A TRANSFERABLE GENERAL-PURPOSE PREDICTOR FOR NEURAL ARCHITECTURE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Understanding and modelling the performance of neural architectures is key to Neural Architecture Search (NAS). Performance predictors for neural architectures are widely used in low-cost NAS and achieve high ranking correlations between predicted and ground truth performance in several search spaces. However, existing predictors are often designed based on network encodings specific to a predefined search space and are not generalizable across search spaces or to new families of architectures. In this work, we propose a general-purpose neural predictor for NAS that can transfer across architecture families, by representing any given candidate Convolutional Neural Network with a computation graph that consists of only primitive operators. Further combined with Contrastive Learning, we propose a semi-supervised graph representation learning procedure that is able to leverage both labelled accuracies and unlabeled information of architectures from multiple families to train universal embeddings of computation graphs and the performance predictor. Experiments conducted on three different NAS benchmarks, including NAS-Bench-101, NAS-Bench-201, and NAS-Bench-301, demonstrate that a predictor pre-trained on other families produces superior transferability when applied to a new family of architectures with a completely different design, after fine-tuning on a small amount of data. We then show that when the proposed transferable predictor is used in NAS, it achieves search results that are comparable to the state-of-the-arts on NAS-Bench-101 at a low evaluation cost.

1 INTRODUCTION

Neural architecture search (NAS) automates neural network design and has achieved remarkable performance on many computer vision tasks. A NAS strategy typically performs alternated search and evaluation over candidate networks to maximize a performance metric. While various strategies such as random search (Li & Talwalkar, 2019), differentiable search (Liu et al., 2018), Bayesian optimization (White et al., 2019), and reinforcement learning (Pham et al., 2018) can be used in the search step, fast and reliable evaluation is key to identifying better architectures at a low cost.

To avoid the excessive cost involved in the complete training and evaluating of each candidate network, most current NAS frameworks resort to performance estimation methods to predict its test accuracy. Frequently used performance estimation methods in the NAS field include partial training (Tan et al., 2018), weight sharing (Pham et al., 2018), neural predictors (Luo et al., 2018) and zero-cost proxies (Abdelfattah et al., 2021). The effectiveness of a performance estimation method is mainly determined by its inference time, i.e., the time it takes to generate an estimate, as well as the ranking correlation between predicted and ground-truth test scores. A higher ranking correlation could better guide NAS toward finding truly superior architectures. And lower inference time converts to more candidate networks visited under a computational budget.

While partial training and weight sharing are extensively used in early NAS works, thanks to several existing NAS benchmarks that provide an ample amount of labeled networks (Ying et al., 2019; Dong & Yang, 2020; Siems et al., 2020) (e.g., NAS-Bench-101 offers 423k networks trained on CIFAR-10), there have been many recent developments in training neural predictors for NAS (Wen et al., 2020; Tang et al., 2020). Without the need for any gradient computation, neural predictors often enjoy the advantage of the lowest inference time and the capability of continued improvement as more NAS benchmarking data are made available.

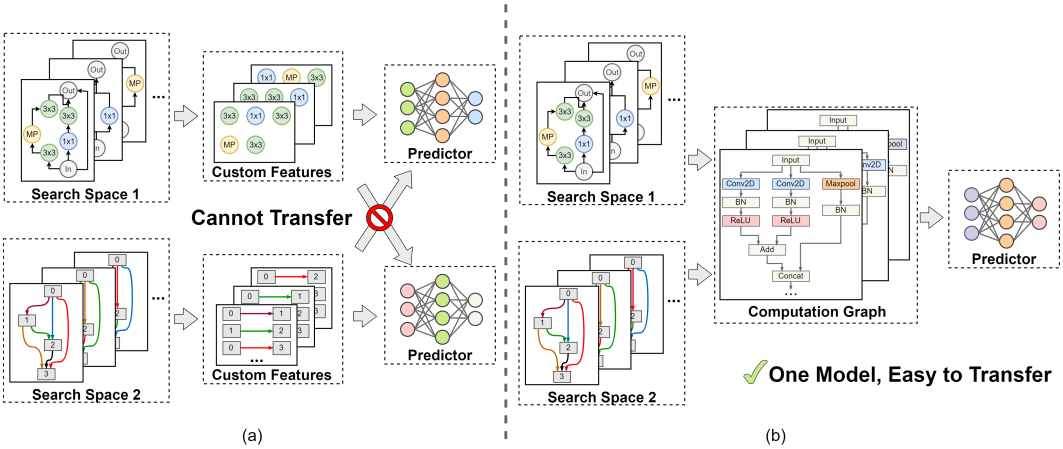

Figure 1: Comparison of predictor training setups: (a) conventional, space-dependent neural predictors; (b) the proposed general-purpose, space-agnostic predictor. Best viewed in color.

However, a major shortcoming of existing neural predictors is that they are not general-purpose. Each predictor is specialized to process networks confined to a specific search space. For example, NAS-Bench-101 limits its search space to a cell, which is a graph of up to 7 internal operators. Prior neural predictors designed for NAS-Bench-101 are unable to handle networks from other search spaces, e.g., DARTS (Liu et al., 2018), which adopts a different set of candidate operators and a different form of topology, in which each edge (instead of each node) must choose an operator. This search-space-specific design seriously limits the practicality and transferability of existing NAS predictors, since for any new search space that may be used in reality, a predictor must be re-designed and re-trained based on a large number of labeled networks in the new search space.

Another emerging approach to performance estimation is the zero-cost proxies (Abdelfattah et al., 2021), which can naturally support any network structure as input. However, the performance of zero-cost proxies may vary significantly depending on the search space. For example, Synflow achieves a high Spearman's ranking correlation coefficient of 0.74 when tested on the NAS-Bench-201 search space, whereas this correlation drops to 0.37 when applied to NAS-Bench-101 (Abdelfattah et al., 2021). Moreover, when compared to neural predictors, zero-cost proxies often have longer per-network inference time and require an additional batch of image data to make predictions.

In general, neural predictors offer better estimation quality and can be continuously improved, but do not transfer well across search spaces, while zero-cost proxies are naturally universal estimation methods, but could be sub-optimal on certain search spaces. In this paper, We propose a general-purpose predictor for NAS that is transferable across search spaces like the zero-cost proxies, yet still preserves the benefits of conventional predictors. Our contributions are summarized as follows:

- We propose the use of *computation graphs* to provide a universal representation of networks from different search spaces. Figure 1 highlights the differences between (a) existing NAS predictors and (b) the proposed framework. The key is to introduce a universal search space representation consisting of graphs of only primitive operators to model any network structure, such that a general-purpose predictor can be learned based on NAS benchmarks available from multiple search spaces and be transferred to a new search space.

- We combine recent advances in Graph Neural Networks (GNN) (Morris et al., 2019), self-attention mechanisms (Vaswani et al., 2017) and Contrastive Learning (CL) (Chen et al., 2020a) and propose a semi-supervised framework to learn a generalizable neural predictor. Inspired by CL, we introduce a graph representation learning procedure to learn a generalizable architecture encoder based on the structural information of vast unlabeled networks of a target search space. The embeddings obtained this way are then used to train a predictor based on labeled accuracies in source families to achieve transferability to the target search space.

- We perform experiments to show that our predictor could jointly leverage training data from multiple NAS-Bench families to boost performance, enabling the possibility of continuous improvement when more NAS benchmarking data become available. Experimental results on three representative NAS Benchmarks indicate that our predictor is a better option for transferable performance estimation than zero-cost proxies. With the proposed pre-training procedure and fine-tuning only on no more than 50 labeled architectures in the target search space, our predictor could achieve a Spearman's rank correlation coefficient of 0.917 and 0.892 on NAS-Bench-201 and NAS-Bench-301, respectively. When using the transferable predictor for search on NAS-Bench-101 (without further fine-tuning during search), we obtain results that are better than other non-transferable approaches like SemiNAS (Luo et al., 2020), NAO (Luo et al., 2018), and BANANAS (White et al., 2019).

2 Related works

Neural predictor is a popular choice of performance estimation in low-cost NAS. Existing predictor-based NAS works include NAO (Luo et al., 2018), which adopts an encoder-decoder setup for architecture encoding/generation, and a simple neural performance predictor that predicts based on the encoder outputs. SemiNAS (Luo et al., 2020) progressively updates an accuracy predictor during the search. BANANAS (White et al., 2019) relies on an ensemble of accuracy predictors as the inference model in its Bayesian Optimization process. Tang et al. (2020) constructs a similar auto-encoder-based predictor framework and supplies additional unlabeled networks to the encoder to achieve semi-supervised learning. Wen et al. (2020) proposes a sample-efficient search procedure with customized predictor designs for NAS-Bench-101 and ImageNet (Deng et al., 2009) search spaces. Our approach to constructing accuracy predictors is similar to arch2vec (Yan et al., 2020) in that the embedding is pre-trained. However, our embeddings are general, not restricted to the specific underlying search space. Furthermore, arch2vec focuses on solving the NAS problem directly. In contrast, our work focuses on learning generalizable accuracy predictors. NPENAS (Wei et al., 2020), BRP-NAS (Dudziak et al., 2020) are also notable predictor-based NAS approaches.

Zero-cost proxies are originally proposed as parameter saliency metrics in model pruning techniques. With the recent advances of pruning-at-initialization algorithms, only a single forward/backward propagation pass is needed to assess the saliency. Metrics used in these algorithms are becoming an emerging trend for transferable, low-cost performance estimation in NAS. Abdelfattah et al. (2021) transfers several zero-cost proxies to NAS, such as Synflow (Tanaka et al., 2020), Snip (Lee et al., 2018), Grasp (Wang et al., 2020) and Fisher (Turner et al., 2019). Zero-cost proxies could work on any search space and Abdelfattah et al. (2021) shows that on certain search spaces, they help to achieve results that are comparable to predictor-based NAS algorithms.

3 Computation Graph: a Unified Architecture Representation

Operator-grouping is an implicit but widely adopted technique for improving the efficiency of NAS. Without the loss of generality, we consider operations in Convolutional Neural Networks (CNN) for classification and define a *primitive* operator as an atomic node of computation in a CNN. In other words, a primitive operator is one like Convolution, ReLU, Add, Pooling or BatchNorm, which are considered to be a single point of execution that cannot be further reduced. Operator-grouping re-labels pre-defined combinations of primitive operators as new atomic groups. For example, a *Conv3x3* operator defined by the NAS-Bench-101 search space is a combination of 3 primitive operators: Convolution3x3-BatchNorm-ReLU. Different search spaces may propose different grouping strategies. E.g., in NAS-Bench-201, a grouped Conv3x3 operator is instead a combination of ReLU-Convolution3x3-BatchNorm. Operator-grouping simplifies the search space, helps to incorporate useful prior knowledge such as the frequent co-occurrence of Convolution, BatchNorm, ReLU in CNNs, and could lead to better search results. However, since the input features for neural predictors are commonly designed around the grouped operators, operator-grouping is also the main culprit of low transferability in existing predictors.

We define a computation graph as a detailed graphical representation of a neural network without any customized grouping, i.e., each node in the graph is a primitive operator and the network computation graph is made up of only such nodes and edges that direct the flow of information. Without

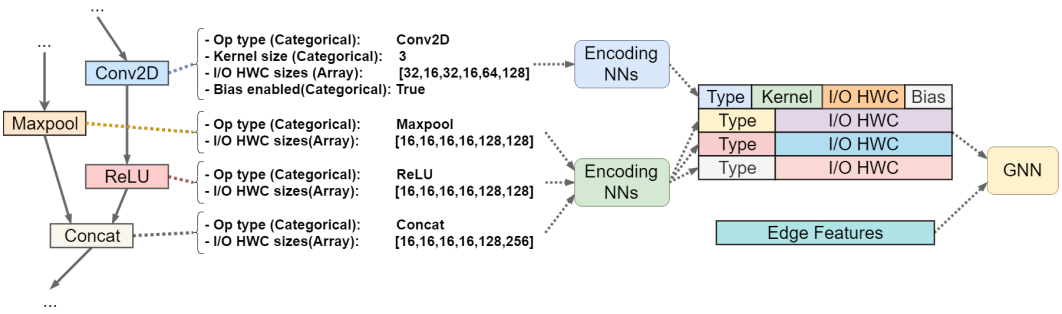

Figure 2: Example depicting the key graphical features extracted from compute graphs and how they are encoded as node features for the GNN.

operator-grouping, computation graphs define a search space that could represent any network structure since the number of primitive operators is usually far less than the number of possible groupings. We propose to learn a neural predictor based on features extracted from computation graphs, which makes our predictor a transferable performance estimation method.

While there are many potential ways to construct a computation graph, we adopt a simple approach of simplifying the model optimization graph maintained by a deep learning framework like PyTorch (Paszke et al., 2019) or TensorFlow (Abadi et al., 2015). As the name suggests, a model optimization graph is originally intended for gradient calculations and weight updates, and it is capable of supporting any network structure. In this work, all the computation graphs used in our experiments are simplified from TensorFlow model optimization graphs. Specifically, we extract the following nodes from a model optimization graph to form a computation graph:

- Nodes that refer to trainable neural network weights. For these nodes, we extract the operator type, such as Conv1D, Conv2D, Linear, etc, input/output channel sizes, input/output image height/width sizes, convolution kernel sizes (if available), and bias information as their node features.
- Nodes that refer to activation functions like ReLU or Sigmoid, pooling layers like MaxPool or AveragePool, as well as batch normalizations. For these nodes, we extract the operator type, input/output channel and image height/width sizes.
- Key supplementary nodes that indicate how information is processed, e.g., addition, multiplication and concatenation. For these nodes, we also extract the type, input/output channel and image height/width sizes.

Formally, a computation graph G consists of a vertex set $V = \{v_1, v_2, v_3, ...\}$ and an edge set E, where v refers to a primitive operator node and E contains pairs of vertices (v_s, v_d) indicating a connection between v_s and v_d. Under this definition, the problem of performance estimation becomes finding a function F such that for computation graph G_i, which is generated from a candidate neural network, $F(G_i) = P_i$, where P_i is the ground truth test score. One simple approach is to formulate F using Graph Neural Networks (GNNs). Figure 2 illustrates how we transform a computation graph into learnable feature vectors for a GNN. Representing networks as computation graphs enable us to break the barrier imposed by search space definitions and fully utilize all available data for predictor training, regardless of where a labeled network is from, e.g., NAS-Bench-101, 201 or 301. We could also effortlessly transfer a predictor trained on one search space to another, by simply fine-tuning it on additional data from the target space. In Section 4.1, we further leverage Contrastive Learning to extract additional knowledge from unlabeled computation graphs.

4 Neural Predictor via Graph Representation Learning

While features of computation graphs presented in Section 3 can naturally be fed into a GNN for accuracy prediction, we propose a two-stage approach to improve generalization and leverage unlabeled data via graph representation learning. We first find a vector representation (embedding)

which converts the graph features mentioned in Section 3 into a fixed-sized latent vector via a graph contrastive learning procedure, before feeding the latent vector to an MLP accuracy regressor. Given a target family for performance estimation, a salient advantage of our approach is its ability to leverage vast unlabeled data, e.g., computation graphs of the target family, which are typically available in abundance. In fact, our approach is able to jointly leverage labeled and unlabeled architectures from multiple search spaces to maximally utilize available information.

4.1 Computation Graph Representation using Contrastive Learning

For each computation graph, we would like to produce a vector representation in the sphere of $\mathbb{R}^m$ for a fixed hyper-parameter m. Similar to the word2vec representation of words (Mikolov et al., 2013), we would like to infer relationships between the networks by considering the angles between the corresponding vector representations. In this paper, we propose to learn representations of computation graphs using contrastive learning: only *similar* computation graphs will have close vector representations.

We follow the framework of SimCLR (Chen et al., 2020a;b), which was applied to image classification (see Fig. 4.1).

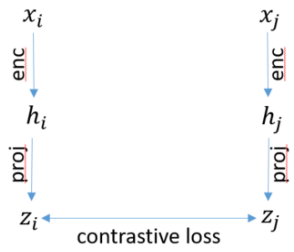

Figure 3: Contrastive loss

This framework learns a *base encoder*, that, given an image, outputs its vector representation. To train the base encoder, a *projection head* projects representations into a lower-dimensional space where a *contrastive loss function* is then optimized. The projection head, $Proj(*)$, is an MLP that maps vectors in $\mathbb{R}^m$ to vectors into the sphere of $\mathbb{R}^p$, where $p < m$. The contrastive loss is designed to force representations of *similar* images to be aligned. For images, similar images should belong to the same class. In addition to the contrastive loss function and the notion of similarity, *data augmentation* is also crucial for training the base encoder. The augmentation process occurs during training for each batch. For each batch element, a new data point is added to the batch before assembling the contrastive loss function. The new data point is just a slight random perturbation of the image. The idea here is that this pair, the image and its augmentation, constitute the only *positive pair*, that is, since the images are *similar* their corresponding vector representations should be aligned (all other pairs between the current batch element and other batch elements are *negative*).

When attempting to apply contrastive learning to computation graphs, we need to address the following challenging issues: 1) defining a sensible pairwise similarity function, 2) designing an appropriate contrastive loss function, 3) implementing a suitable form of data augmentation, and finally, 4) choosing an architecture for the base encoder.

Since our focus is on the accuracy of neural networks, our vector representation will be tilted to encode features that are useful for clustering networks based on accuracy, although our approach could also be used for other clustering criteria (e.g., latency, network family affiliation, topological structure, etc.). We start by considering the similarity between computation graphs. If network accuracies are known, we can define the similarity of two computation graphs as their absolute difference in accuracy, that is, irrespective of architectural differences, two computation graphs will be close in vector representation if their accuracies achieved on some benchmark, e.g., ImageNet test accurarcies, are close.

Alternatively, we can define similarity based on network structures, since it is intuitively expected that networks that, as graphs, are topological close should have also comparable performance. This approach is particularly useful when a sizeable number of unlabeled computation graphs are readily available for training. In what follows, we describe our approach that uses spectral properties of undirected graphs to compare computation graphs (see, Dwivedi & Bresson (2021), Dwivedi et al. (2020)). Given a computation graph, we consider its underlying undirected graph G. Let A be its

adjacency matrix, D be its degree diagonal matrix. The normalized Laplacian matrix is defined as

$$\Delta = I - D^{-1/2}AD^{-1/2} = U^T \Lambda U,$$

where Λ is the diagonal matrix of eigenvalues and U the matrix of eigenvectors. It is known that the eigenvalues Λ encode important connectivity features of G. For instance, 0 is the smallest eigenvalue and has multiplicity 1 iff G is connected. More generally, the eigenvalues of Δ can be used to measure the *distance* between graphs. We do this by first truncating the eigenvalues by considering only the k smallest eigenvalues of Δ, where k is usually much smaller than the number of nodes of any computation graph of interest. For our experiments we chose $k = 11$. Given two computation graphs, we define their distance as the Euclidean norm of the corresponding k eigenvalues sorted from smallest to largest (note that this is not a true metric between graphs). Smaller eigenvalues focus on general features of the graph, whereas larger eigenvalues focus on features at higher granularity (see Wills & Meyer (2019) for more details). We use $\sigma_S(g_1, g_2)$ to denote this spectral distance between computation graphs g_1, g_2.

Next, we address the contrastive loss function. A key difference between images and computation graphs is that our task of interest is regression rather than classification. We address this issue by replacing a single positive pair with a probability distribution over all pairs which smoothly favors similar computation graphs. Consider a batch of N computation graphs, whose vector representation is $I = \{h_1, h_2, \ldots, h_N\} \subset \mathbb{R}^m$. For each i, let $z_i = Proj(h_i) \in \{||z|| = 1 : z \in \mathbb{R}^p\}$, and let the cosine similarity be $sim(z_i, z_j) = z_i \cdot z_j/\tau$, where the temperature $\tau > 0$ is a hyper-parameter ($\tau = 0.1$ in all of our experiments) and $\cdot$ is the dot product. In SimCLR (Chen et al., 2020a), for each i in the batch, called *anchor* index, there is an associated element $j(i)$ (in an augmented batch), called *positive* index, while $r \neq i, j(i)$ are *negative* indices. The contrastive loss maximizes the probability of the positive pair $(i, j(i))$:

$$\mathcal{L}_{SimCLR} = -\sum_{i \in I} \log \frac{\exp(sim(z_i, z_j(i)))}{\sum_{r \neq i} \exp(sim(z_i, z_r))}.$$

For images, $j(i)$ is a small perturbation of image i. If classes are known, the SupCon loss function (Khosla et al., 2020) is

$$\mathcal{L}_{SupCon} = \sum_{i \in I} \frac{-1}{P(i)} \sum_{s \in P(i)} \log \frac{\exp(sim(z_i, z_s))}{\sum_{r \neq i} \exp(sim(z_i, z_r))}. \tag{1}$$

where $P(i)$ are the non-anchor indices whose class is the same as i. Thus, not just $j(i)$, but all indices whose class is the same as i contribute to the probability of positive pairs, not just $j(i)$.

Our contrastive loss will take elements from both $\mathcal{L}_{SimCLR}$ and $\mathcal{L}_{SupCon}$. Consider next the case in which the batch consists of computation graphs. First, if the family of networks each computation graph belongs to is known, e.g., NB201, NB101, OFA, etc., we treat the family affiliation as a class and follow the $\mathcal{L}_{SupCon}$ approach. Second, rather than using the uniform distribution over $P(i)$ as in (1), we use a convex combination over $P(i)$ that will be based on the similarity of the corresponding computation graphs

$$\mathcal{L} = -\sum_{i \in I} \sum_{s \in P(i)} \alpha_s^{(i)} \log \frac{\exp(sim(z_i, z_s))}{\sum_{r \neq i} \exp(sim(z_i, z_r))}, \tag{2}$$

where $\alpha_s^{(i)} \geq 0$ and $\sum_{s \in P(i)} \alpha_s^{(i)} = 1$. For computation graph i, we simply define $\alpha_*^{(i)}$ to be the softmax of $\sigma_S(i, *)$ (we used a temperature of 0.05 in all of our experiments).

The challenge of data augmentation for computation graphs is that slightly perturbing a computation graph may drastically change its accuracy on a benchmark, for instance, when changing an activation function. In addition, arbitrary small changes to a computation graph may make it fall outside of the family of networks of interest. To address this issue, rather than randomly perturbing a computation graph, we randomly pick a very similar graph (e.g., in σ_S sense) from the training set to form a positive pair. As suggested in (Khosla et al., 2020), we learn the embeddings using large batch sizes.

Finally, we describe our encoder architecture to generate embeddings. The simplest encoder is just a GNN. Since predicting accuracy involves both local and global graph features, we also propose

alternative encoders that usually perform better. In particular, we consider the addition of attention mechanisms (as in Vaswani et al. (2017)) to capture global graph relationships that are harder to explore using regular GNNs. More precisely, we use transformer encoders (Vaswani et al., 2017) with and without positional embeddings (as described in Chen et al. (2020a;b)) by itself or combined with GNN encoders using composition, concatenation or residual connections between the two. In practice, our best encoders concatenate a GNN encoder and a transformer encoder or concatenate a GNN encoder with the node embedding of Section 3 followed by a transformer encoder. Finally, as suggested in (Khosla et al., 2020), we normalize the output representations.

4.2 AN MLP ACCURACY PREDICTOR BASED ON VECTOR REPRESENTATIONS

Given a training set of accuracy-labeled computation graphs, we first find the embeddings as described in Section 4.1. Notice that the representations are in $\mathbb{R}^m$ for hyper-parameter m. Our MLP is just a regressor that predicts accuracies given m-sized latent features. In practice, we proceed as follows. First, we train several vector representations over a small set of hyper-parameters, this is, in fact, the bulk of the running time. For each one of these, we train our MLP regressor. For each one of these predictors, we calculate the Spearman correlation with respect to each of the training families we used for training and select the configuration with the highest value. We empirically observed that our model selection approach is very stable, that is, the selected model does not change over different MLP random runs.

So far, we have assumed that no accuracy data is available from the target family. To eliminate the bias introduced by training on different families, we use *fine tuning*. We fine-tune our MLP using a very small fraction of labeled data from the target family. This step seems inexpensive and necessary for transferability. This approach is used, for instance, to evaluate the quality of embeddings in contrastive learning (Chen et al., 2020a). In practice, the tiny amount of labeled data used for fine-tuning can be added to the number of evaluations needed during NAS, minimally affecting performance.

5 EXPERIMENTATION

In this section, we evaluate our proposed transferable predictor and demonstrate its superiority over existing alternatives. We begin by comparing the ranking correlations between our predictor and other transferable zero-cost proxies on popular NAS benchmarks. We then show that methods with higher ranking correlations often produce better search results in Section 5.2.

5.1 COMPARISON OF RANKING CORRELATIONS

In this experiment, we study how the performance estimation quality of our transferable predictor compares to other transferable zero-cost proxy baselines. We consider the search spaces of NAS-Bench-101 (Ying et al., 2019), NAS-Bench-201-CIFAR-10 (Dong & Yang, 2020) and NAS-Bench-301 (Siems et al., 2020), which cover a wide range of classification network architectures, e.g., ResNet, Inception, DARTS. We sample 5000 instances from NAS-Bench-101, 4096 instances from NAS-Bench-201 and 1000 instances from NAS-Bench-301 as independent test sets. We execute the zero-cost proxies directly on these test sets and report the Spearman correlation coefficient ($\rho \in [-1, 1]$), with values closer to 1 indicating higher ranking correlations between the predicted performance and ground truth test accuracies.

For our predictor we differentiate between *target* and *source* families. The test set belongs to the target family. Furthermore, we assume that for the target family we have a large amount of unlabeled data and a very small amount of labeled data. Source families are fully labeled (i.e., accuracies are known). We consider all the cases in which one of NAS-Bench-101, NAS-Bench-201, NAS-Bench-301 is the target family, while the reminder two are source families. Our predictor uses the structural information of the target family for training the embeddings (Section 4.1), the source families for training the MLP (Section 4.2) and the small amount labeled from the target family to fine-tune the MLP (also Section 4.2). We use all the unlabeled data of the target family and all labeled data from the source families. And, specifically for fine-tuning, for target families NAS-Bench-101 and NAS-Bench-301 we sample 50 labeled networks, while for NAS-Bench-201 we sample 40.

Table 1: Spearman correlation coefficients (ρ). Average of 5 fine tuning runs.

Method	NAS-Bench-201	NAS-Bench-101	NAS-Bench-301
Synflow (Tanaka et al., 2020)	0.823	0.361	-0.210
Jacov (Mellor et al., 2021)	0.859	0.358	-0.190
Fisher (Turner et al., 2019)	0.687	-0.277	-0.305
GradNorm (Abdelfattah et al., 2021)	0.714	-0.256	-0.339
Grasp (Wang et al., 2020)	0.637	0.245	-0.055
Snip (Lee et al., 2018)	0.718	-0.165	-0.336
GNN-fine-tune	0.884 ± 0.03	0.542 ± 0.14	0.872 ± 0.01
CL-fine-tune	$\mathbf{0.917} \pm 0.01$	$\mathbf{0.553} \pm 0.09$	$\mathbf{0.892} \pm 0.01$

We consider the following variants:

- **CL-fine-tune** is our best predictor model that leverages CL-based pre-training (Section 4.1) as described above.
- **GNN-fine-tune** serves as the ablation comparison model to verify the usefulness of Contrastive Learning. Here, we directly pre-train + fine-tune a k-GNN (Morris et al., 2019) predictor on the same amount of labeled data as CL-fine-tune.

Table 1 summarizes the results. We first note that although zero-cost proxies can work on any search space, their prediction quality is often unstable. While most proxies generate positive correlations on NAS-Bench-201, the correlation suffers a significant drop when transferred to NAS-Bench-101 and 301, which is similar to the observation made by Abdelfattah et al. (2021). Since there is no easy way to correct or improve poorly-performing zero-cost proxies, their transferability and generalizability remain questionable. In comparison, our predictor learns generalizable features from the computation graphs, which makes it capable of utilizing any existing labeled/unlabeled data. And with a small amount of fine-tuning data, our predictor transfers well to all test search spaces with correlations far exceeding zero-cost proxy baselines. Finally, we observe that the CL-fine-tune variant produces higher correlations than the GNN-fine-tune variant and is more stable across different runs, which demonstrates the usefulness of CL pre-training using unlabeled data.

5.2 Search results

We now demonstrate that our predictor is the better option for transferable performance estimation in NAS. To highlight the impact of different estimation methods, we adopt a common evolutionary search procedure described in Algorithm 1. We vary the choice of M and report the accuracy and rank of the best architecture found. Specifically, we compare the CL-fine-tune predictor against the Synflow zero-cost proxy and a random estimation baseline. For each method, we conduct 5 search runs on NAS-Bench-101 which has 423,624 labeled candidates, also on NAS-Bench-201 with 15,625 searchable candidates, and NAS-Bench-301 with over 10^{18} candidates.

Algorithm 1 Our EA Search Algorithm (More details in Appendix)

1: **Input:** A set of random architectures with evaluated performances P; A performance estimator M; Budget B; NAS-Benchmark D;
2: **for** $t = 1, 2, \ldots, T$ **do**
3: Select top-k best architectures from P, denote as P_{best}.
4: Perform crossover + mutation on P_{best} to create a new candidate pool P_{new}.
5: Rank candidates in P_{new} according to the estimated performance from M.
6: Use D to query the ground truth scores of the first B candidates of P_{new}, denote as P_{child}.
7: $P \leftarrow P_{child} + P$.
8: **end for**

To establish a fair comparison, we ensure that the same number of networks are queried (#Q) using D in every run by adjusting B, T, as well as the initial size of P. Intuitively, the number of queries made to the benchmark simulates the real-world computational cost of NAS. Table 2 reports the search results. We observe that on NAS-Bench-101, searching with CL-fine-tune produces better

results than Synflow, which is most likely due to the difference in their ranking correlations (0.553 vs. 0.361 as reported in Table 1). On NAS-Bench-201, Since Synflow and CL-fine-tune both achieve high correlations (0.823 vs. 0.917), they could find the best architecture among the entire search space in every run, which is significantly better than the random baseline. In Table 1, we observe that Synflow produces a negative correlation on the NAS-Bench-301 test set, which means that it often ranks bad-performing architectures higher the good ones. We see in Table 2 that this is detrimental to the search since Synflow is unable to outperform the random baseline. In comparison, with a small amount of fine-tuning data, the CL-fine-tune predictor achieves exceptional transferability and is able to better guide the search process.

Table 2: Search results on NAS-Bench-101, 201 and 301 using the same EA search algorithm but with different performance estimation methods. #Q represents the number of unique networks queried during search. Note that the #Q for CL-fine-tune also counts the fine-tuning instances.

Method	**NAS-Bench-101**			**NAS-Bench-201**			**NAS-Bench-301**	
	#Q	Acc. (%)	Rank	#Q	Acc. (%)	Rank	#Q	Acc. (%)
Random	700	94.11 ± 0.10	26.0	90	93.91 ± 0.2	104	800	94.75 ± 0.08
Synflow	700	94.18 ± 0.05	5.8	90	**94.37** ± 0.0	**1.0**	800	94.60 ± 0.11
CL-fine-tune	700	**94.23** ± 0.01	**2.2**	90	**94.37** ± 0.0	**1.0**	800	**94.83** ± 0.06

Last but not least, we compare our best search results on NAS-Bench-101 to other state-of-art NAS approaches in Table 3. We observe that our EA + CL-fine-tune setup is competitive among other NAS algorithms. For instance, our setup requires fewer queries to find the second-best architecture (94.23) in NAS-Bench-101 compared to BANANAS. And our search result is better than SemiNAS, NAO and RE in terms of the number of queries and accuracy. More importantly, our predictor could transfer to other search spaces with only a small amount of labeled data, which is a unique advantage compared to other non-transferable predictors used in algorithms like Neural-Predictor-NAS.

Table 3: Comparison against other NAS approaches on NAS-Bench-101.

NAS algorithm	**#Queries**	**Best Acc. (%)**
Random Search	2000	93.66
RE (Real et al., 2019)	2000	93.97
SemiNAS (Luo et al., 2020)	2000	94.02
SemiNAS (RE) (Luo et al., 2020)	2000	94.03
SemiNAS (RE) (Luo et al., 2020)	1000	93.97
NAO (Luo et al., 2018)	2000	93.90
BANANAS (White et al., 2019)	800	**94.23**
GA-NAS (Rezaei et al., 2021)	378	**94.23**
Neural-Predictor-NAS (Wen et al., 2020)	256	94.17
NPENAS (Wei et al., 2020)	150	94.14
BRP-NAS (Dudziak et al., 2020)	140	94.22
EA + CL-fine-tune	700	**94.23**

6 CONCLUSION

In this work, we propose the use of computation graphs as a universal representation for any CNN network structures. On top of this representation, we design a novel, performant, transferable neural predictor that incorporates Graph Convolutional Networks, Self-Attention and Contrastive Learning. Experimental results suggest that our predictor could transfer to any search space and achieve superior prediction quality, with a Spearman correlation coefficient over 0.8, while only requiring a small amount of labeled data. When used in NAS, our predictor helps to generate results that are competitive among other non-transferable state-of-the-art methods. In general, our transferable predictor alleviates the need to manually design and re-train new performance predictors for any new search spaces in the future, which helps to further reduce the computational cost and carbon footprint of NAS.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*, 2021.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv preprint arXiv:2002.05709*, 2020a.

Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. Big self-supervised models are strong semi-supervised learners. *arXiv preprint arXiv:2006.10029*, 2020b.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. 2020.

Łukasz Dudziak, Thomas Chau, Mohamed S Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D Lane. Brp-nas: Prediction-based nas using gcns. *arXiv preprint arXiv:2007.08668*, 2020.

Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs, 2021.

Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks, 2020.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *CoRR*, abs/2004.11362, 2020. URL `https://arxiv.org/abs/2004.11362`.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638, 2019. URL `http://arxiv.org/abs/1902.07638`.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. 2018. URL `http://arxiv.org/abs/1806.09055`.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pp. 7816–7827, 2018.

Renqian Luo, Xu Tan, Rui Wang, Tao Qin, Enhong Chen, and Tie-Yan Liu. Semi-supervised neural architecture search. *arXiv preprint arXiv:2002.10389*, 2020.

Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pp. 7588–7598. PMLR, 2021.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pp. 3111–3119. Curran Associates, Inc., 2013.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4602–4609, 2019.

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

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018. URL `http://arxiv.org/abs/1802.03268`.

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

Seyed Saeed Changiz Rezaei, Fred X Han, Di Niu, Mohammad Salameh, Keith Mills, Shuo Lian, Wei Lu, and Shangling Jui. Generative adversarial neural architecture search. *arXiv preprint arXiv:2105.09356*, 2021.

Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777*, 2020.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile, 2018.

Hidenori Tanaka, Daniel Kunin, Daniel LK Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *arXiv preprint arXiv:2006.05467*, 2020.

Yehui Tang, Yunhe Wang, Yixing Xu, Hanting Chen, Boxin Shi, Chao Xu, Chunjing Xu, Qi Tian, and Chang Xu. A semi-supervised assessor of neural architectures. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1810–1819, 2020.

Jack Turner, Elliot J Crowley, Michael O'Boyle, Amos Storkey, and Gavin Gray. Blockswap: Fisher-guided block substitution for network compression on a budget. *arXiv preprint arXiv:1906.04113*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*, 2020.

Chen Wei, Chuang Niu, Yiping Tang, Yue Wang, Haihong Hu, and Jimin Liang. Npenas: Neural predictor guided evolution for neural architecture search. *arXiv preprint arXiv:2003.12857*, 2020.

Wei Wen, Hanxiao Liu, Yiran Chen, Hai Li, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. In *European Conference on Computer Vision*, pp. 660–676. Springer, 2020.

Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. *arXiv preprint arXiv:1910.11858*, 2019.

Peter Wills and Francois G. Meyer. Metrics for graph comparison: A practitioner's guide, 2019.

Shen Yan, Yu Zheng, Wei Ao, Xiao Zeng, and Mi Zhang. Does unsupervised architecture representation learning help neural architecture search?, 2020.

Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. 2019.

7 APPENDIX

7.1 MORE ON OUR EA SEARCH ALGORITHM

In our EA algorithm, we propose a combination of crossover and mutation procedures to create a larger pool of mutated child architectures. Although the detailed implementation of these two procedures might differ depending on the underlying search space, we still provide a high-level description for them here.

In the *crossover* procedure, we randomly select two unique architectures from the current top-k best set of architectures, namely, $parent_1$ and $parent_2$. During crossover, we randomly select one operator in $parent_2$ and use it to replace another random operator in $parent_1$. The replaced operator cannot be the same as the selected operator. Therefore, our crossover is single-point and uniformly random. After crossover, we perform additional mutations on the child architecture.

In the *mutation* procedure, given an architecture that is randomly selected from the top-k set or is coming from the crossover procedure, we perform 1-edit random mutations on its internal structures. The actual definition for 1-edit change is determined by the search space. For example, on NAS-Bench-101, 1-edit mutations include the following:

- Swap an existing operator in the cell with another uniformly sampled operator.
- Add a new operator to the cell with random connections to other operators.
- Remove an existing operator in the cell and all of its incoming/outgoing edges.
- Add a new edge between two existing operators in the cell.
- Remove an existing edge from the cell.

A 1-edit mutation means we choose one mutation type from the list above and execute it on the given architecture. Intuitively, under different search space definitions, this list of valid mutations could be different. It is also worth mentioning that in our search we allow for more than 1-edit mutations, i.e., we could randomly perform consecutive mutations on an architecture to boost the diversity of the new population.

7.2 KEY HYPER-PARAMETERS FOR NAS

Table 4 reports some of the key hyper-parameters we used to generate the search results. For some of the search spaces, we had to choose slightly different values for CL-fine-tune to account for the extra instances needed for fine-tuning and to make sure each search run issues the same amount of queries to the NAS benchmark.

Table 4: Summary of key hyper-parameters. $|P_{init}|$ refers to the size of the starting population.

Method	**NAS-Bench-101**				**NAS-Bench-201**				**NAS-Bench-301**			
	k	B	$\|P_{init}\|$	T	k	B	$\|P_{init}\|$	T	k	B	$\|P_{init}\|$	T
Random	20	100	100	6	10	20	10	4	20	100	100	7
Synflow	20	100	100	6	10	20	10	4	20	100	100	7
CL-fine-tune	20	100	50	6	10	10	10	5	20	100	50	7

7.3 PREDICTOR DETAILS AND HYPER-PARAMS PARAMETERS

7.3.1 VECTOR REPRESENTATION

We describe the encoder of Section 4.1 in more detail.

First, we use a *node embedding* that process a computation graph representation and outputs a graph with 64 node features.

The *GNN encoder* consists of 4 or 6 GNN layer, with node features remaining at 64. The aggregator function is the mean, so the output is of size 64.

We use a standard *transformer encoder* with up to 2 layers and 2 attention heads. The input and output size are the same and are either 64 or 128.

The *projection* is a 4 layer MLP with input 128, layer size 64 and activation ReLu.

We construct the CL encoder in two different ways:

1. Apply the node embedding, then we apply in parallel the GNN encoder and the transformer encoder, concatenate the two in the end.
2. Apply node embedding, then the GNN encoder. Concatenate the node embedding and the GNN encoder results. Apply the transformer encoder.

7.3.2 MLP

The MLP of Section 4.2 consists of 5 layers. The input size is $m = 128$, the size of the vector representation. The rest of the layers are of size 200, except the last one which is of size 1. Between each of the dense layers the activation is ReLU.