IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 9 February 2023, accepted 27 February 2023, date of publication 6 March 2023, date of current version 9 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3252887

RESEARCH ARTICLE

Novel Surrogate Measures Based on a Similarity Network for Neural Architecture Search

ZEKI KUŞ[1], **CAN AKKAN**[2], **AND AYLA GÜLCÜ**[3]
[1]Department of Computer Engineering, Fatih Sultan Mehmet University, Beyoğlu, 34445 Istanbul, Turkey
[2]Sabancı Business School, Sabancı University, Tuzla, 34956 Istanbul, Turkey
[3]Department of Software Engineering, Bahçeşehir University, Beşiktaş, 34353 Istanbul, Turkey
Corresponding author: Ayla Gülcü (ayla.gulcu@eng.bau.edu.tr)

ABSTRACT We propose two novel surrogate measures to predict the validation accuracy of the classification produced by a given neural architecture, thus eliminating the need to train it, in order to speed up neural architecture search (NAS). The surrogate measures are based on a solution similarity network, where distance between solutions is measured using the binary encoding of some graph sub-components of the neural architectures. These surrogate measures are implemented within local search and differential evolution algorithms and tested on NAS-Bench-101 and NAS-Bench-301 datasets. The results show that the performance of the similarity-network-based predictors, as measured by correlation between predicted and true accuracy values, are comparable to the state-of-the-art predictors in the literature, however they are significantly faster in achieving these high correlation values for NAS-Bench-101. Furthermore, in some cases, the use of these predictors significantly improves the search performance of the equivalent algorithm (differential evolution or local search) that does not use the predictor.

INDEX TERMS Neural architecture search, surrogate model, similarity-based prediction, evolutionary algorithm.

I. INTRODUCTION

In computer vision, the success of a deep neural network is highly dependent on its architecture and hyper-parameters. However, the process of selecting the best performing architecture along with its hyper-parameters manually requires intense engineering efforts with high computational costs. *Neural Architecture Search (NAS)* is a subfield of AutoML [10] which aims to automate this tedious architecture design process. It has also significant overlap with another AutoML subfield, *Hyper-parameter Optimization (HPO)* which aims to optimize the training-related parameters such as the learning rate and the batch size. Unlike HPO, NAS focuses on optimizing model-related parameters such as the number of layers, number of units and connection types among those units (see [9] for a recent survey on AutoML methods).

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Wei.

Elsken et al. [7] categorize NAS methods according to the adopted search space, selected search strategy and performance prediction strategy. The search strategy selects an architecture from a predefined search space, and the performance prediction strategy returns the predicted classification performance of the selected architecture back to the search strategy. Elsken et al. [7] mention five main groups of search strategies as Bayesian Optimization (BO) methods, Evolutionary Algorithms (EA), Reinforcement Learning (RL) approaches, Gradient methods and Random Search (RS) methods. A similar grouping is also provided in [2]. The use of EAs to evolve the neural architectures, also known as *neuro-evolution*, dates back to decades ago (see [37]). In the earlier studies, EAs were used to optimize both the neural architecture and its weights, but were found to be poor for contemporary architectures with millions of weights. Therefore, gradient-based methods are used for optimizing the weights and EAa are used solely for optimizing the neural architecture in recent studies. Recent studies have shown that Regularized Evolution (RE) and Differential Evolution (DE) are the two most successful neuro-evolutionary approaches for NAS [1], [23], [38].

NAS can require prohibitively large computational resources for performance prediction. For example, RL approach of Zoph et al. [39] resulted in a state of the art performance in the image classification domain, however, searching for an architecture for CIFAR-10 dataset [14] required 800 GPUs for 28 days. A strategy to reduce the search space that has found wide acceptance in the literature is developed by Zoph et al. [40], where the search is done over neural building blocks instead of over whole architectures. In their proposed *NASNet* search space, the building blocks, called *cells*, represent a part of a fixed architecture. Using a cell structure as the building block of the final architecture is also adopted in the *DARTS* search space of [17].

Even with the reduced search space of cells, NAS is still quite challenging, hence the design of a fast and effective prediction strategy is essential for the success of a NAS algorithm. Within a limited computational budget predictor-based NAS methods train a surrogate model which is then used to predict the final accuracy or ranking of architectures, without fully training the architectures. Thus, they should take less time to evaluate any given architecture, and the predicted accuracy values should have high correlations with the actual accuracy values in the current search space.

In this study, we present two surrogate models, WA (weighted-average) and RG (regression-based) for NAS, which both utilize the information in the solution similarity network. Two different algorithms, Local Search (LS) and Differential Evolution (DE), are selected as the search algorithms, and each of those algorithms are coupled with each of the surrogate models as their performance predictors. New solutions generated during the search are added to the similarity network as new nodes, where similar solutions are connected with edges. In the solution network, some solutions have their true accuracy values queried from the corresponding NAS-Bench dataset, whereas the accuracy of the others are predicted by the surrogate model, WA or RG, using their neighbors in the solution network. The construction of the solution network and deciding whether to predict or to evaluate a given solution's accuracy are handled within the search algorithm.

Our contributions are as follows:

- For fast NAS, we propose two novel surrogate models based on a solution similarity network. Each of the surrogate models, Weighted Average-based and Regression-based prediction methods, have been implemented within Local Search (LS) and DE algorithms and tested on the NAS-Bench-101 and NAS-Bench-301 datasets by adhering to the NAS best practices.
- We introduce a new binary encoding for NAS-Bench-101 and NAS-Bench-301, which is an extension to the path encoding. The encoding is used to measure the distance between solutions in the proposed solution similarity network, and it uses a richer set of graph sub-components of the neural architectures in order to distinguish them better than path encoding.
- We demonstrate that a relatively simple similarity-based solution network can form the foundation of effective prediction tools for NAS. Our computational experiments reveal that the prediction methods can improve the search performance significantly, as demonstrated for both NAS-Bench-101 and NAS-Bench-301 when used within DE. However, their improvement in LS is not as significant. When it comes to the predictive power of the surrogate models, our correlation results show that these prediction models are fast and start giving high quality predictions quickly. For example, DE with weighted averages as the predictor yields excellent correlation results for NAS-Bench-101 where most powerful predictors fail.

The rest of the paper is organized as follows. In the next section we provide a brief overview of the NAS literature. In Section III we describe the two benchmark datasets we have used. In Section IV, first the distance measure and the solution encoding on which it is based on are discussed. Then, the two surrogate models are presented. In Section V, LS and the DE algorithms are discussed, with an emphasis on how the surrogate models are integrated into the search process. The results of the computational experiments are presented in Section VI, which are followed by our concluding remarks in Section VII.

II. A BRIEF REVIEW OF THE RELATED NAS LITERATURE

Recently, the introduction of several benchmark datasets has increased research on the development of predictor-based NAS algorithms, which use efficient strategies to predict the performance of partially-trained or even untrained neural network architectures in order to find high-quality architectures within a limited computational budget. Differentiable-based [18], RL [39] and EAs [1], [23] have been applied successfully for NAS. Real et al. [23] showed, for the first time, that architectures generated by EAs surpass hand-engineered architectures. They use a modified version of RE in the *NASNet* search space. They state that the best architecture obtained by this method yields comparable accuracy with respect to the baseline architecture, *NASNet-A* [40], with considerably smaller number of parameters. Later, it is shown by Awad et al. [1] that DE was able to perform better than RE on NAS-Bench-101 benchmark. Siems et al. [26] used 10 different search algorithms to sample architectures from a large search space to create the first surrogate benchmark, NAS-Bench-301. They selected only two EAs, DE and RE, as the meta-heuristics that yield a good coverage of this large search space.

Most of the recent NAS algorithms use performance prediction strategies that focus on training a model to predict the final validation accuracy of an architecture. Neural networks, Gaussian processes and tree-based methods are among the most widely-used models as NAS predictors. White et al. [33]

propose a Bayesian optimization (BO) and neural predictor framework based algorithm, *BANANAS*, that achieves competitive results on DARTS and NAS-Bench-101 benchmarks. It outperforms RE of Real et al. [23] which was reported previously in [38] as the state-of-the-art on NAS-Bench-101.

Wen et al. [31] propose a two stage predictor which is a cascade of a classifier and a regressor both of which are based on Graph Convolutional Networks. The results show that their approach outperforms RE on NAS-Bench-101. Wei et al. [30] propose a predictor guided EA for NAS called *NPENAS*. Two different predictors, a Bayesian optimization (BO) based predictor and a graph-based neural network predictor, are utilized. The results state that NPENAS with BO predictor provides slightly better performance than BANANAS [33] on NAS-Bench-101 and DARTS search space and achieves a test error close to the performance of the predictor in [31] on NAS-Bench-201. In [36], a few weak predictors are used to fit small local spaces and to progressively shrink the search space towards a subspace where good architectures reside. The proposed method, *WeakNAS*, in which multilayer perceptron, regression tree and random forest models are used as weak predictors, is said to be very cost efficient and able to find some of the best performing architectures on both NAS-Bench-101 and NAS-Bench-201 with fewer database queries than RE in [23].

White et al. [35] present a large-scale comparative study of performance predictors used in NAS algorithms. 31 predictors that fall into four different categories, namely, (i) model-based methods, (ii) learning curve-based methods, (iii) zero-cost methods, and (iv) one-shot (weight sharing) methods, are compared across a variety of initialization time and query time budgets by measuring the Pearson correlation and rank correlation metrics. Based on the experimental results, they provide recommendations for the best predictors to use under different initialization and query-time budgets. Moreover, they state that using a more complex predictor that combines complementary information from three families of predictors leads to significant performance improvement. In another study, Ru et al. [24] propose to predict the final test performance based on training speed estimation.

III. BENCHMARK DATASETS

In this section, we provide brief descriptions of the two benchmark datasets, NAS-Bench-101 and NAS-Bench-301, that have been used in this study. There are also other benchmarks in the literature such as NAS-Bench-201 [6] and NAS-HPO-Bench [13], but they are not included in this study due to their limited search space sizes.

NAS-Bench-101 is the first large scale tabular NAS architecture dataset identifying 423k unique convolutional architectures [38]. The search space is restricted to the space of small feed forward structures, called cells. Each cell is stacked 3 times followed by a downsampling layer, and this pattern is repeated 3 times followed by global average pooling and a final dense softmax layer in order to create a whole architecture. The cell search space consists of all possible directed acyclic graphs (DAGs) with at most 7 nodes, where each node can represent any of the following three operations: 3×3 max-pool, 1×1 convolution and 3×3 convolution. There are also two special nodes IN and OUT, representing the input and output tensors, respectively. The number of edges is also restricted to 9. To build the dataset, all valid architectures within the given search space have been trained and evaluated on CIFAR-10, three times with four increasing number of epochs: 4, 12, 36, 108. All the metrics regarding this training process, e.g. the training/validation/test accuracy, the training time in seconds and the number of trainable model parameters, are provided. The researchers just perform a query in this dataset in order to learn the performance of a proposed architecture, thus avoiding a costly train procedure. It should also be noted that test accuracy is only allowed to be used for an offline evaluation of a given NAS algorithm, which enables a fair comparison between different algorithms.

NAS-Bench-301 is the first benchmark covering a realistically-sized search space of over 10^{18} architectures [26]. As it would be impossible to fully train and evaluate all the architectures in this large search space, a performance estimation of the architectures is provided by a surrogate model which was trained with a sample of architectures in the search space. Therefore, this benchmark is called a surrogate benchmark. About 60k architectures were sampled from the DARTS search space [18] by using several NAS methods like random search, evolutionary algorithms and bayesian optimization methods. Instead of exhaustively evaluating the entire search space, only these sample architectures were fully trained and evaluated which were then used to train several surrogate models. When the performance of a given architecture is queried, the best performing surrogate model's prediction is returned.

Similar to *DARTS* search space, each convolutional cell is a DAG consisting of 2 input nodes, 4 intermediate nodes and 1 output node in which the outputs of all intermediate nodes are concatenated. Each node x_i represents a feature map and each directed edge (i, j) represents an operation that transforms x_i. The input node in cell k receives the output feature maps from the previous two cells, cell $k - 1$ and cell $k - 2$. Input and intermediate nodes are connected by directed edges representing one of the following operations: 3 C- 3 and 5 C- 5 separable convolutions, 3 C- 3 and 5 C- 5 dilated separable convolutions, 3 C- 3 max pooling, 3 C- 3 average pooling, identity, and zero. The output of an intermediate node is computed considering all of its immediate predecessors and operations represented by the edges. An architecture is then formed by stacking multiple cells together. In the cells located at the 1/3 and 2/3 of the total depth of the network, a stride of 2 is applied in the first operation in order to reduce the input feature map's height and width by a factor of two. These cells are called *reduction cells*; and a stride of 1 is applied for the remaining cells, which are called *normal cells*.

IV. SOLUTION NETWORK-BASED SURROGATE MODELS

A category of surrogate models often used in evolutionary optimization is the so-called similarity-based models (see Tong et al. [28] for a taxonomy). These models predict the fitness of a solution by using a set of similar solutions whose true fitness values are known. Three commonly used strategies are fitness inheritance, k-Nearest Neighbors regression (kNN-R) and fitness imitation. Fitness inheritance predicts the fitness of a new solution based on the fitness of its parents. In kNN-R, the fitness of a new individual is determined by its k nearest neighbor individuals. In fitness imitation, the population is clustered into several groups, e.g. using the k-means algorithm, and one individual is selected as the representative of each group. The representative individuals are evaluated using the true fitness function, whereas the fitness values of the remaining individuals in the same cluster are predicted from the representative individual (typically the one closest to the center of the cluster) based on a distance measure [12].

Inspired by these strategies, we have developed the solution network approach. The solutions found in the search process are added as the nodes of this network. Edge (i, j) is added between nodes i and j if the distance between the corresponding solutions is less than or equal to a threshold, δ. This is the single linkage hierarchical clustering, one of the oldest clustering methods (see [20]), which results in maximal connected subgraphs (i.e. clusters). The distance measure is based on a binary encoding of solutions. Thus, two solutions are connected by an edge only if they are sufficiently similar. The prediction of a solution's accuracy is done by using its neighbors in the network.

The nodes of the network are grouped into two subsets, the set of nodes corresponding to solutions whose accuracy values are predicted, $\mathcal{E}$, and those whose true validation accuracy values are known, $\mathcal{T}$. Determining whether a node will go into the set $\mathcal{E}$ or $\mathcal{T}$ is done differently in the local search algorithm (discussed in Section V-A) and in the differential evolution algorithm (discussed in SectionV-B). In this section, we first describe the distance measures used for neural architecture encodings in Section IV-A, and present a new encoding which is an extension to the existing path encoding to alleviate its shortcomings in some scenarios. We describe our first prediction method, Weighted Average-based prediction, in Section IV-C and the second one, Regression-based prediction, in Section IV-D.

A. DISTANCE MEASURES FOR ARCHITECTURE ENCODINGS

There is a quite expansive literature on similarity/distance measures of graphs. Emmert-Streib et al. [8] provide a very good review of the graph matching and graph similarity literature. In [4], 76 distance and similarity measures based on binary vectors, some of which date back to early 1900s have been identified.

Since the DAGs representing neural networks have both node and arc labels, the distance measure for them must take all of these into account. A similar problem exists in chemistry, where a stream of research deals with distance/similarity measures for graph models of molecular structures, in which nodes represent atoms and there are labels associated with both nodes and edges [19]. A common approach for such graphical models is to create a binary vector, $\mathbf{F}$, where each element $\mathbf{F}_i$ equals 1 if a graphical substructure exists or 0 if not. For some organic compounds researchers define thousands of substructures. For example, Varmuza et al. [29] use a total of 1365 substructures that include rings and trees. Given such a binary vector, a widely used distance measure is the Jaccard distance (also known as the Tanimoto index/distance) defined as

$$JD(\mathbf{E}, \mathbf{F}) = \frac{b+c}{a+b+c} \tag{1}$$

where, $a = \mathbf{E} \cdot \mathbf{F}$ (the number of features that exist in both), $b = \mathbf{E} \cdot \neg\mathbf{F}$ (the number of features that exist only in $\mathbf{E}$), $c = \neg\mathbf{E} \cdot \mathbf{F}$ (the number of features that exist only in $\mathbf{F}$) and $d = \neg\mathbf{E} \cdot \neg\mathbf{F}$ (the number of features that do not exist in both $\mathbf{E}$ and $\mathbf{F}$). Another widely used distance is the Hamming distance (H) and its normalized version (NH):

$$H(\mathbf{E}, \mathbf{F}) = b + c \tag{2}$$

$$NH(\mathbf{E}, \mathbf{F}) = \frac{b+c}{a+b+c+d} \tag{3}$$

Recently, the idea of using presence/lack of graphical substructures to represent a graph was introduced in the context of NAS by White et al. [33]. They introduce a binary feature for each possible path from the input to the output node of an architecture cell, given in terms of the operations (e.g., input $\rightarrow$ conv1 $\times$ 1 $\rightarrow$ pool3 $\times$ 3 $\rightarrow$ output would be a path). The encoding of an architecture is represented by a binary vector mapping corresponding paths to 1s and lacking paths to 0s.

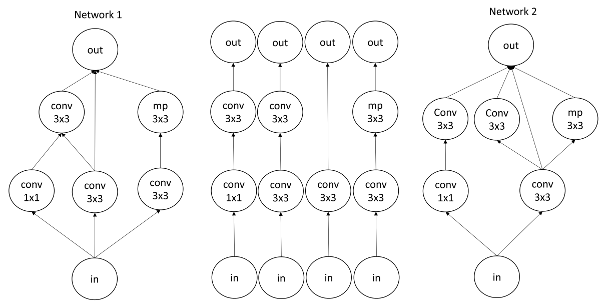


FIGURE 1. NAS-Bench-101: Same path encoding for two different neural network architectures.

Figure 1 depicts two networks that have the same four paths out of 364 possible paths for the NAS-Bench-101 benchmark instances. Thus, path encoding would result in a distance of 0 between these two networks ($b = c = 0$ using the notation given above for Eqn. 1). To alleviate this shortcoming, we introduce a set of new substructures for NAS-Bench-101 and NAS-Bench-301 to extend path encoding.

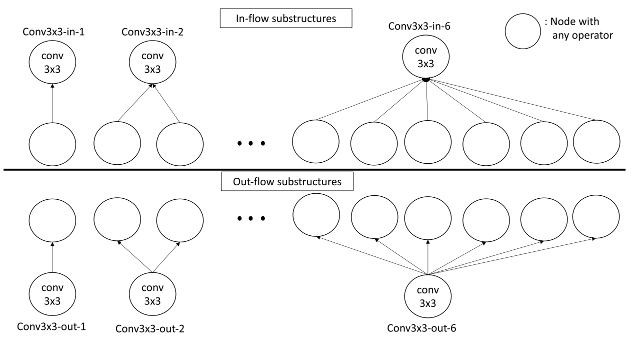


FIGURE 2. NAS-Bench-101: In-flow/out-flow substructures for a node with conv3 × 3 operator.

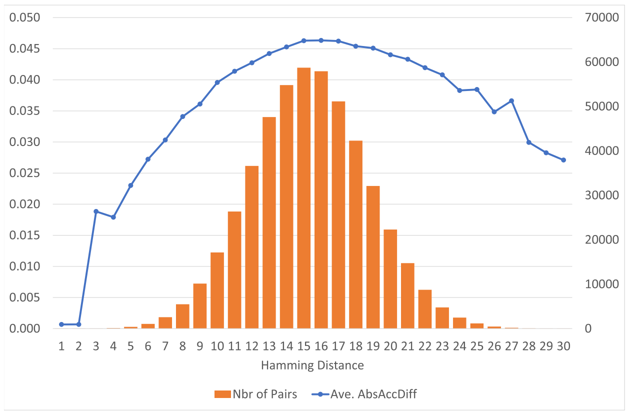


FIGURE 3. NAS-Bench-101: Hamming distance versus average *AbsAccDiff* for 1000 randomly generated neural networks.

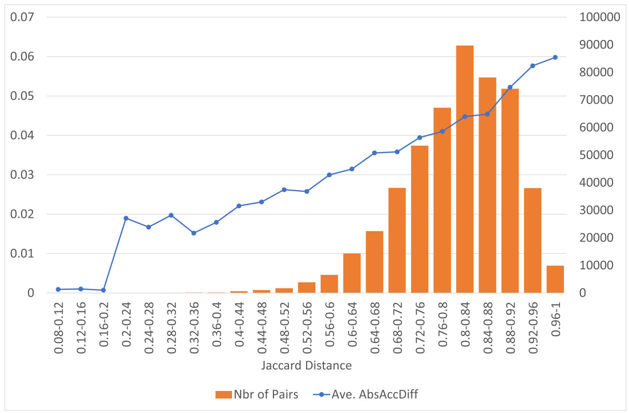


FIGURE 4. NAS-Bench-101: Jaccard distance versus average *AbsAccDiff* for 1000 randomly generated neural networks.

B. NEW GRAPH SUBSTRUCTURES FOR EXTENDING THE PATH ENCODING

Below, we first present, in detail, the approach we took for the NAS-Bench-101 dataset, and then we discuss the modifications made for the NAS-Bench-301 dataset.

In NAS-Bench-101, a node in the graph can have at most 6 in-coming or out-going edges. The number of edges that go into and out of a node with a specific operator can differentiate architectures with the same path encoding. Figure 2 depicts 12 substructures we propose for a conv3 × 3 node. We define the same in-flow and out-flow substructures for conv1 × 1 and mp3 × 3 nodes. Furthermore, we define 6 in-flow substructures for both *out* and *in* nodes. This results in a total of 48 substructures in addition to the in-out paths. Hence, the size of the binary encoding vector, n, equals the number of in-out paths (364) plus the number of the new additional substructures (48), which is 412. By definition, $n = a + b + c + d$, as defined for Eqn 1. Using the extended binary encoding of size $n = 412$, distance calculation between Networks 1 and 2 in Figure 1 would give $b = 4$ (the number of substructures that exists in Network 1 but lack in Network 2, namely, con3 × 3-out-2, in-out-3, conv3 × 3-in-2, out-in-3), $c = 3$ (the number of substructures that exist in Network 2 but lack in Network 1, namely, con3 × 3-out-3, in-out-2, out-in-4). 6 of the in-flow/out-flow substructures exist in both Network 1 and Network 2 (namely, conv1 × 1-in-1, conv1 × 1-out-1, conv3 × 3-in-1, conv3 × 3-out-1, mp3 × 3-in-1, mp3 × 3-out-1), and since the two networks share 4 in-out paths, $a = 6 + 4 = 10$. This means $d = 412 - 17 = 395$, i.e. 395 substructures lack in both of the networks.

For the distance measure to perform well within the search heuristic, any two networks with a small distance between them should have similar validation accuracy. To test this, a set of 1000 randomly generated NAS-Bench-101 neural networks were used. All networks were tested for graph-isomorphism to verify that there is no isomorphic pair. For all pairs of networks, their pairwise distances using JD and H, and the absolute difference between their accuracy values, *AbsAccDiff*, were calculated. Figures 3 and 4 depict the average *AbsAccDiff* as a function of the Hamming distance and Jaccard distance, respectively. It is important to note that for all pairs of non-isomorphic networks the encoding yields a positive value for $b + c$.

Comparing the two charts, we can see that the Jaccard distance performs better: as the Jaccard distance between two networks gets larger the difference in their accuracy values grows approximately linearly, whereas, we see that beyond a certain threshold Hamming distance (around 16-17) *AbsAccDiff* decreases as the Hamming distance increases.

Tong et al. [28] state that "the accuracy of similarity-based models deteriorates significantly when the problem is highly nonlinear and/or the search space is enormous." Thus they recommend using similarity-based models as local surrogates. Figure 4 depicts a picture that is consistent with their recommendation; the similarity-based fitness estimation is likely to work better when Jaccard distance between solutions is small.

Ideally, one would want a graph distance measure to be fine-grained, yielding different distances for different pairs of graphs. Dehmer et al. [5] use cumulative similarity/distance distributions to visualize this characteristic of the Jaccard distance and the graph edit distance (GED). They observe that GED's cumulative distance distribution has a step-function

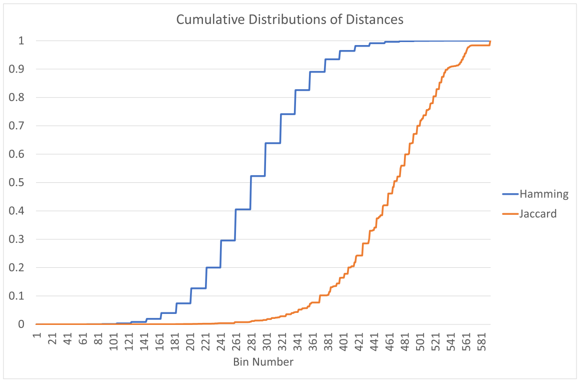


FIGURE 5. NAS-Bench-101: Cumulative distributions of the Jaccard and Normalized Hamming distances (each divided into 593 bins covering their respective ranges).

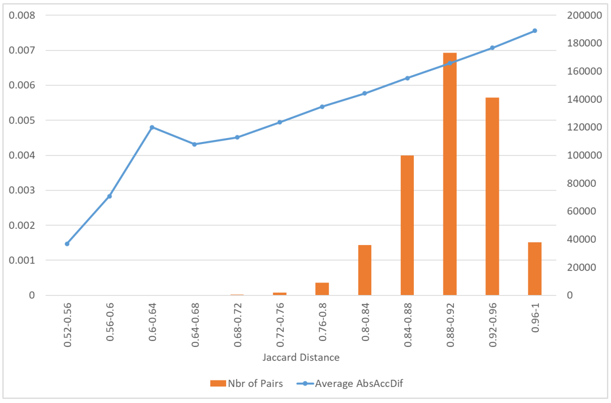


FIGURE 6. NAS-Bench-301: Jaccard distance versus average *AbsAccDiff* for 1000 randomly generated neural networks.

characteristic, where for a given GED a cluster of graphs exist with the same GED. They demonstrate that the distribution of the Jaccard distance is significantly smoother and becomes more smooth as larger sets of graph substructures are used in its calculation. In Figure 5 we plot the cumulative distance distributions for the Jaccard and Hamming distances and observe that Jaccard distance's distribution is significantly smoother than Hamming distance's as well. Hence we conclude that the Jaccard distance based on the 412 graph substructures we define for NAS-Bench-101 architectures provides a sufficient level of discriminating power between solutions.

Compared to NAS-Bench-101, NAS-Bench-301 places tighter constraints on the structure of the neural architectures. Thus, the extension of the path encoding requires a different set of substructures. For each regular node (excluding nodes c_{k-2}, c_{k-1} and c_k) we add 7 binary digits for each of the two in-coming edges. For each in-coming edge, there is one digit for each operator type (taking the value 1 if the edge has the corresponding operator and 0 otherwise). This results in a total of 56 binary digits that are added the path encoding.

Furthermore, as we have done for NAS-Bench-101 architectures, out-going edges from each node are also encoded. For a given node i, for each edge that may go out of that node, say (i, j), 7 binary digits, one for each edge operator type, are added. So, for instance, for node c_{k-2}, there could be edges to nodes 0, 1, 2, 3, which results in adding 28 binary digits, whereas for node 0, there could be edges to nodes 1, 2, 3, resulting in 21 binary digits. This results in a total of 98 binary digits added to the path encoding leading to a total length of 11358.

Figure 6 depicts the average *AbsAccDiff* as a function of the Jaccard distance between pairs of 1000 random networks, and, as was the case for NAS-Bench-101 architectures, as the Jaccard distance between two networks gets larger the difference in their accuracy values grows approximately linearly. Here, the relationship can be stated as piecewise linear, however, in the Jaccard distance range in which almost all data falls (i.e. $JD > 0.68$) the relationship is linear.

C. WEIGHTED AVERAGE-BASED PREDICTION

The weighted average method is based on the idea that neural networks that are similar to each other have similar validation accuracy performances. Thus, given a network whose accuracy is to be predicted, the simplest approach that uses the solution similarity network would be taking the average of its' neighbors accuracy values, as shown in Eqn 4:

$$\frac{\sum_{j \in N^T(i)} A(j)}{|N^T(i)|} \tag{4}$$

Here note that only the neighbors with known true validation accuracy are used in this formula. Our preliminary computational analysis of this simple average approach showed that it did not work well after testing for several δ values. Therefore, we tested a weighted-average method, which utilized the similarity of the neighbors. The empirical evidence discussed in the previous section shows that given two solutions i and j, if $JD(i, j)$ is small, we can expect, on the average, their accuracy values to be quite close to each other. Then, a reasonable and simple method for calculating $\tilde{A}(i)$, the predicted accuracy for solution i, could be given by Eqn. 5:

$$\frac{\sum_{j \in N^T(i)} S_{i,j} A(j)}{\sum_{j \in N^T(i)} S_{i,j}}, \tag{5}$$

where $S_{i,j}$ gives the similarity between solutions i and j, $A(j)$ denotes the true (queried) accuracy of solution j, and $N^T(i)$ is the set of neighbors of solution i whose true accuracy are known (please refer to Table 3 for a complete list of notations). We repeated our preliminary computational analysis of this weighted average approach, and saw that although it was better than the simple average, it still was not yielding results better than those obtained by DE of [1].

In the third, and final method, the weighted average approach was enhanced by using not only the solutions in $N^T(i)$ but also $N^E(i)$. Since the main objective of fitness estimation is to reduce the number of solutions for which $A(j)$ is calculated, $N^T(i)$ is likely to be too small for some i for Eqn. 5 to work well. For example, consider the similarity network depicted in Figure 7. Calculation of $\tilde{A}(8)$ with Eqn. 5

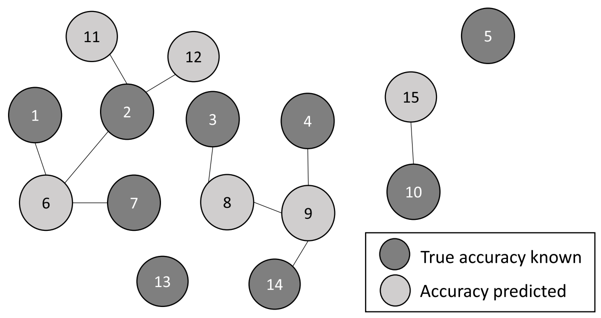


FIGURE 7. Example 1 – Similarity network.

uses $A(3)$. Let's assume $JD(4, 9)$ is very small, say almost 0, and $\tilde{A}(9) \approx A(9)$. In that case, including $\tilde{A}(9)$ in the fitness estimation of solution 8 is likely to yield a better result than only using $\tilde{A}(3)$. Thus, rather than having a binary categorization of the solutions as those that can be used in estimation and others that cannot, one could benefit from a reliability measure, $r_j \in (0, 1]$, for the fitness value associated with a solution j. Letting $N(j)$ denote the set of neighbors of solution j, we define r_j as in Eqn. 6

$$r_j = \begin{cases} \dfrac{1}{|N(j)|} \sum_{i \in N(j)} r_i \exp^{-JD(i,j)} & \text{if } j \in \mathcal{E} \\ 1 & \text{if } j \in \mathcal{T} \end{cases} \tag{6}$$

Since calculation of r_j may require r_i for some $i \in \mathcal{E}$, an iterative procedure is needed, which stops when all r values converge.

After r_j values converge, one can calculate $\tilde{A}(j)$ as in Eqn. 7, where $e^{-JD(i,j)}$ is a measure of similarity between solutions i and j:

$$\frac{\sum\limits_{i \in N^E(j)} r_i e^{-JD(i,j)} \tilde{A}(i) + \sum\limits_{i \in N^T(j)} r_i e^{-JD(i,j)} A(i)}{\sum\limits_{i \in N(j)} r_i e^{-JD(i,j)}} \tag{7}$$

As was the case for r_j, calculation of $\tilde{A}(j)$ is an iterative process that stops when sufficient convergence is obtained. Note that for any given solution j, the iterative calculation of r_j and $\tilde{A}(j)$ should be done for all nodes in $\mathcal{E}$ of the maximally connected subgraph of node j, $MCS(j)$. For the example network, $MCS(8)$ equals $\{8, 9\}$, thus iterations are stopped when convergence is obtained for both 8 and 9.

Given a new solution for which accuracy prediction will be done for the first time, both r_j and $\tilde{A}(j)$ can be initialized to some arbitrary value (we use 0.5) when starting the iterations. For a solution whose r_j and $\tilde{A}(j)$ were calculated in a previous iteration of the search algorithm and are being updated in the current iteration, r_j and $\tilde{A}(j)$ are initialized with their current values.

The calculations of r_j and $\tilde{A}(j)$ are depicted in Tables 1 and 2 for the example given in Figure 7. In Table 1 the neighbors of each solution and edge distances are given, along with whether true or predicted accuracy is to be determined

TABLE 1. Example 1 – data for reliability, r_j, calculations.

j	True?	Neigh, i	$JD(j,i)$	j	True?	Neigh, i	$JD(j,i)$
1	1	$\{6\}$	0.07724	6	0	$\{1,2,7\}$	0.07724, 0.26312, 0.45618
2	1	$\{6\}$	0.26320	7	1	$\{6\}$	0.45618
3	1	$\{8\}$	0.46528	8	0	$\{3,9\}$	0.46528, 0.53702
4	1	$\{9\}$	0.23907	9	0	$\{4,8\}$	0.23907, 0.53702
5	1	$\{\}$		10	1	$\{\}$	

for each solution. In Table 2 the iterative calculation of r_j values are shown. By definition, for a solution whose true accuracy is used (i.e. $j \in \mathcal{T}$), $r_j = 1$. Since all neighbors of $j = 6$ are in $\mathcal{T}$, r_6 needs to be calculated only once. On the other hand, r_8 and r_9 require iterative calculation until convergence (here we assume convergence occurs if the change in both r_8 and r_9 from their previous iteration values are less than or equal to 0.001). Once r_j values converge, $\tilde{A}(j)$ values are calculated as shown in Table 2, assuming $A(j)$ equals 0.9767, 0.5420, 0.8550, 0.6619, 0.8618, 0.725, and 0.9729, for $j = 1, 2, 3, 4, 5, 7$ and 10, respectively. As in r_6, only one calculation is sufficient for $\tilde{A}(6)$, since all neighbors of 6 are in $\mathcal{T}$. On the other hand, for $\tilde{A}(8)$ and $\tilde{A}(9)$ six iterations are needed for convergence.

Considering the prediction approach described above, there are two critical decisions to be made: (i) between which nodes to add the edges, (ii) which nodes' accuracy to predict. We refer to the latter as the node type decision. These decisions affect both the quality of the accuracy estimations and total time spent for neural network training and testing in order to obtain the *true* accuracy values. Overall, these two decisions are closely linked to how the similarity network grows within the search algorithm. Hence, these decisions are handled differently within the local search and the differential evolution algorithms, as discussed in Sections V-A and V-B.

D. REGRESSION-BASED PREDICTION

As depicted in Figures 4 and 6, there is approximately a linear relationship between the Jaccard distance between two solutions and the absolute difference between their accuracy values. Motivated by this observation, a linear regression-based estimation approach is developed.

Let $N^T(k)$ be the set of neighbors of solution k with true accuracy values. Choosing an anchor solution $i \in N^T(k)$, one can obtain a regression model to predict the accuracy of a solution as a function of its Jaccard distance to i. The regression model can be built using the distances and accuracy values of solutions $j \in N^T(k) \setminus \{i\}$ as long as $|N^T(k)|$ is not too small (e.g. [11] recommends the sample size in linear regression to be at least 8 for data with low variance). Thus, given $\{(JD(i,j), A(j)) : j \neq i, j \in N^T(k)\}$ the regression equation for estimation would be $\tilde{A}_i(j) = b_{0,i} + b_{1,i}JD(i,j)$ and predicted accuracy of solution k based on anchor i, denoted by $\tilde{A}_i(k)$ would be $b_{0,i}+b_{1,i}JD(i,k)$. Clearly, different anchor solutions would yield different regression equations.

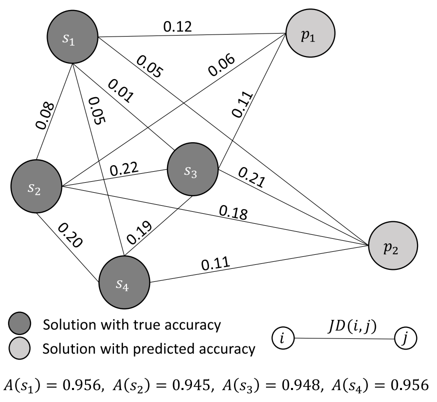
Since at a given iteration of a search algorithm there are several solutions whose accuracy will be predicted, one can

TABLE 2. Example 1 – reliability, r_j, and predicted accuracy, $\tilde{A}$, calculations.

Iter.	0	1	2	3	4	Iter.	0	1	2	3	4	5	6
r_6	0.5	0.77599				$\tilde{A}(6)$	0.5	0.76467					
r_8	0.5	0.46010	0.47173	0.46833	0.46932	$\tilde{A}(8)$	0.5	0.73765	0.77734	0.79764	0.80103	0.80276	0.80305
r_9	0.5	0.53980	0.52814	0.53154	0.53054	$\tilde{A}(9)$	0.5	0.62007	0.68147	0.69173	0.69697	0.69785	0.69830

TABLE 3. Notation.

$A(i)$	True validation accuracy of solution i
$\tilde{A}(i)$	Predicted accuracy of solution i
$\mathcal{N}_g$	Set of new solutions generated in generation/iteration g
$\mathcal{T}$	Set of solutions whose true validation accuracy is obtained/queried
$\mathcal{E}$	Set of solutions whose accuracy is predicted
$G(V, E)$	Solution similarity network with set of nodes V and edges E
$N(v)$	Neighbors of node v in $G(V, E)$
$N^T(v)$	Neighbors of node v with true accuracy in $G(V, E)$
$N^E(v)$	Neighbors of node v with predicted accuracy in $G(V, E)$
$\mathcal{C}$	Candidate edges in the current generation
$JD(v, w)$	Jaccard distance between binary encodings of solutions v and w
δ	maximum Jaccard distance allowed for any edge in $G(V, E)$



s1
s2
s3
s4
p1
p2
0.12
0.05
0.06
0.01
0.08
0.05
0.11
0.22
0.21
0.18
0.20
0.19
0.11
● Solution with true accuracy
○ Solution with predicted accuracy
i
JD(i, j)
j
A(s1) = 0.956, A(s2) = 0.945, A(s3) = 0.948, A(s4) = 0.956

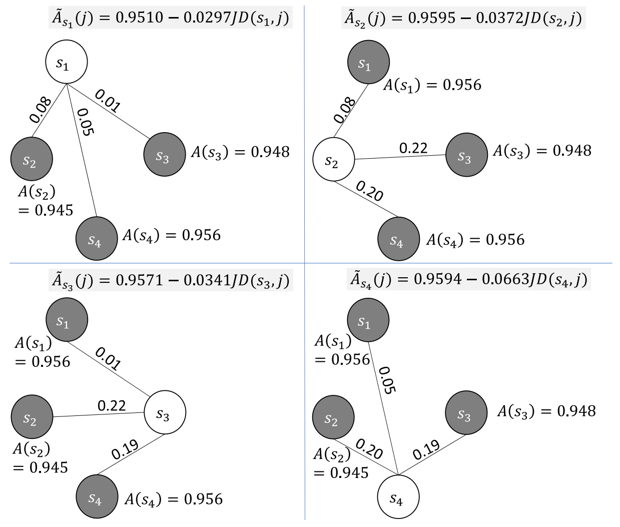
FIGURE 8. Example 2 – Regression-based prediction: solution set.

increase the set of solutions used to obtain each regression equation as follows. Let $\mathcal{P}$ denote the set of solutions p_i whose accuracy values will be predicted in the current iteration such that $N^T(p_i) \neq \emptyset$. Then, the regression models can be based on the graph $G_R(R, E_R)$ comprised of nodes $\mathcal{R} = \bigcup_{p_i \in \mathcal{P}} N^T(p_i)$ where E_R is the set of edges between nodes in R.

In the example depicted in Figure 8, $\mathcal{P} = \{p_1, p_2\}$, $N^T(p_1) = \{s_1, s_2, s_3\}$, $N^T(p_2) = \{s_1, s_2, s_3, s_4\}$, and therefore, $\mathcal{R} = \{s_1, s_2, s_3, s_4\}$.

For $i \in R$, let $E_R(i)$ denote the set of edges connected to i in $G_R(R, E_R)$. Letting n_{min} denote the minimum sample size required for running a regression model, one can obtain a regression equation for each anchor i with $|E_R(i)| \geq n_{min}$. Assuming $n_{min} = 3$, Figure 9 depicts the four subgraphs and the associated regression equations.

For a given solution k whose accuracy is to be predicted, the average of all estimations, $\tilde{A}_i(k)$, over all $i \in R$ that



$\tilde{A}_{s_1}(j) = 0.9510 - 0.0297JD(s_1, j)$
$\tilde{A}_{s_2}(j) = 0.9595 - 0.0372JD(s_2, j)$
$\tilde{A}_{s_3}(j) = 0.9571 - 0.0341JD(s_3, j)$
$\tilde{A}_{s_4}(j) = 0.9594 - 0.0663JD(s_4, j)$
A(s1) = 0.956
A(s2) = 0.945
A(s3) = 0.948
A(s4) = 0.956
0.08
0.05
0.01
0.22
0.20
0.19

FIGURE 9. Example 2 – Regression-based prediction: models.

are connected to k can be used as an aggregate predictor. Using the regression equations in Figure 9, $\tilde{A}_i(p_1)$ equals 0.9475, 0.9573 and 0.9534, for i equals s_1, s_2 and s_3, respectively, whereas $\tilde{A}_i(p_2)$ equals 0.9496, 0.9528, 0.9499 and 0.9521 for i equals s_1, s_2, s_3, and s_4, respectively. Thus, $\tilde{A}(p_1) = 0.9527$ and $\tilde{A}(p_2) = 0.9511$.

V. SEARCH ALGORITHMS

We implemented the proposed surrogate models in two search algorithms: surrogate-assisted local search (SuALS) and surrogate-assisted differential evolution (SuADE). Local Search (LS) is a strong NAS performer and suggested to be used as a baseline for NAS [25], [34]. On the other hand, it is shown by Awad et al. [1] that DE was able to perform better than RE which is another strong EA on NAS-Bench-101 benchmark. They conclude that DE, by keeping the population in continuous space, helps maintain diversity and explore the high-dimensional search spaces effectively. Siems et al. [26] used 10 different search algorithms to sample architectures from a large search space to create NAS-Bench-301 surrogate benchmark. They selected only two EA-based algorithms, DE and RE, as the meta-heuristics that yield a good coverage of this large search space. When the architectures discovered by each algorithm are visualized using their t-SNE embeddings, it was clear that DE was the only algorithm which was able to find good architectures in the center of the embedding space.

A. SURROGATE-ASSISTED LOCAL SEARCH

In some recent research ([25], [34]), it has been shown that LS is a strong performer in NAS. White et al. [34], using NAS-Bench-101, NAS-Bench-201 and NAS-Bench-301 datasets, show that when the noise in the validation accuracy is reduced to a minimum, hill-climbing outperforms many state-of-the-art algorithms. Schneider et al. [25] determine that the strong performance of BANANAS stems mainly from local optimization of architectures in which an architecture is mutated by changing a single operation or edge randomly and the architecture yielding the best performance becomes the next one for accuracy evaluation.

One difference of SuALS (see Algorithm 1) from a traditional local search is maintaining a solution network. Each generated solution, v_i, is added to the solution network by the updateNetw($network$, v_i, δ) function. The second difference is in the way the number of generated neighbors of an incumbent solution is limited by the parameters $maxTrue$ and $numPred$.

SuALS starts with randomly generating NI initial solutions (randArch(NI) function). The true accuracy values of all of these solutions are queried. The best of these solutions becomes the first incumbent solution, v_0. Each query of true accuracy uses up the budget, B, which could be specified either in number of queries or the training/validation time required. From a given incumbent solution, first up to $maxTrue$ neighbors are generated by mutation and their true accuracy values are queried. randNewNeigh(v_i) in line 8 is the mutation method of [25], i.e. a single edge or operator of architecture v_i is changed. If this happens to be a solution that already exists in the network, another neighbor is randomly selected. If all neighbors of the incumbent have already been generated (which may happen in NAS-Bench-101 instances) and v_i is NULL, then the algorithm jumps to Line 39. If any neighbor v_i improves the incumbent it becomes the new incumbent solution (line 36). If none of them improves the incumbent (i.e. $ctrTrue$ equals $maxTrue$), then $numPred$ neighbors are generated by mutation and their accuracy values are predicted. This set of solutions are kept sorted in $setPred$ in decreasing predicted accuracy. The solution with the highest predicted accuracy is picked from $setPred$ (line 27), its true accuracy is queried and removed from $setPred$ until the incumbent is improved or $maxTrueFromPred$ solutions' true accuracy values are queried (by definition $maxTrueFromPred \leq NumPred$). If $maxTrueFromPred$ limit is reached without improving the incumbent, the new incumbent becomes the solution with next largest true accuracy (NextBestTrue() function in line 39). Two versions of NextBestTrue() were implemented. In the first one, a solution is selected as the incumbent only once whereas in the second one this restriction is not used. Our experiments revealed that, for the NAS-Bench-101 instances the second version was often getting stuck at an incumbent, so the first version was used for these instances. For the NAS-Bench-301 instances, the second version was used.

Algorithm 1 SuALS

```
   Input : $NI$; $\delta$; $B$; $maxTrue$; $maxTrueFromPred$; $numPred$
 1 $ctrTrue \leftarrow 0$; $ctrTrueFromPred \leftarrow 0$
 2 $\{a_1, \ldots, a_{NI}\} \leftarrow$ randArch($NI$)
 3 updateUsedBudget($usedBudget$, $\{a_1, \ldots, a_{NI}\}$)
 4 updateNetw(network, $\{a_1, \ldots, a_{NI}\}$, $\delta$);
     $b \leftarrow \arg\max_{j=1,\ldots,NI} f(a_j)$
 5 $v_0 \leftarrow a_b$; $i \leftarrow 1$   // local search starts with
     the best solution
 6 while $usedBudget < B$ do
 7     if $ctrTrue < maxTrue$ then
           // randomly pick a neighbor w/o
           replacement
 8         $v_i \leftarrow$ randNewNeigh($v_{i-1}$)
 9         if $v_i <> NULL$ then
10             updateNetw($network$, $v_i$, $\delta$)
11             $v_i.accType \leftarrow$ true; $f(v_i) \leftarrow$
                 queryTrueAccuracy($v_i$)
12             updateUsedBudget($usedBudget$, $v_i$)
13             $ctrTrue \leftarrow ctrTrue + 1$
14         end
15     end
16     else
17         if $ctrTrueFromPred = 0$ then
               // rand pick $numPred$ neighbors
               w/o replacement
18             for ($ctrPred \leftarrow 0$; $ctrPred \leq$
                 $numPred$; $ctrPred \leftarrow ctrPred + 1$) do
19                 $n_i \leftarrow$ randNewNeigh($v_{i-1}$)
20                 if $n_i <> NULL$ then
21                     $n_i.accType \leftarrow$ predicted;
                         $setPred$.ADD($n_i$)
22                     updateNetw($network$, $n_i$, $\delta$)
23                 end
24             end
25             predictAccuracy($network$, $setPred$)
26         end
27         $v_i \leftarrow$ PickBestPredictedSoln($setPred$)
28         if $v_i <> NULL$ then
29             $ctrTrueFromPred \leftarrow ctrTrueFromPred + 1$
30             $setPred$.REMOVE($v_i$)
31             $v_i.accType \leftarrow$ true; $f(v_i) \leftarrow$
                 queryTrueAccuracy($v_i$)
32             updateUsedBudget($usedBudget$, $v_i$)
33         end
34     end
35     if $v_i <> NULL\ AND\ f(v_i) > f(v_{i-1})$ then
           // new solution becomes incumbent
36         $i \leftarrow i + 1$; $ctrTrue \leftarrow 0$; $ctrTrueFromPred \leftarrow 0$;
             $setPred \leftarrow NULL$
37     end
38     else if $v_i = NULL\ OR$
         $ctrTrueFromPred = maxTrueFromPred$ then
39         $v_i \leftarrow$ NextBestTrue($network$, $v_i$)
40         $i \leftarrow i + 1$; $ctrTrue \leftarrow 0$; $ctrTrueFromPred \leftarrow 0$;
             $setPred \leftarrow NULL$
41     end
42 end
```

B. SURROGATE-ASSISTED DIFFERENTIAL EVOLUTION

Differential Evolution (DE) algorithm ([22], [27]) has been widely applied in many fields due to its effectiveness and simplicity. As a type of Evolutionary Algorithm (EA), it is a population-based search algorithm and makes use of basic operations like selection, mutation and crossover, with several parameters to be tuned. DE maintains a population of NP D-dimensional real-valued vectors, so-called individuals, each representing a candidate solution. At generation $g = 0$, the individuals, $\{\mathbf{x}_{i,0} = (x^1_{i,0}, .., x^D_{i,0})\}$ for $i = 1, \ldots, NP$ are initialized randomly from uniform distribution $U(x^j_{low}, x^j_{high})$ for each $j = 1, \ldots, D$. After the initialization, DE performs the evolutionary operations, crossover, mutation and selection, to generate a new population. At generation g, a *mutant* vector $\mathbf{v}_{i,g} = (v^1_{i,g}, .., v^D_{i,g})$ is produced for each *target* vector $\mathbf{x}_{i,g}$ using a certain mutation operator. We tested two commonly used mutation strategies, $DE/rand/1$ (Eq. 8) and $DE/current\text{-}to\text{-}best/1$ (Eq. 9) in the literature.

$$\mathbf{v}_{i,g} = \mathbf{x}_{r1,g} + F\dot{(}\mathbf{x}_{r2,g} - \mathbf{x}_{r3,g}) \tag{8}$$

$$\mathbf{v}_{i,g} = \mathbf{x}_{i,g} + F\dot{(}\mathbf{x}_{best,g} - \mathbf{x}_{i,g}) + F\dot{(}\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \tag{9}$$

In the notation $DE/target/nbr$, *target* specifies how the target vector is chosen. For example, *rand* means the target vector is randomly chosen, while *current-to-best* means the current target vector and the best-so-far vector are used as the base vectors. *nbr* in the notation indicates the number of vector differences contributing to the differential. $F > 0$ is a parameter for scaling the difference vector, and $r1$, $r2$, $r3$ are distinct integers randomly selected from $[1, NP]$, which are also different from i.

Each pair of target vector and its corresponding mutant vector is subjected to a binomial crossover operation to generate a *trial* vector, $\mathbf{u}_{i,g}$ as follows:

$$u^j_{i,g} = \begin{cases} v^j_{i,g}, & \text{if } rand \leq CR \text{ or } j = n_j \\ x^j_{i,g}, & \text{otherwise} \end{cases} \quad \text{, for } j = \{1, ..D\} \tag{10}$$

where $CR \in [0, 1)$ is the crossover rate, $rand$ is a random variate sampled from $U(0, 1)$ distribution, and n_j is a randomly selected number from the set $\{1, \ldots, D\}$. n_j is used to ensure that $\mathbf{u}_{i,g}$ gets at least one dimension from $\mathbf{v}_{i,g}$.

Before evaluating the objective function values of the newly generated trial vectors, $\mathbf{u}_{i,g}$, they are subjected to a boundary check mechanism. If a value in a dimension exceeds the bounds, then it is set to a uniformly generated random value within the specified range. In the selection phase, each target individual $\mathbf{x}_{i,g}$ competes with its corresponding trial vector $\mathbf{u}_{i,g}$ for being included in the next generation. If $\mathbf{x}_{i,g}$ is better than $\mathbf{u}_{i,g}$ in terms of the objective function value, then $\mathbf{x}_{i,g}$ is selected as a parent for the next generation, $\mathbf{x}_{i,g+1}$.

Algorithm 2 provides the pseudo-code of the SuADE algorithm. The main difference between a regular DE algorithm and SuADE is the accuracy prediction step, including the maintenance of the solution similarity network. In the initialization step two different approaches were used for generating the solutions of the initial population (denoted by the parameter *InitType*). In the first one, *NP* solutions are generated randomly. This method was used for NAS-Bench-101. In the second one, 10 solutions are generated randomly and the remaining $NP - 10$ solutions are generated randomly from the neighbors of the best one among the initial 10. This method was used for the NAS-Bench-301 instances.

In line 3 the function queryAccuracy() returns the validation accuracy of the given architecture using the methods described in [38] for NAS-Bench-101 and in [26] for NAS-Bench-301. In line 8 the function produceTrialVector() returns a trial vector as given in Eqn 10. In line 11 the function predictAccuracy() returns the predicted accuracy using the chosen surrogate model for that implementation of the search algorithm.

Unlike in SuALS, the type of a node depends on its neighbors. Therefore, updating the network and determining the type of nodes are done together. After determining the true and predicted accuracy values of the solutions in *setTrue* and *setPred*, respectively, at the beginning of each generation the highest accuracy solution in *setTrue*, X_{best}, is used to update the best solution found so far, X^* (line 7).

Algorithm 2 SuADE

```
   Input : NP; $\delta$; B; InitType; nodeUB?
 1 setPop ← initialize(NP, InitType)
 2 {setTrue, setPred} ← updateNetwAndNodeT(network,
     setPop, $\delta$, nodeUB?)
 3 queryAccuracy(usedBudget, setTrue)
 4 predictAccuracy(network, setPred)
 5 while usedBudget < B do
       // for each generation
 6     $X_{best}$ ← bestSolution(setPop)
 7     $X^*$ ← bestFitness(setTrue, $X^*$)
 8     setU ← produceTrialVector(setPop, $X_{best}$)
 9     {setTrue, setPred} ← updateNetwAndNodeT(network,
         setU, $\delta$, nodeUB?)
10     queryAccuracy(usedBudget, setTrue)
11     predictAccuracy(network, setPred)
12     $v_b$ ← bestSolution(setPred)
13     queryAccuracy(usedBudget, $v_b$)
14     setTrue.ADD($v_b$)
15     $v_b$.accType ← true
16     setPop ← nextGenPopulation(setPop, setU)
17 end
18 return($X^*$)
```

The first step of Algorithm 3 is determining the nodes to be added. Each element of the vector of solution encodings, $\mathcal{V}$, is mapped to a solution. If that solution is infeasible or it is feasible but there is a solution already in the network whose Jaccard distance to this one is 0, then it is not included in the set of new nodes, $\mathcal{N}$, to be added to the network. The second step is determining a set of candidate edges, $\mathcal{C}$, between $\mathcal{N}$ and $V \bigcup \mathcal{N}$. A candidate edge is defined between nodes n and w, where $n \in \mathcal{N}$ and $w \in V \bigcup \mathcal{N}$ if $JD(n, w) \leq \delta$. These edges are sorted in increasing $JD(n, w)$, and edge and node type decisions are made in this order. Figure 10 depicts a small example with $\mathcal{N} = \{18, 19, 20\}$, where edges in $\mathcal{C}$ are shown by dashed lines.

All nodes in $\mathcal{N}$ are added to the network. Edge addition decisions are made differently for weighted-average and regression-based prediction methods. For regression-based prediction *nodeUB*? parameter is set to FALSE. In that case,

Algorithm 3 updateNetwAndNodeT

```
   Input: $network$, $\mathcal{V}$, $\delta$, $nodeUB?$
1  $\mathcal{N} \leftarrow$ determineNewNodes($\mathcal{V}$)
2  $\mathcal{C} \leftarrow$ candidateEdges($\mathcal{N}$, $network$, $\delta$)
3  $\mathcal{S} \leftarrow$ sortByIncDistance($\mathcal{C}$)
4  for each $(v, w)$ in $\mathcal{S}$ do
5      $network \leftarrow$ addEdgeNode($(v, w)$, $network$, $nodeUB?$)
6      $\{\mathcal{T}, \mathcal{E}\} \leftarrow$ nodeType($v$, $w$, $network$)
7      if $nodeUB? = TRUE$ then
8          updateNodeLimits($\mathcal{E}$, $\mathcal{T}$, network)
9  $\{\mathcal{T}, \mathcal{E}\} \leftarrow$ nodeTypeForIsolated($\mathcal{T}$, $\mathcal{E}$, network)
10 return $network$
```

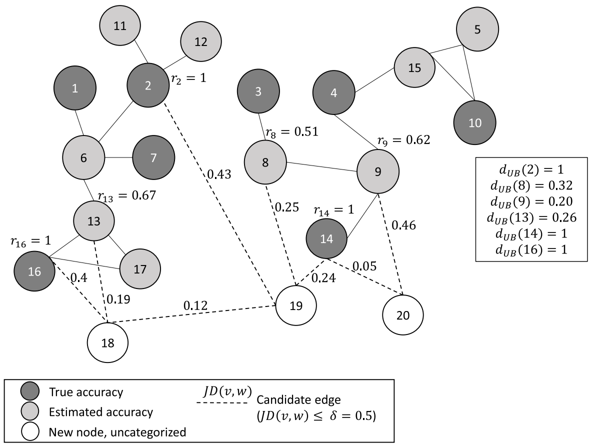


FIGURE 10. Example 4 – Set of candidate edges.

all candidate edges (v, w) are added to the network. For the weighted-average method, $nodeUB?$ is set to TRUE, and node-based upper bounds on edge distances are defined. The intuition behind this can be explained as follows. For instance, assume node $w_1 \in \mathcal{E}$ is connected to a single solution $w_2 \in \mathcal{T}$ with $JD(w_1, w_2) \approx 0$, which should yield a small error in $\tilde{A}(w_1)$. In that case, we may not want to add an edge from a new solution n to w_1 especially if $JD(n, w_1) \approx \delta$, since it is quite likely to increase the error in $\tilde{A}(w_1)$. Hence, a dynamically updated constraint is defined for connecting an edge to a node in $\mathcal{E}$. Let $d_{UB}(v)$ denote an upper limit on the distance of an edge that can be connected to node v, defined as:

$$d_{UB}(v) = \begin{cases} \min_{w \in N(v)} JD(v, w) & \text{if } v \in \mathcal{E} \\ 1 & \text{if } v \in \mathcal{T} \end{cases} \tag{11}$$

Consider node 9 in Figure 10, for which $d_{UB}(9) = .2$, that is one of its three current connections is quite similar to 9 and likely to yield a good predicted accuracy for 9. Adding edge $\{9, 20\}$ with distance 0.46 is likely to degrade the estimation quality for solution 9. It is worth noting that $d_{UB}(v)$ does not depend on the reliability values discussed in Section IV-C. An intuitive argument behind this is that the reliability values are likely to improve as the network grows, so it is more important to make the edge formation decisions based on distances alone. Of course, the effect of edge $\{9, 20\}$ on the estimation quality of 9 also depends on the node type decision for 20.

Given the upper bounds, edge $\{v, w\}$ is added to the network only if v or $w \notin \mathcal{T}$ and,

$$JD(v, w) \leq K \times \min\{d_{UB}(v), d_{UB}(w)\} \tag{12}$$

After an edge $\{v, w\}$ is added to the network, assuming v is in $\mathcal{N}$, its type is determined as follows:

(i) If $w \in \mathcal{T}$, v is added to $\mathcal{E}$;
(ii) If $w \in \mathcal{N}$, one of v or w is randomly assigned to $\mathcal{T}$ and the other is added to $\mathcal{E}$;
(iii) If $w \in \mathcal{E}$, then v is added to $\mathcal{T}$ if $\tilde{A}(w) > \alpha$, and added to $\mathcal{E}$, otherwise.

Considering case (iii) above, if α equals 0, v would always be added to $\mathcal{T}$, which is likely to improve the estimation for w. However, let's assume $\tilde{A}(w) = 0.5$, that is, solution w is predicted to be quite poor. Thus, it is quite likely that v is also a poor solution and including v in $\mathcal{T}$ would be a waste of limited computational budget. In other words, we would like to improve the estimation of good solutions rather than poor ones. Therefore, letting A^* denote the maximum true accuracy obtained so far, we set α to $\max\{A^* - 0.1, 0.60\}$, for the NAS-Bench-101 instances, and to $\max\{A^* - 0.0025, 0.94\}$, for the NAS-Bench-301 instance (these parameter values were determined by looking into the distribution of accuracy values of the solutions obtained during the search).

After determining the node types, $d_{UB}(v)$ and $d_{UB}(w)$ are updated using Equation 11. For the example in Figure 10, assuming $K = 1$, this algorithm first adds edge $\{14, 20\}$, makes the updates $\mathcal{E} = \mathcal{E} \bigcup \{20\}$ and $d_{UB}(20) = 0.05$. Then, it adds $\{18, 19\}$, sets $\mathcal{T} = \mathcal{T} \bigcup \{18\}$, $\mathcal{E} = \mathcal{E} \bigcup \{19\}$, $d_{UB}(18) = 1$, and $d_{UB}(19) = 0.12$. Finally it adds $\{13, 18\}$, and updates $d_{UB}(13)$ as $\min\{0.26, 0.19\} = 0.19$.

After the new edges are created it is possible for some $n \in \mathcal{N}$ to have no edges. Since estimation cannot be made for such *isolated* nodes, there are two options: they can either be added to $\mathcal{T}$ or not added to the network at all. The former would serve towards the exploration function of search algorithms, whereas the latter would serve the exploitation function. To strike a balance between the two, $\min\{\tau, |\mathcal{I}|\}$ isolated nodes are selected for adding to $\mathcal{T}$, where $\mathcal{I}$ is the set of isolated nodes and $\tau \in \mathbb{N}$ is a parameter. The selected ones are those that have the shortest distance to X^*, the best solution found so far (see Algorithm 2).

VI. EXPERIMENTAL RESULTS

In this section, we report the results of the experimental work done in line with the NAS best practices checklist in [16]. In the Appendix, we give full details of our answers to this checklist. Our implementation is made publicly available at Github [15]. For each of the benchmark datasets, we evaluate both the performance of the two similarity-based

TABLE 4. Parameter settings for the prediction methods.

		Best		
Algorithm	Param.	NAS-Bench-101	NAS-Bench-301	Used
SuADE-WA	$\delta; k$	$0.3; 1$	$0.5; 1.6$	$0.4; 1.4$
SuADE-RG	$\delta; n_{min}$	$0.3; 35$	$0.4; 30$	$0.4; 25$
SuALS-WA	$\delta; k$	$0.7; 1$	$0.5; 1.4$	$0.6; 1.2$
SuALS-RG	$\delta; n_{min}$	$0.5; 25$	$0.3; 40$	$0.6; 20$

predictors (weighted average and regression-based) and the performance of each search algorithm (SuADE and SuALS) coupled with each predictor.

For both SuADE and SuALS algorithms, we did not perform a dataset-specific parameter tuning process, in accordance with NAS best practices. Instead, we have adopted the hyper-parameter values that were used or recommended in earlier studies. SuADE requires the population size, *NP*, the crossover rate, *CR* and the scaling factor, *F*, and τ to be set. For *CR* and *F* we directly adopted the values in [1], setting them both to 0.5, whereas, we set *NP* to 30 rather than 20 used in [1] and we set τ to 3. The four parameters of SuALS-WA and SuALS-RG, namely, *NI*, *numPred*, *numTrue* and *maxTrueFromPred* were given the values 10, 10, 3, and 5, respectively, for both datasets.

In addition to those search algorithm parameters, solution network maintenance carried out in the prediction models also requires a couple of parameters. These parameters are δ and n_{min} for the regression-based prediction model, and δ and k for the weighted average prediction model. A robustness analysis has been carried out for these parameters, with all combinations of $\delta = \{0.3, 0.4, 0.5, 0.6, 0.7\}$ and $n_{min} = \{20, 25, 30, 35, 40\}$ for the regression-based prediction, and $\delta = \{0.3, 0.4, 0.5, 0.6\}$ and $k = \{1, 1.2, 1.4, 1.6\}$ for the weighted average prediction model. Table 4 provides the common set of parameters used for both datasets, along with the best performing parameters. The prediction performance analysis in Section VI-A and search performance analysis in Section VI-B are reported for the parameters given in the column titled "used" in this table.

SuADE, as opposed to SuALS, tends to produce solutions with relatively larger distances between them. Because of this, for a given δ it results in fewer neighbors for solutions. Thus, having a relatively large n_{min} in SuADE-RG would result in some solutions' predictions to be made by a simple average of neighbors' accuracy values.

Figures 11 through 14 depict the performances of the four algorithms with the used parameters listed in Table 4 along with those corresponding to the best and worst parameter settings (based on the accuracy obtained at the termination). In seven of these eight charts we observe a narrow gap between the best and worst performances throughout the search. The only exception is for SuALS-RG tested on NAS-Bench-101 (see Figure 14) but the gap is relatively wide only during the first half of the search process which narrows down significantly by the termination of the search. Overall, we can conclude that all four algorithms are quite robust with respect to the two parameters.

A. PREDICTION PERFORMANCE

In the NAS literature, the performance of a predictor is judged to be its generalization ability to unseen architectures. Correlation measures between the predicted and the actual accuracy values, such as Pearson, Spearman's rank, Kendall's tau rank correlations, are used to evaluate the power of a given predictor (e.g. see White et al. [35] and Ning et al. [21]). In addition to the prediction ability, the time to reach that predictive power is also considered while comparing different NAS predictors.

As stated in [35], some predictors require a costly query routine where true accuracy of some architectures are obtained, and some predictors employ a costly initialization routine. NAS benchmark datasets reduce costly true accuracy query routine to simple database queries. Each query gives not only the true accuracy of the associated neural architecture but also its total training time. Some predictors also employ an update routine where initial computations are updated. The total time spent during all these steps determine the speed of a predictor. A fair comparison between different predictors can only be made considering the time it takes to reach certain levels of prediction power.

In this study, we aimed to improve the prediction performance of our network-based model incrementally by adding a number of newly created architectures with true accuracy values at each iteration and calculating/updating predicted accuracy of new/existing architectures in the network. As our approach is very different from those in the literature, it is difficult to separate the times for the initialization, query and update routines in SuADE and SuALS algorithms in a comparable manner to the algorithms in literature. Since the queried training time of architectures is many orders of magnitude larger than the times taken to maintain the similarity network and calculate the predicted accuracy values, the number of queries and the training times obtained by these queries determine the wall-clock time of SuADE and SuALS algorithms. Pearson correlation, Spearman's rank correlation and Kendall's tau rank correlation values are reported through this wall-clock time which is equal to the sum of the queried training times from the benchmark datasets. It is also important to note that true accuracy values of the architectures with predicted accuracy values are queried in order to calculate the correlations, and never used to drive the search. Therefore, only the queried times for the architectures whose true accuracy values are used to update the model are included in the wall-clock time.

1) NAS-BENCH-101 EXPERIMENTS

All algorithms were allowed to run for a total time budget of 8×10^4 seconds. For each algorithm, we performed 500 independent runs and report the mean for each of three criteria; namely, Pearson correlation, Spearman's rank correlation and Kendall's tau rank correlation. The results are shown in Figure 15, where x-axis represents the estimated wall-clock time, as the cumulative time taken for training

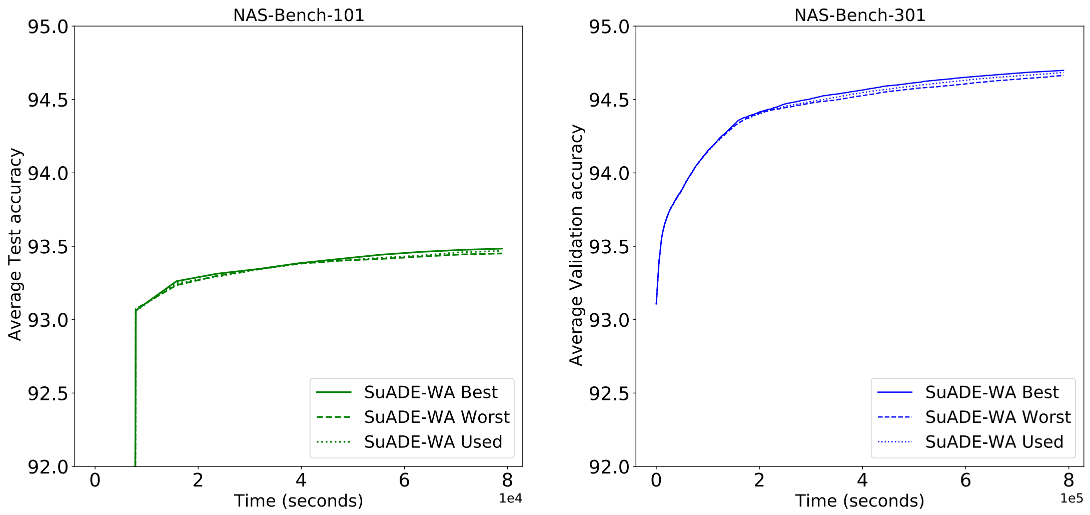


FIGURE 11. **Robustness analysis of SuADE-WA.**

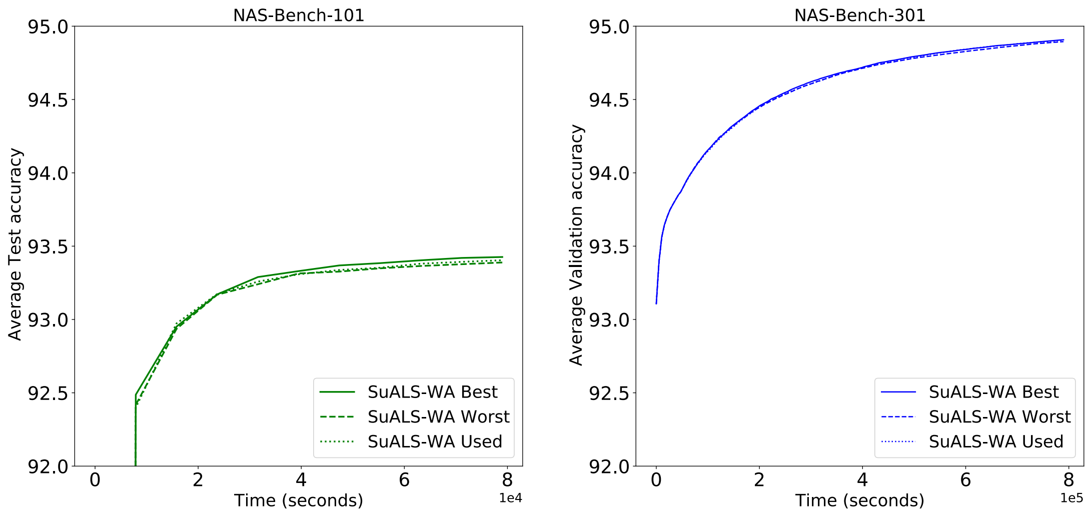


FIGURE 12. **Robustness analysis of SuALS-WA.**

each of the architectures found as returned by NAS-Bench-101. The correlation reported for a given wall-clock time is for all predicted architectures generated up to that time.

The results show that SuADE-WA and SuADE-RG perform significantly better than both SuALS versions in terms of all three criteria throughout the whole search process. Although SuALS-WA seems to perform better than SuALS-RG; both yield weak correlations.

In addition to the final prediction performance, the time taken to reach a level of prediction performance is an important criterion for assessing a NAS predictor. We observe that for the SuADE algorithms the correlations start high, and while they remain stable for the weighted average predictor, there is some decline for the regression-based predictor. In [35] Kendall's tau rank correlation values of several predictors under different initialization and query time settings are presented. At an initialization time of 10^6 seconds, BANANAS [33] exhibits the highest Kendall's tau rank correlation value of 0.8. On the other hand, BANANAS yields correlation values of approximately 0.25 and 0.55 for the initialization times of 10^4 and 10^5 seconds, respectively. As only the initialization routine is costly for BANANAS; we take this initialization time to compare it to SuADE-WA. SuADE-WA achieves a mean Kendall's tau rank correlation value of 0.66 at very early stages of the search. It also achieves the highest correlation value of 0.73 within the first 3×10^4 seconds. It is shown in [35] that XGBoost [3] which was used as a model-based predictor while creating NAS-Bench-301 performs weakly on NAS-Bench-101. This is attributed to the complexity of the NAS-Bench-101 search space as it contains more diverse set of architectures ranging from a single node and no edges, to five nodes with nine connecting edges. Thus, the correlation performance of SuADE-WA is a significant achievement.

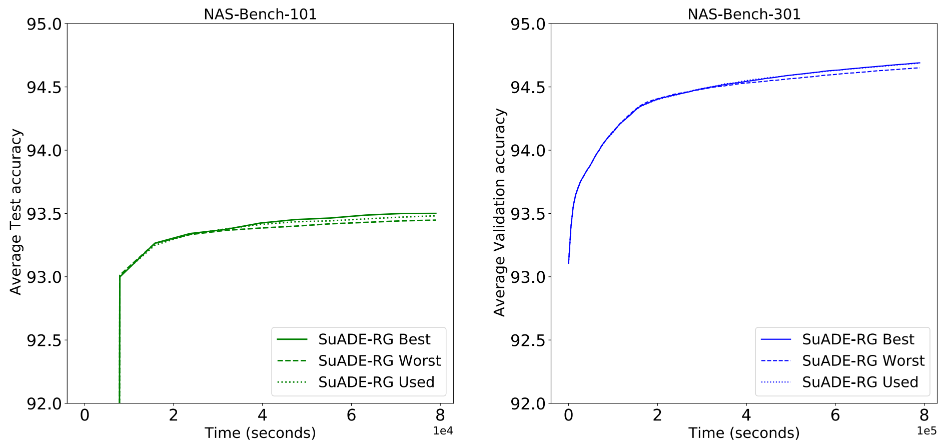


FIGURE 13. Robustness analysis of SuADE-RG.

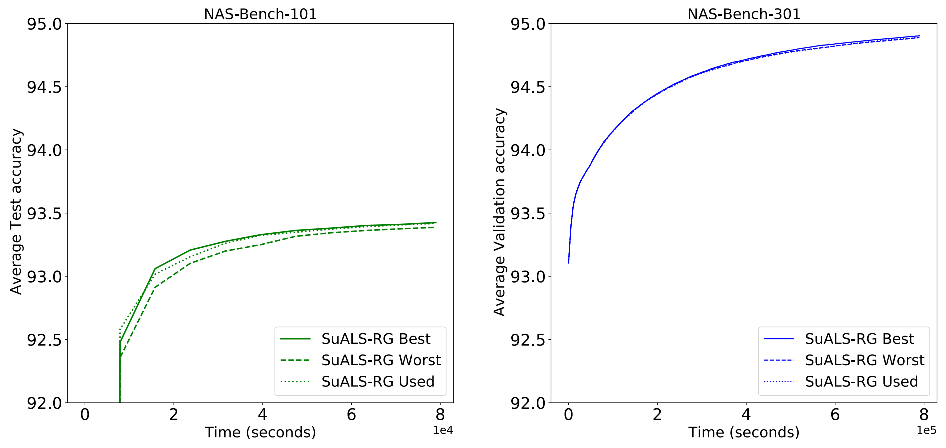


FIGURE 14. Robustness analysis of SuALS-RG.

2) NAS-BENCH-301 EXPERIMENTS

In this section we report the results of the set of experiments discussed in the previous section for the NAS-Bench-301 dataset. Each of the algorithms is allowed to run for a total budget of 150 queries. The results are shown in Figure 16. In order to be able to compare our results to other results in the literature in terms of both the predictive power and the time it takes to reach that power, we used the estimated wall-clock time provided for a given architecture, and the x-axis shows the total time spent querying the architectures during the search.

In Figure 16 we see that SuALS algorithms start with all three correlations around zero and improve quickly. SuADE algorithms start quite high and remain high throughout. SuALS-WA exceeds the performance of the SuADE algorithms with respect to Spearman's and Kendall's tau rank correlations. White et al. [35] present a comparison of several predictors under several initialization and query time budgets considering both the correlation performance and their ability to speed up NAS process. It is shown in the study that some NAS predictors that result in high Kendall's tau correlations in smaller search spaces do not perform well on large search space like NAS-Bench-301. For example, BANANAS [33] yields a very poor Kendall's tau rank correlation value of 0.1 after an initialization time of 10^6 seconds. According to the study, a strong predictor, Sum of Training Losses at Last Epoch E (SoTL-E), achieves a Kendall's tau rank correlation value close to 0.7 after a query time of 10^5 seconds. SuALS-WA achieves the same Kendall's tau value within similar time.

Overall, for all three correlation measures, SuADE-WA gives higher values throughout the search than SuALS-RG for both NAS-Bench-101 and NAS-Bench-301. Similarly, for all three correlation measures, SuADE-WA gives higher values throughout the search than SuALS-WA for NAS-Bench-101. However, for NAS-Bench-301 we see that SuALS-WA yields better rank correlations (Spearman's and Kendall's Tau).

B. SEARCH PERFORMANCE

1) NAS-BENCH-101 EXPERIMENTS

The search performances of the four algorithms developed in this research along with the Differential Evolution (DE)

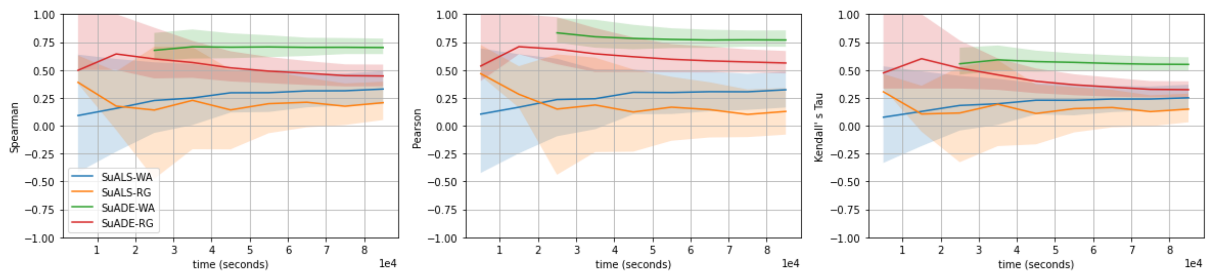


FIGURE 15. Correlation results on NAS-Bench-101 with the first and the third quartiles shaded.

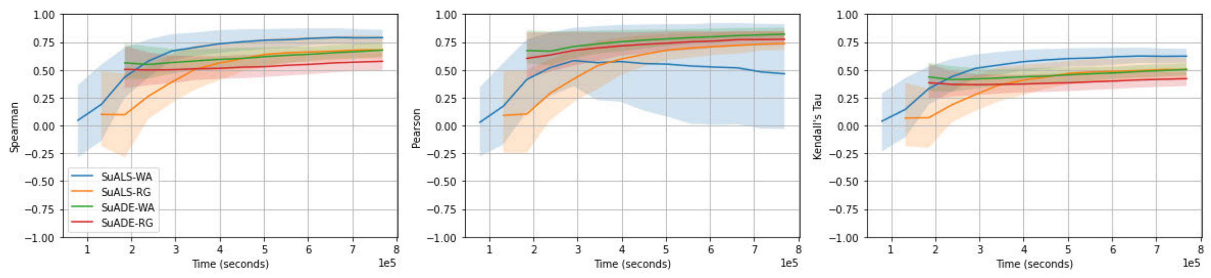


FIGURE 16. Correlation results on NAS-Bench-301 with the first and the third quartiles shaded.

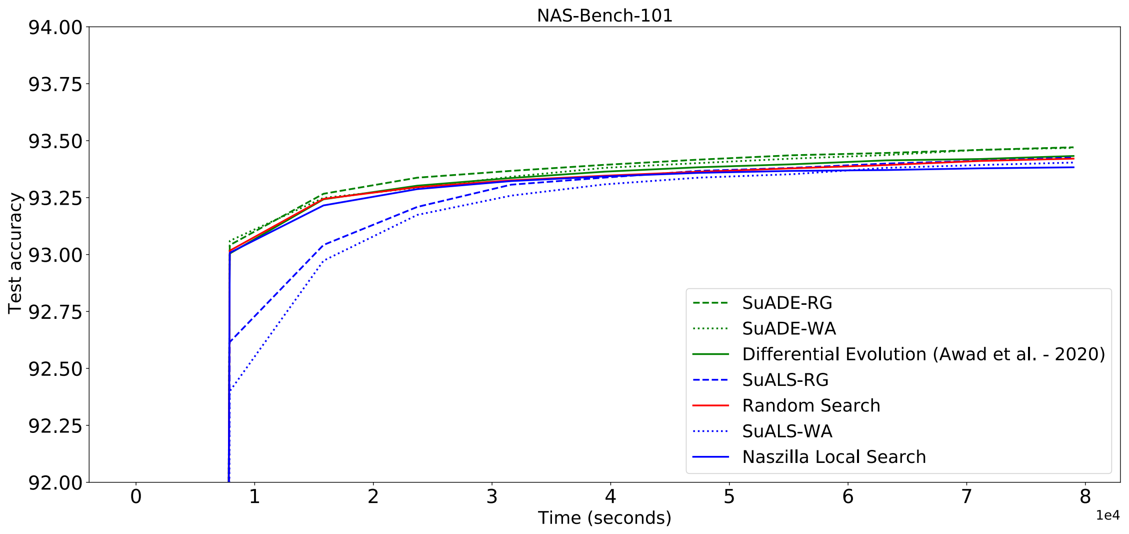


FIGURE 17. NAS-Bench-101 search performance.

algorithm of Awad et al. [1], local search algorithm (LS) and Random Search (both are available in the Naszilla website maintained in Github, and are based on the work reported in [32], [33], and [34]) for NAS-Bench-101 are shown in the Figure 17.

Looking into these results, it is seen that SuADE-RG and SuADE-WA achieved the best average test accuracy values within the specified time limit (SuADE-RG performed slightly better than SuADE-WA). Other algorithms, including the Naszilla DE and LS, did not perform better than random search. The improved performance of SuADE with the two prediction methods suggest that these fast and good prediction methods facilitate evaluation of more solutions and searching the solution space more efficiently without consuming extra wall-clock time.

It is interesting that SuALS-WA and SuALS-RG initially perform significantly worse than LS of Naszilla, even though eventually they catch up and obtain slightly better results. The initial under-performance may be partly explained by the low correlations depicted in Figure 15.

2) NAS-BENCH-301 EXPERIMENTS

Figure 18 shows the search performance of different heuristics and estimators. Based on the results, SuALS-WA reaches

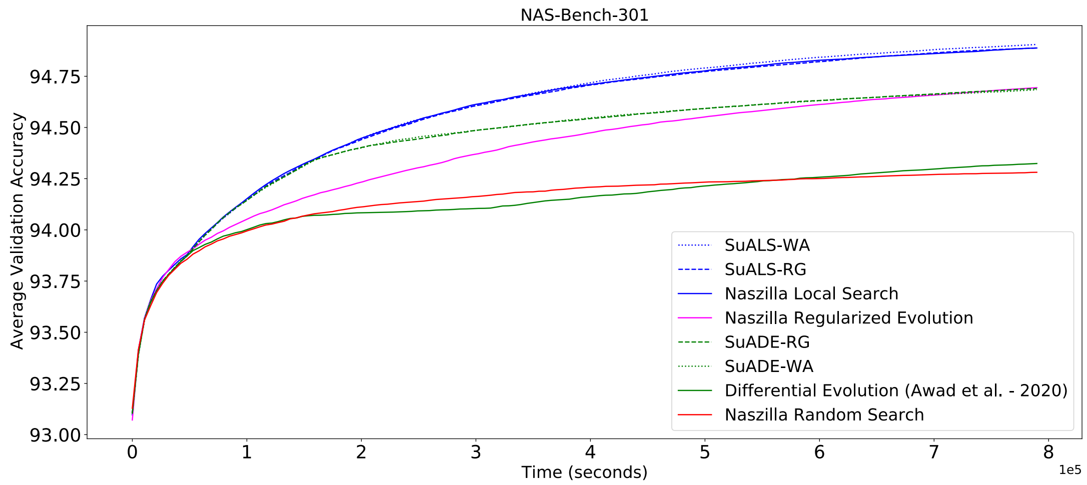


FIGURE 18. NAS-Bench-301 search performance.

the highest average validation accuracy. SuALS-WA outperforms Random Search, Differential Evolution, and Regularized Evolution in terms of average validation accuracy by 0.6, 0.56, and 0.19, respectively. There is an insignificant difference between the performances of SuALS-WA, SuALS-RG and Naszilla LS. The final average accuracy obtained by SuALS-WA and SuALS-RG is only 0.02 and 0.01 higher than that of Naszilla LS, respectively. As a comparison, we can note that SuADE-RG and SuADE-WA improved the performance of the baseline DE by 0.3647 and 0.3513, respectively. SuADE-RG performed slightly better than SuADE-WA but the difference is insignificant. These results confirm that the similarity-network-based estimators help Differential Evolution perform a faster and better exploration of the search space.

Another important observation is that all four algorithms developed here achieve the fastest improvement in accuracy, reaching approximately 94.35 in less than 1.6×10^5 seconds. Only Naszilla LS yields a similar performance among the other algorithms from the literature.

Even though SuALS-RG performed very well it is interesting to note that the correlations results for it (Section VI-A2) were really poor (close to zero or negative) at the early stages of the search and then significantly improved to be comparable with the other algorithms. It is plausible that the poor correlations at the early stages could have helped diversify the local search (as in simulated annealing when moves that worsen the fitness of solutions in the early stages are allowed).

VII. CONCLUDING REMARKS

In this research we have developed and tested two new neural network accuracy predictors that make use of a solution similarity network. We integrated these two predictors, namely weighted average and regression, into two search algorithms, differential evolution and local search, that are known to perform well for NAS. The resulting four search algorithms (SuADE-WA, SuADE-RG, SuALS-WA, SuALS-RG) have been tested on two benchmark datasets with significant characteristic differences in their search spaces. NAS-Bench-101 puts fewer restrictions on the design of the neural architecture, resulting in a very large search space, whereas NAS-Bench-301 with significant design restrictions has a much smaller search space. Thus, perhaps not surprisingly, for NAS-Bench-101 SuADE algorithms performed better than SuALS algorithms, as differential evolution is a population based algorithm and makes use of large moves while local search makes small local moves from a single incumbent solution. Consistent with these observations, for the NAS-Bench-301 dataset with more design restrictions, SuALS algorithms performed better than SuADE algorithms.

At the outset, the research question posed here was whether a relatively simple similarity-based solution network could work well as a prediction tool in NAS. There are some obvious advantages to such an approach. Firstly, it does not require a computationally expensive initialization stage and instead, training is integrated into the search process. Our correlation results showed that our algorithms obtain high correlations quickly and these correlation stay mostly stable. In other words, these prediction models are fast and start giving high quality predictions quickly. In addition to these correlation results, our computational experiments demonstrate that the prediction methods can improve the search performance significantly, as demonstrated for both NAS-Bench-101 and NAS-Bench-301 when used within differential evolution. Especially for NAS-Bench-101 with its large search space, and differential evolution which uses large moves, prediction quality is important for choosing the right moves.

The computational experiments have showed that the performances of the prediction models depend on the search algorithm within which they are implemented. For instance, they did not yield significant improvements when used within local search. Several design characteristics of local search could explain this. Firstly, since the moves consists of small mutations, the new solutions are close to the incumbent solution and not too different in terms of accuracy performance. Secondly, since the search is not population based, selection of next solution is done among similar solutions. Both of these reasons make prediction less critical than in differential evolution.

As future research, it would be interesting to see how well the weighted average approach performs when incorporated into other large step or population based search algorithms.

APPENDIX

NAS BEST PRACTICE CHECKLIST

We now describe how we addressed the individual points of the NAS best practice checklist [16].

1) **Best Practices for Releasing Code**

For all experiments we report:

a) Did we release code for the training pipeline used to evaluate the final architectures? Does not apply
b) Did we release code for the search space? Yes, for both NAS-Bench-101 and NAS-Bench-301 search spaces.
c) Did we release the hyperparameters used for the final evaluation pipeline, as well as random seeds? No, Does not apply
d) Did we release code for your NAS method? Yes
e) Did we release hyperparameters for your NAS method, as well as random seeds? Yes

2) **Best practices for comparing NAS methods**

a) For all NAS methods we compare, did we use exactly the same NAS benchmark, including the same dataset (with the same training-test split), search space and code for training the architectures and hyperparameters for that code? Yes
b) Did we control for confounding factors (different hardware, versions of DL libraries, different runtimes for the different methods)? No
c) Did we run ablation studies? Yes
d) Did we use the same evaluation protocol for the methods being compared? Yes
e) Did we compare performance over time? Yes
f) Did we compare to random search? Yes
g) Did we perform multiple runs of your experiments and report seeds? Yes
h) Did we use tabular or surrogate benchmarks for in-depth evaluations? We used both.

3) **Best practices for reporting important details**

a) Did we report how we tuned hyperparameters, and what time and resources this required? Yes
b) Did we report the time for the entire end-to-end NAS method (rather than, e.g., only for the search phase)? Yes
c) Did we report all the details of your experimental setup? Yes

REFERENCES

[1] N. Awad, N. Mallik, and F. Hutter, "Differential evolution for neural architecture search," 2020, *arXiv:2012.06400*.

[2] D. Baymurzina, E. Golikov, and M. Burtsev, "A review of neural architecture search," *Neurocomputing*, vol. 474, pp. 82–93, Feb. 2022.

[3] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 785–794.

[4] S. S. Choi, S. H. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," in *Proc. WMSCI 13th World Multi-Conf. Systemics, Cybern. Inform., Jointly With 15th Int. Conf. Inf. Syst. Anal. Synth., (ISAS)*, vol. 2009, pp. 80–85.

[5] M. Dehmer and K. Varmuza, "A comparative analysis of the Tanimoto index and graph edit distance for measuring the topological similarity of trees," *Appl. Math. Comput.*, vol. 259, pp. 242–250, May 2015.

[6] X. Dong and Y. Yang, "NAS-bench-201: Extending the scope of reproducible neural architecture search," 2020, *arXiv:2001.00326*.

[7] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, 1997–2017, 2019.

[8] F. Emmert-Streib, M. Dehmer, and Y. Shi, "Fifty years of graph matching, network alignment and network comparison," *Inf. Sci.*, vols. 346–347, pp. 180–197, Jun. 2016.

[9] X. He, K. Zhao, and X. Chu, "AutoML: A survey of the state-of-the-art," *Knowl.-Based Syst.*, vol. 212, Jan. 2021, Art. no. 106622.

[10] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated machine learning: methods, systems, challenges*. Cham, Switzerland: Springer, 2019.

[11] D. G. Jenkins and P. F. Quintana-Ascencio, "A solution to minimum sample size for regressions," *PLoS ONE*, vol. 15, no. 2, Feb. 2020, Art. no. e0229345.

[12] Y. Jin and B. Sendhoff, "Reducing fitness evaluations using clustering techniques and neural network ensembles," in *Proc. Genetic Evol. Comput. Conf.*, 2004, pp. 688–699.

[13] A. Klein and F. Hutter, "Tabular benchmarks for joint architecture and hyperparameter optimization," 2019, *arXiv:1905.04970*.

[14] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep. TR-2009, 2009.

[15] Z. Kuş, A. Gülcu, and C. Akkan, "Zekikus/novel-surrogate-measures-based-on-a-similarity-network-for-neural-architecture-search," Tech. Rep., Aug. 2022. [Online]. Available: https://github.com/zekikus/Novel-Surrogate-Measures-based-on-a-Similarity-Network-for-Neural-Architecture-Search

[16] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 9820–9837, 2020.

[17] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," 2018, *arXiv:1806.09055*.

[18] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *Proc. IEEE Conf. Comput. Vis. pattern Recognit.*, Jun. 2019, pp. 82–89.

[19] G. M. Maggiora and V. Shanmugasundaram, "Molecular Similarity Measures," in *Chemoinformatics and Computational Chemical Biology, Methods in Molecular Biology*, J. Bajorath, Ed. Cham, Switzerland: Springer, 2011, pp. 39–100.

[20] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: An overview, II," *WIREs Data Mining Knowl. Discovery*, vol. 7, no. 6, pp. 1–16, Nov. 2017.

[21] X. Ning, C. Tang, W. Li, Z. Zhou, S. Liang, H. Yang, and Y. Wang, "Evaluating efficient performance estimators of neural architectures," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 12265–12277.

[22] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*. Cham, Switzerland: Springer, 2006.

[23] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 4780–4789.

[24] B. Ru, L. Schut, D. W. M. van, C. Lyle, M. Fil, and Y. Gal, "Speedy performance estimation for neural architecture search," in *Proc. 35th Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2021, pp. 1–14.

[25] L. Schneider, F. Pfisterer, M. Binder, and B. Bischl, "Mutation is all you need," 2021, *arXiv:2107.07343*.

[26] A. Zela, J. Siems, L. Zimmer, J. Lukasik, M. Keuper, and F. Hutter, "Surrogate NAS benchmarks: Going beyond the limited search spaces of tabular NAS benchmarks," 2020, *arXiv:2008.09777*.

[27] R. Storn and K. Price, "Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces," *J. Global Optim.*, vol. 11, no. 4, pp. 341–359, 1997.

[28] H. Tong, C. Huang, L. L. Minku, and X. Yao, "Surrogate models in evolutionary single-objective optimization: A new taxonomy and experimental study," *Inf. Sci.*, vol. 562, pp. 414–437, Jul. 2021.

[29] K. Varmuza, W. Demuth, M. Karlovits, and H. Scsibrany, "Binary substructure descriptors for organic compounds," *Croatica Chemica Acta*, vol. 78, no. 2, pp. 141–149, 2005.

[30] C. Wei, C. Niu, Y. Tang, Y. Wang, H. Hu, and J. Liang, "NPENAS: Neural predictor guided evolution for neural architecture search," 2020, *arXiv:2003.12857*.

[31] W. Wen, H. Liu, Y. Chen, H. Li, G. Bender, and P.-J. Kindermans, "Neural predictor for neural architecture search," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2020, pp. 660–676.

[32] C. White, W. Neiswanger, S. Nolen, and Y. Savani, "A study on encodings for neural architecture search," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 20309–20319.

[33] C. White, W. Neiswanger, and Y. Savani, "BANANAS: Bayesian optimization with neural architectures for neural architecture search," 2019, *arXiv:1910.11858*.

[34] C. White, S. Nolen, and Y. Savani, "Exploring the loss landscape in neural architecture search," 2020, *arXiv:2005.02960*.

[35] C. White, A. Zela, B. Ru, Y. Liu, and F. Hutter, "How powerful are performance predictors in neural architecture search?" 2021, *arXiv:2104.01177*.

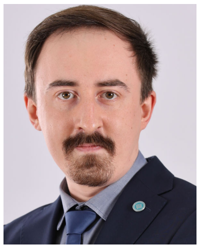
[36] J. Wu, X. Dai, D. Chen, Y. Chen, M. Liu, Y. Yu, Z. Wang, Z. Liu, M. Chen, and L. Yuan, "Stronger NAS with weaker predictors," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 28904–28918.

[37] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, no. 9, pp. 1423–1447, Sep. 1999.

[38] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "NAS-bench-101: Towards reproducible neural architecture search," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 7105–7114.

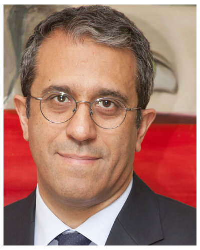
[39] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017, *arXiv:1611.01578v2*.

[40] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 8697–8710.



ZEKI KUŞ received the B.S. and M.S. degrees in computer engineering from Fatih Sultan Mehmet University, Istanbul, Turkey, in 2017 and 2019, respectively, where he is currently pursuing the Ph.D. degree in computer engineering.

He has been a Research Assistant with the Department of Computer Engineering, Fatih Sultan Mehmet University, since 2019. His research interests include meta-heuristics, deep neural networks, and image processing.



CAN AKKAN received the B.S. degree in industrial engineering from Middle East Technical University, Ankara, Turkey, in 1989, and the Ph.D. degree in operations research from Cornell University, Ithaca, NY, USA, in 1993.

In 2012, he was an International Faculty Fellow with the MIT Sloan School of Management. He is currently a Professor with the Sabancı Business School, Sabancı University, Istanbul, Turkey. Before joining Sabancı University, he was an Assistant Professor with the College of Administrative Sciences and Economics, Koç University. His current research interests include combinatorial optimization and heuristic search for computationally intensive problems.

Prof. Akkan is a member of INFORMS and the Turkish Operations Research Society (YAD). He was a recipient of the CEEMAN Teaching Champion Award, in 2015.

AYLA GÜLCÜ received the B.S. and M.S. degrees from the Department of Electronics and Computer Science, Marmara University, Istanbul, Turkey, in 2002 and 2006, respectively, and the Ph.D. degree in engineering management from Marmara University, in 2014.

She studied discrete optimization and meta-heuristics during her Ph.D. She has been teaching courses like data mining, algorithm design and analysis, and neural networks in addition to several programming courses. She is currently an Associate Professor with the Department of Software Engineering, Bahçeşehir University. Her research interests include meta-learning, deep learning, and reinforcement learning.