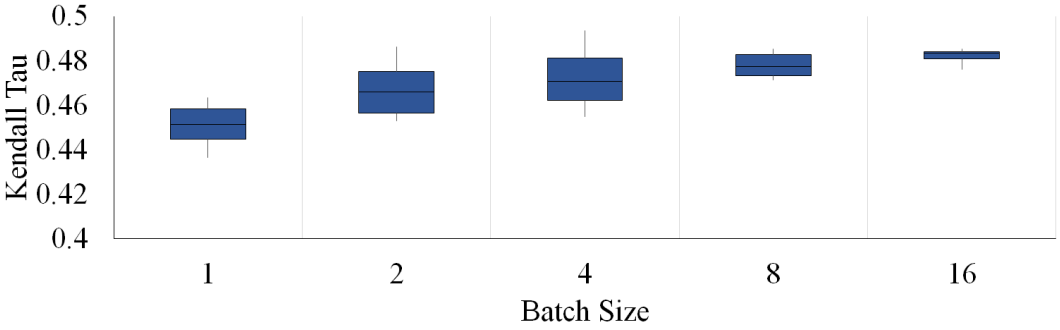


Figure 7: Effect on score-accuracy KTR of `EZNAS-A` with respect to batch size. This test was done on the NDS PNAS design space over 7 seeds for 400 Neural Networks.

2.2 Our evolutionary algorithm

We explicitly describe our evolutionary search method in Algorithm 1. `evolution_space` refers to the evolution task dataset. the `evaluate` function evaluates every individual in the offspring and assigns a fitness value to them. This fitness value is the minimum score-accuracy KTR obtained by testing it on each of the `evolution_space`. `test_space` refers to the test task dataset, and is used after the last generation to identify good programs. The `genValidPopulation` function generates `N` valid individuals. The `VarOr` function is the variation algorithm we have used to mate, mutate and cross-over the population. We repeat the VarOr function generates half the offsprings, while the other half is randomly initialized at each generation. This is to promote diversity in the population. The `selectValid` function selects all the valid individuals produced by the VarOr function by evaluating each individual on a single neural network statistic. The `selTop3` selects the three individuals with the highest fitness. The top 3 individuals are propagated to the next generation with no variation. `selTournament` selects the best individual among `x` randomly chosen individuals, `k` times. We set the `x=4` and `k=N`.

2.3 Details of Sequential Program Representation

Our initial attempts at discovering NASMs took a different approach to program representation. This *sequential program representation* posed no structural limitations on the program. We have 22 static memory addresses, which contained network statistics and are referenced with integers `0-21`. To store intermediate tensors generated by the program, we allocate 80 dynamic memory addresses, which can be over-written in the program as well. To store intermediate scalars generated by the program, we allocate 20 memory addresses. As seen in Figure 8, we represent the programs as integers, where each instruction is expressed as 4 integers. The first integer provides the write address, the second integer provides the operation ID and the third and fourth integers provide the read addresses for the operation. We initialize *valid* random integer arrays and convert them to programs to evaluate and fetch the fitness (score-accuracy KTR). We allow `Mate, InsertOP, RemoveOP, MutateOP & MutateOpArg` as variation functions. The `Mate` function takes two individuals, and takes the first half of each individual. Then, these components are interpolated to generate a new individual. The `InsertOP` function inserts an operation at a random point in the program. The `RemoveOP` function removes an operation at a random point in the program. The `MutateOP` changes a random operation in program without changing read/write addresses. The `MutateOpArg` function simply replaces one of the read arguments of any random instruction with another argument from the same address space (dynamic address argument cannot be replaced by a scalar address argument).

While we are able to discover weak NASMs with this formulation, we observe that there are too many redundancies in the programs discovered. Program length bloating as well as operations that do not contribute to the final output were frequently observed. Due to these issues, the evolution time evaluation of individual fitness quickly became an intractable problem. To address this, we change our program representation to a expression tree representation in the results reported in the paper. This representation necessitates contribution of each operation to the final output, which means there is no redundant compute. While the sequential program representation is valid, we believe

Algorithm 1 EZNAS Search Algorithm

```
evolution_space = ['DARTS', 'NB201-CIFAR10']
test_space = ['PNAS', 'ENAS']
population = genValidPopulation(N)
evol_dataset = loadDataset(evolution_space)
evaluate(population)
top_gen = []
for gen=1:T do
    offspring = []
    if len(top_gen)>0 then
        offspring.append(top_gen)
    top_gen = []
    while len(offspring) < N//2 do
        # Applies crossover, mutation & reproduction
        individuals = VarOr(population)
        offspring.append(selectValid(individuals))
    # Half the population is randomly
    # initialized at each generation
    individuals = genValidPopulation(N//2)
    offspring.append(individuals)
    population = evaluate(offspring)
    top_gen.append(selTop3(population))
    if gen!=T then
        population = selTournament(population)
test(population)
top_NASMs = selTop3(population)
```

that significant engineering efforts are required to ensure discovery of meaningful programs. Our sequential program representation is directly inspired by the formulation used in AutoML-Zero [1]. AutoML-Zero makes significant approximations in the learning task to evolutionarily discover MLPs. While AutoML-Zero has a much larger program space to search for, approximations in computing individual fitness are not feasible in our formulation as generating exact score-accuracy KTR is an important factor in selecting individuals with high fitness.

2.4 Noise and Perturbation for Network Statistics

To generate network statistics, we use three types of input data. The first is simply a single random sample from the dataset (e.g. a single image or a batch of images from CIFAR-10). To generate a noisy input, we simply use the default `torch.randn` function as `input = torch.randn(data_sample.shape)`. The third type of input we provide is a data-sample which has been perturbed by random noise (`input = data_sample + 0.01**0.5*torch.randn(data_sample.shape)`).

2.5 Network Initialization Seed Test

In Figure 9, we use different seeds to change the initialization and input tensors, but keep the neural architectures being sampled fixed in the respective spaces. The variance in the score accuracy KTR is much lesser than in Figure 10 where the seed also controls the neural architectures being sampled. This shows the true variation in our `EZNAS-A` NASM with respect to network initialization.

2.6 Hardware used for evolution and testing

Our evolutionary algorithm runs on Intel(R) Xeon(R) Gold 6242 CPU with 630GB of RAM. Our RAM utilization for evolving programs on a single Image Classification dataset was approximately 60GB. RAM utilization can vastly vary (linearly) based on the number of neural network statistics that are being used for the evolutionary search. Our testing to generate the statistics for the seed

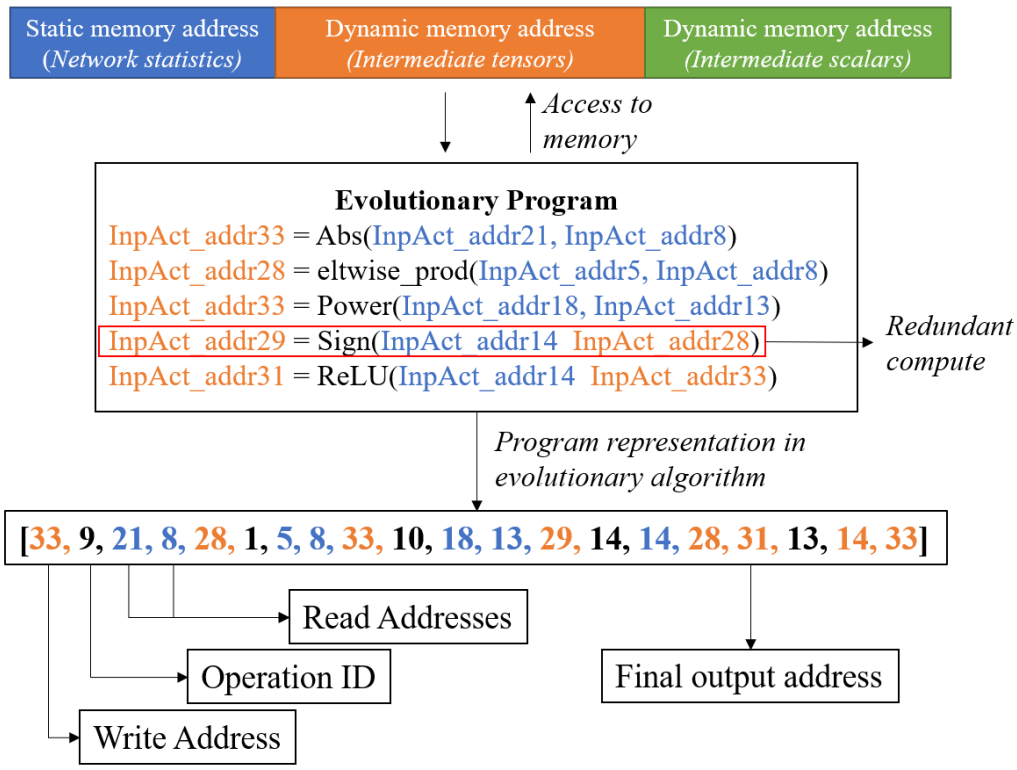


Figure 8: Our sequential program representation.

experiments as well as the final Spearman ρ and Kendall Tau Rank Correlation Coefficient is done on an NVIDIA DGX-2 server with 4 NVIDIA V-100 32GB GPUs.

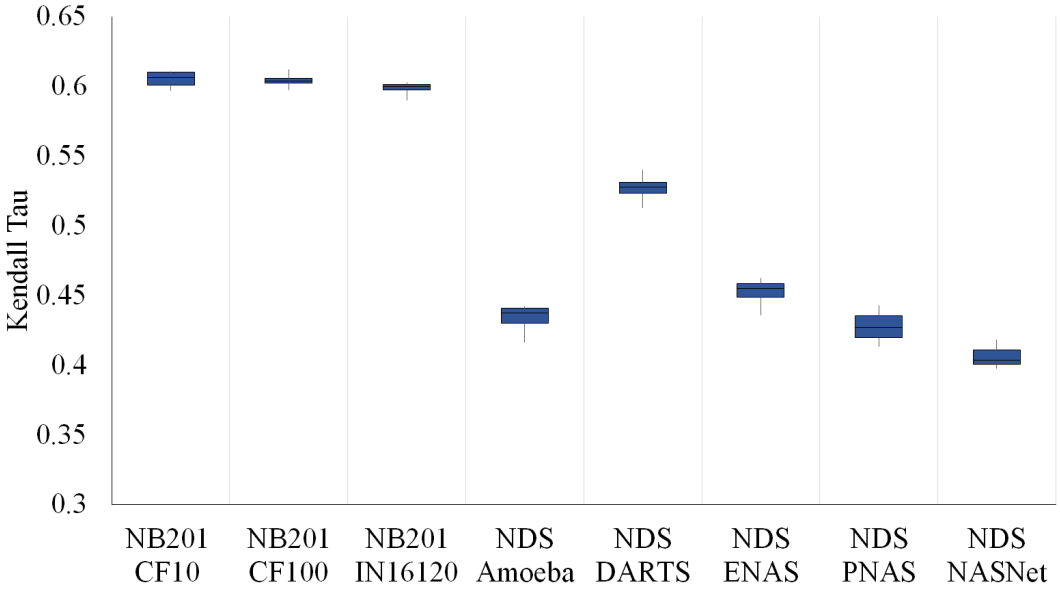


Figure 9: Experiment demonstrating the effect of seed on our score-accuracy KTR for neural network initialization with a batch size of 1.

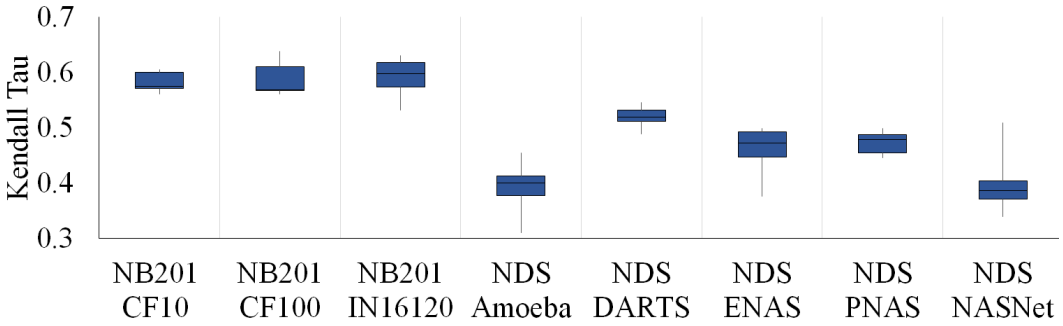


Figure 10: Effect of seed on our score-accuracy KTR with a batch size of 1. CIFAR and ImageNet abbreviated as CF and IN respectively. This test was done on each design space over 7 seeds for 400 Neural Networks.

2.7 Hyper-parameters for discovering `EZNAS-A`

```
num_generation:  15
population_size:  50
tournament_size:  4
MU: 25
lambda_ :  50
crossover_prob:  0.4
mutation_prob:  0.4
min_tree_depth:  2
max_tree_depth:  6
```

2.8 Varying network statistics and `to_scalar`

In Figure 11, we detail 3 tests while evolving on the NDS_DARTS CIFAR-10 search space in an identical fashion to `EZNAS-A` (referred to as `EZNAS-(R-C-B)-(Mean)` here). `EZNAS-(C-B-R)-(Mean)`

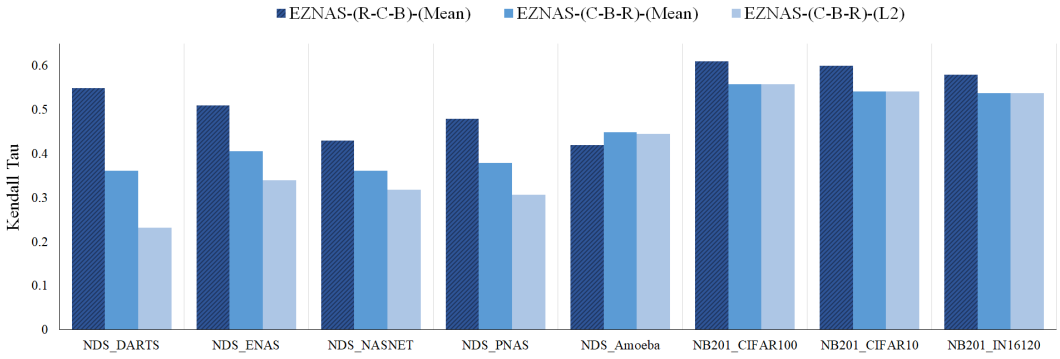


Figure 11: Experiment demonstrating the ability to discover NASMs with an alternate network statistics collection strategy and `to_scalar` function. Experiments are named as EZNAS-(Statistics Collection Structure)-(`to_scalar`). Two statistics collection structures are tested. (R-C-B) is a `ReLU-Conv2D-BatchNorm2D` structure, (C-B-R) is a `Conv2D-BatchNorm2D-ReLU` structure. (`to_scalar`) can be Mean or L2.

and `EZNAS-(C-B-R)-(L2)` correspond to alternate `to_scalar` and network statistics collection tests respectively. We demonstrate that while `EZNAS-(R-C-B)-(Mean)` is superior, we are able to discover NASMs with all three formulations.

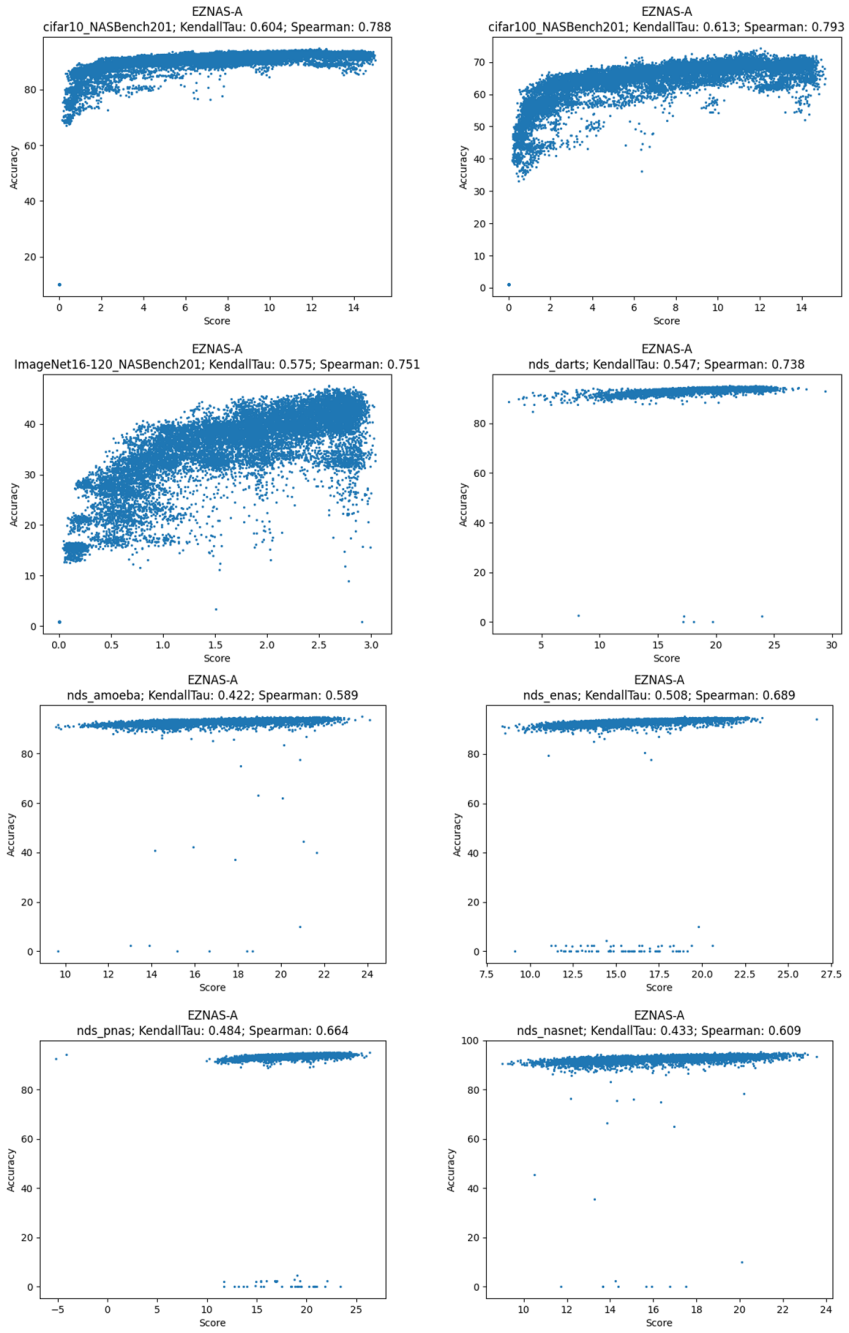


Figure 12: Scatter plot between `EZNAS-A` NASM score and accuracy on NASBench-201 and NDS neural architecture search spaces.

References

[1] Real, E., C. Liang, D. R. So, et al. Automl-zero: Evolving machine learning algorithms from scratch, 2020.

[2] Fortin, F.-A., F.-M. De Rainville, M.-A. Gardner, et al. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[3] Dong, X., Y. Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search, 2020.

Kendall Tau	NASBench201 CIFAR10	NASBench201 CIFAR100	NASBench201 ImageNet16-120	Amoeba CIFAR10	DARTS CIFAR10	ENAS CIFAR10	PNAS CIFAR10	NASNet CIFAR10	FLOPs	Params
NASBench201 CIFAR10	0.533	-0.499	0.4345	0.4422	**0.5777**	0.472	0.4514	0.2633	*0.5610*	0.5610
NASBench201 CIFAR100	0.5242	-0.496	0.4522	0.4152	**0.5627**	0.476	0.453	0.2531	*0.5472*	0.5472
NASBench201 ImageNet16-120	0.4575	-0.424	0.4350	0.4213	**0.5733**	0.456	0.2903	0.2338	*0.5036*	0.5036
Amoeba CIFAR10	0.1146	-0.029	0.3263	0.352	*0.3774*	0.380	0.35278	**0.4042**	0.2614	0.2658
DARTS CIFAR10	0.3236	-0.173	0.2088	0.4187	*0.5077*	0.460	0.47570	0.1439	**0.5079**	0.5042
ENAS CIFAR10	0.2833	-0.143	0.3132	0.4292	0.3795	0.422	0.41701	0.3400	**0.4739**	*0.4704*
PNAS CIFAR10	0.2208	-0.053	0.4212	0.4466	0.4435	**0.514**	*0.4874*	0.3799	0.3363	0.3223
NASNet CIFAR10	0.2370	0.0229	0.2880	*0.3789*	**0.4381**	0.364	0.3594	0.3757	0.1996	0.2102

Figure 13: Full correlation table. Each column represents the dataset evolution was performed on. The DARTS-CIFAR10 column is the `EZNAS-A` NASM. Each row represents the dataset the best discovered NASM program was tested on. Best score-accuracy KTR in bold and underlined. Second best score-accuracy KTR in italics and underlined. These tests are done by evolving on 100 neural networks and testing on the test task dataset (1000 randomly sampled neural networks on NASBench-201 and 200 randomly sampled neural networks on NDS). The network statistics were generated with a batch size of 1.

[4] Radosavovic, I., J. Johnson, S. Xie, et al. On network design spaces for visual recognition, 2019.

[5] Liu, H., K. Simonyan, Y. Yang. Darts: Differentiable architecture search, 2019.

[6] Real, E., A. Aggarwal, Y. Huang, et al. Regularized evolution for image classifier architecture search, 2019.

[7] Pham, H., M. Y. Guan, B. Zoph, et al. Efficient neural architecture search via parameter sharing, 2018.

[8] Zoph, B., V. Vasudevan, J. Shlens, et al. Learning transferable architectures for scalable image recognition, 2018.

[9] Liu, C., B. Zoph, M. Neumann, et al. Progressive neural architecture search, 2018.

Op ID	**Operation**	**Description** Output: `C`, Input: `A, B`
OP0	Element-wise Sum	`C = A+B`
OP1	Element-wise Difference	`C = A-B`
OP2	Element-wise Product	`C = A*B`
OP3	Matrix Multiplication	`C = A@B`
OP4	Lesser Than	`C = (A<B).bool()`
OP5	Greater Than	`C = (A>B).bool()`
OP6	Equal To	`C = (A==B).bool()`
OP7	Log	`A[A<=0] = 1` `C = torch.log(A)`
OP8	AbsLog	`A[A==0] = 1` `C = torch.log(torch.abs(A))`
OP9	Abs	`C = torch.abs(A)`
OP10	Power	`C = torch.pow(A, 2)`
OP11	Exp	`C = torch.exp(A)`
OP12	Normalize	`C = (A - A`$_{mean}$`)/A`$_{std}$ `C[C!=C] = 0`
OP13	ReLU	`C = torch.functional.F.relu(A)`
OP14	Sign	`C = torch.sign(A)`
OP15	Heaviside	`C = torch.heaviside(A, values=[0])`
OP16	Element-wise Invert	`C = 1/A`
OP17	Frobenius Norm	`C = torch.norm(A, p='fro')`
OP18	Determinant	`C = torch.det(A)`
OP19	LogDeterminant	`C = torch.logdet(A)` `C[C!=C]=0`

OP20	SymEigRatio	`A = A.reshape(A.shape[0], -1)`
		`A = A@A.T`
		`A = A + A.T`
		`e,v = torch.symeig(A, eigenvectors=False)`
		`C = e[-1]/e[0]`

OP21	EigRatio	`A = A.reshape(A.shape[0], -1)`
		`A = torch.einsum('nc,mc->nm', [A,A])`
		`e,v = torch.eig(A)`
		`C = (e[-1]/e[0])[0]`

OP22	Normalized Sum	`C = torch.sum(A)/A.numel()`
OP23	L1 Norm	`torch.sum(abs(A))/A.numel()`

OP24	Hamming Distance	`A = Heaviside(A)`
		`B = Heaviside(B)`
		`C = sum(A!=B)`

OP25	KL Divergence	`C = torch.nn.KLDivLoss('batchmean')(A,B)`

OP26	Cosine Similarity	`A = A.reshape(A.shape[0], -1)`
		`B = B.reshape(B.shape[0], -1)`
		`C = torch.nn.CosineSimilarity()(A, B)`
		`C = torch.sum(C)`

OP27	Softmax	`C = torch.functional.F.softmax(A)`
OP28	Sigmoid	`C = torch.functional.F.sigmoid(A)`
OP29	Ones Like	`C = torch.ones_like(A)`
OP30	Zeros Like	`C = torch.zeros_like(A)`
OP31	Greater Than Zero	`C = A>0`
OP32	Less Than Zero	`C = A<0`
OP33	Number Of Elements	`C = torch.Tensor([A.numel()])`

Table 1: List of operations available for the genetic program.