CATE: Computation-aware Neural Architecture Encoding with Transformers

Shen Yan [1] **Kaiqiang Song** [2 3] **Fei Liu** [2] **Mi Zhang** [1]

Abstract

Recent works (White et al., 2020a; Yan et al., 2020) demonstrate the importance of architecture encodings in Neural Architecture Search (NAS). These encodings encode either structure or computation information of the neural architectures. Compared to structure-aware encodings, computation-aware encodings map architectures with similar accuracies to the same region, which improves the downstream architecture search performance (Zhang et al., 2019; White et al., 2020a). In this work, we introduce a **C**omputation-**A**ware **T**ransformer-based **E**ncoding method called CATE. Different from existing computation-aware encodings based on fixed transformation (e.g. path encoding), CATE employs a pairwise pre-training scheme to learn computation-aware encodings using Transformers with cross-attention. Such learned encodings contain dense and contextualized computation information of neural architectures. We compare CATE with eleven encodings under three major encoding-dependent NAS subroutines in both small and large search spaces. Our experiments show that CATE is beneficial to the downstream search, especially in the large search space. Moreover, the outside search space experiment demonstrates its superior generalization ability beyond the search space on which it was trained. Our code is available at: https://github.com/MSU-MLSys-Lab/CATE.

1. Introduction

Neural Architecture Search (NAS) has recently drawn considerable attention (Elsken et al., 2019). While majority of the prior work focuses on either constructing new search spaces (Liu et al., 2018b; Radosavovic et al., 2020; Ru et al., 2020) or designing efficient architecture search and evaluation methods (Luo et al., 2018b; Shi et al., 2020; White et al., 2021), some of the most recent work (White et al., 2020a; Yan et al., 2020) sheds light on the importance of *architecture encoding* on the subroutines in the NAS pipeline as well as on the overall performance of NAS.

While existing NAS methods use diverse architecture encoders such as LSTM (Zoph et al., 2018; Luo et al., 2018b), SRM (Baker et al., 2018), MLP (Liu et al., 2018a; Wang et al., 2020), GNN (Wen et al., 2020; Shi et al., 2020; Yan et al., 2020) or adjacency matrix itself (Kandasamy et al., 2018; Real et al., 2019; White et al., 2020b), these encoders encode either *structures* (Luo et al., 2018b; Ying et al., 2019; Wang et al., 2020; Wen et al., 2020; Shi et al., 2020; Yan et al., 2020) or *computations* (Zhang et al., 2019; Ning et al., 2020b; White et al., 2021) of the neural architectures. Compared to structure-aware encodings, computation-aware encodings are able to map architectures with different structures but similar accuracies to the same region. This advantage contributes to a smooth encoding space with respect to the *actual* architecture performance instead of structures, which improves the efficiency of the downstream architecture search (Zhang et al., 2019; 2020; White et al., 2020a).

We argue that current architecture encoders limit the power of computation-aware architecture encoding for NAS. The major limitations lie in their representation power and the effectiveness of their pre-training objectives. Specifically, (Zhang et al., 2019) uses shallow GRUs to encode computation, which is not sufficient to capture deep contextualized computation information. Moreover, their decoder is trained with the reconstruction loss via asynchronous message passing. This is very challenging in practice because directly learning the generative model based on a single architecture is not trivial. As a result, its pre-training is less effective and the downstream NAS performance is not as competitive as state-of-the-art structure-aware encoding methods. (White et al., 2020a) proposes a computation-aware encoding method based on a fixed transformation called path encoding, which shows outstanding performance under the predictor-based NAS subroutine. However, path encoding scales exponentially without truncation and it inevitably causes information loss with truncation. Moreover, path encoding exhibits worse generalization performance in outside search space compared to the adjacency matrix encoding

[1]Michigan State University [2]University of Central Florida [3]Tencent AI Lab. Correspondence to: Shen Yan <yanshen6@msu.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

since it could not generalize to unseen paths that are not included in the training search space.

In this work, we propose a new computation-aware neural architecture encoding method named CATE (**C**omputation-**A**ware **T**ransformer-based **E**ncoding) that alleviates the limitations of existing computation-aware encoding methods. As shown in Figure 1, CATE takes paired computationally similar architectures as its input. Similar to BERT, CATE trains the Transformer-based model (Vaswani et al., 2017) using the masked language modeling (MLM) objective (Devlin et al., 2019). Each input architecture pair is corrupted by replacing a fraction of their operators with a special mask token. The model is trained to predict those masked operators from the corrupted architecture pair.

CATE differs from BERT (Devlin et al., 2019) in two aspects. First, each prediction in LMs has its inductive bias given the contextual information from different positions. This, however, is not the case in architecture representation learning since the prediction distribution is uniform for any valid graph, making it difficult to directly learn the generative model from a single architecture. Therefore, we propose a pairwise pre-training scheme that encodes computationally similar architecture pairs through two Transformers with shared parameters. The two individual encodings are then concatenated, and the concatenated encoding is fed into another Transformer with a cross-attention encoder to encode the joint information of the architecture pair. Second, the fully-visible attention mask (Raffel et al., 2020) could not be used for architecture representation learning because it does not reflect the single-directional flow (*e.g.* directed, acyclic, single-in-single-out) of neural architectures (Xie et al., 2019a; You et al., 2020a). Therefore, instead of using a bidirectional Transformer encoder as in BERT, we directly use the adjacency matrix to compute the causal mask (Raffel et al., 2020). The adjacency matrix is further augmented with the Floyd algorithm (Floyd, 1962) to encode the long-range dependency of different operations. Together with the MLM objective, CATE is able to encode the computation of architectures and learn dense and deep contextualized architecture representations that contain both local and global computation information in neural architectures. This is important for architecture encodings to be generalized to outside search space beyond the training search space.

We compare CATE with eleven structure-aware and computation-aware architecture encoding methods under three major encoding-dependent subroutines as well as eight NAS algorithms on NAS-Bench-101 (Ying et al., 2019) (small), NAS-Bench-301 (Siems et al., 2020) (large), and an outside search space (White et al., 2020a) to evaluate the effectiveness, scalability, and generalization ability of CATE. Our results show that CATE is beneficial to the downstream architecture search, especially in the large search space. Specifically, we found the strongest NAS performance in all search spaces using CATE with a Bayesian optimization-based predictor subroutine together with a novel computation-aware search. Moreover, the outside search space experiment shows its superior generalization capability beyond the search space on which it was trained. Finally, our ablation studies show that the quality of CATE encodings and downstream NAS performance are non-decreasingly improved with more training architecture pairs, more cross-attention Transformer blocks and larger dimension of the feed-forward layer.

2. Related Work

Neural Architecture Search (NAS). NAS has been started with genetic algorithms (Miller et al., 1989; Kitano, 1990; Stanley & Miikkulainen, 2002) and recently becomes popular when (Zoph & Le, 2017; Baker et al., 2017) gain significant attention. Since then, various NAS methods have been explored including sampling-based and gradient-based methods. Representative sampling-based methods include random search (Li & Talwalkar, 2019), evolutionary algorithms (Real et al., 2019; Lu et al., 2020), local search (Ottelander et al., 2020; White et al., 2020b), reinforcement learning (Zoph et al., 2018; Tan et al., 2019), Bayesian optimization (Kandasamy et al., 2018; Zhou et al., 2019), Monte Carlo tree search (Negrinho & Gordon, 2017; Wang et al., 2020) and Neural predictor (Baker et al., 2018; Liu et al., 2018a; Wen et al., 2020; Tang et al., 2020; Ning et al., 2020a; Luo et al., 2020; Shi et al., 2020; Yan et al., 2020; White et al., 2021; Ru et al., 2021). Weight-sharing methods (Bender et al., 2018; Pham et al., 2018) have become popular due to their computation efficiency. Based on weight-sharing, gradient-based methods are proposed to optimize the architecture selection with gradient decent (Luo et al., 2018b; Liu et al., 2019a; Xie et al., 2019b; Dong & Yang, 2019; Yan et al., 2019; You et al., 2020b; Peng et al., 2020; Zela et al., 2020; Chen & Hsieh, 2020). For comprehensive surveys, we suggest referring to (Elsken et al., 2019; Xie et al., 2020).

Neural Architecture Encoding. Majority of existing NAS work use one-hot adjacency matrix to encode the structures of neural architectures. However, adjacency matrix-based encoding grows quadratically as the search space scales up. (Ying et al., 2019) proposes categorical adjacency matrix-based encoding to ensure fixed length encodings. They also propose continuous adjacency matrix-based encoding that is similar to DARTS (Liu et al., 2019a), where the architecture is created by taking fixed number of edges with the highest continuous values. However, this approach is not easily applicable to some NAS algorithms such as regularized evolution (Real et al., 2017) without major changes. Tabular encoding in the form of ConfigSpace (Lindauer et al., 2019) is often used for hyperparameter optimization (Li et al.,

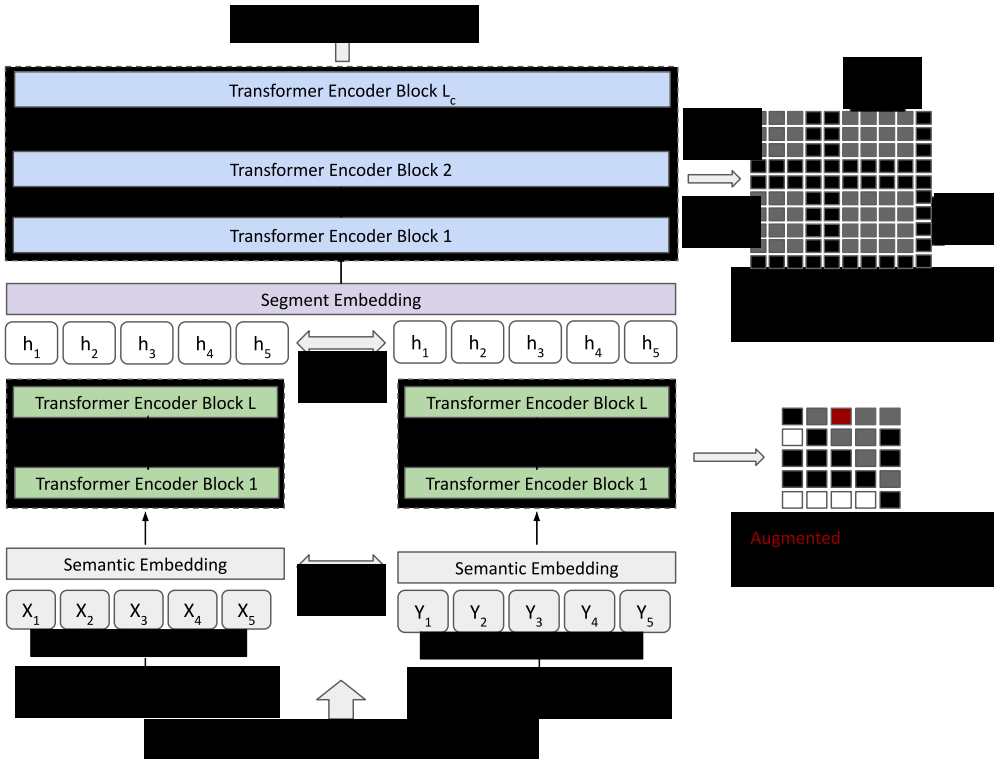

Figure 1. Overview of CATE. CATE takes computationally similar architecture pairs as the input. The model is trained to predict masked operators given the pairwise computational information. Apart from the cross-attention blocks, the pretrained Transformer encoder is used to extract architecture encodings for the downstream encoding-dependent NAS subroutines.

2018; Falkner et al., 2018) and recently adopted by NAS-Bench-301 (Siems et al., 2020) to represent architectures by introducing categorical hyperparameters for each operation along each potential edge. Recent NAS methods (Luo et al., 2018a; Wang et al., 2020; Wen et al., 2020; Shi et al., 2020) use adjacency matrix as the input to LSTM/MLP/GNN to encode the structures of neural architectures in the latent space. (Yan et al., 2020) validates that pre-training architecture representations without using accuracies can better preserve the local structural relationship of neural architectures in the latent space. (Wei et al., 2020b) proposes to learn architecture representations using contrastive learning to find low-dimensional embeddings. (Choi et al., 2021) studies various locality-based self-supervised objectives on the effect of architecture representations. One disadvantage of these methods is that they rely on a prior where the edit distance closeness between different architectures is a good indicator of the relative performance; however, structure-aware encodings may not be computationally unique unless some certain graph hashing is applied (Ying et al., 2019; Ning et al., 2020b). (White et al., 2021; Wei et al., 2020a) use path encoding and its categorical and continuous variants, which encode computation of architectures so that isomorphic cells are mapped to the same encoding. (Zhang et al., 2019) uses GRU-based asynchronous message passing to encode computation of architectures and the model is trained with the VAE loss. (Lukasik et al., 2021) proposes a two-sided variational encoder-decoder GNN to learn smooth embeddings in various NAS search spaces. CATE is inspired by the advantage of computation encoding and addresses the drawbacks of (Zhang et al., 2019; White et al., 2021). Another line of work is based on the intrinsic properties of the architectures. (Hesslow & Poli, 2021) generates architecture representations by using contrastive learning over data Jacobian matrix values computed based on different initializations, and the generated embeddings are independent of the parameterization of the search space.

Context Dependency. Our work is close to self-supervised learning in language models (LMs) (Dong et al., 2019). In particular, ELMo (Peters et al., 2018) uses two shallow unidirectional LSTMs (Hochreiter & Schmidhuber, 1997) to encode bidirectional text information, which is not sufficient for modeling deep interactions between the two directions. GPT-2 (Radford et al., 2019) proposes an autoregressive language modeling method with Transformer (Vaswani et al., 2017) to cover the left-to-right dependency and is further generalized by XLNet (Yang et al., 2019) which encodes bidirectional context. (Ro)BERT/BART/T5 (Devlin et al., 2019; Liu et al., 2019b; Lewis et al., 2020; Raffel et al.,

2020) use bidirectional Transformer encoder to encode both left and right context. In architecture representation learning, however, the attention mask in the encoder cannot be used to attend to all the operators because it does not reflect the single-directional flow of the computational graphs (Xie et al., 2019a; You et al., 2020a).

3. CATE

3.1. Search Space

We restrict our search space to the cell-based architectures. Following the configuration in (Ying et al., 2019), each cell is a labeled directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ as a set of N nodes and $\mathcal{E}$ as a set of edges that connect the nodes. Each node $v_i \in \mathcal{V}$, $i \in [1, N]$ is associated with an operation selected from a predefined set of V operations, and the edges between different nodes are represented as an upper triangular binary adjacency matrix $\mathbf{A} \in \{0,1\}^{N \times N}$.

3.2. Computation-aware Neural Architecture Encoder

Our proposed computation-aware neural architecture encoder is built upon the Transformer encoder architecture which consists of a semantic embedding layer and L Transformer blocks stacked on top. Given $\mathcal{G}$, each operation v_i is first fed into a semantic embedding layer of size d_e:

$$\mathbf{Emb}_i = \mathbf{Embedding}(v_i) \tag{1}$$

The embedded vectors are then contextualized at different levels of abstract. We denote the hidden state after l-th layer as $\mathbf{H}^l = [\mathbf{H}^l_1, ..., \mathbf{H}^l_N]$ of size d_h, where $\mathbf{H}^l = T(\mathbf{H}^{l-1})$ and T is a transformer block containing n_{head} heads. The l-th Transformer block is calculated as:

$$\mathbf{Q}_k = \mathbf{H}^{l-1}\mathbf{W}^l_{qk}, \mathbf{K}_k = \mathbf{H}^{l-1}\mathbf{W}^l_{kk}, \mathbf{V}_k = \mathbf{H}^{l-1}\mathbf{W}^l_{vk} \tag{2}$$

$$\hat{\mathbf{H}}^l_k = \textit{softmax}(\frac{\mathbf{Q}_k\mathbf{K}^T_k}{\sqrt{d_h}} + \mathbf{M})\mathbf{V}_k \tag{3}$$

$$\hat{\mathbf{H}}^l = \textit{concatenate}(\hat{\mathbf{H}}^l_1, \hat{\mathbf{H}}^l_2, \ldots, \hat{\mathbf{H}}^l_{n_{head}}) \tag{4}$$

$$\mathbf{H}^l = \mathbf{ReLU}(\hat{\mathbf{H}}^l\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{5}$$

where the initial hidden state $\mathbf{H}^0_i$ is $\mathbf{Emb}_i$, thus $d_e = d_h$. $\mathbf{Q}_k$, $\mathbf{K}_k$, $\mathbf{V}_k$ stand for "Query", "Key" and "Value" in the attention operation of the k-th head respectively. $\mathbf{M}$ is the attention mask in the Transformer, where $\mathbf{M}_{i,j} \in \{0, -\infty\}$ indicates whether operation j is a dependent operation of operation i. $\mathbf{W}_1 \in \mathbb{R}^{d_c \times d_{ff}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d_c}$ denote the weights in the feed-forward layer.

Direct/Indirect Dependency Mask. A pair of nodes (operations) within an architecture are dependent if there is either a directed edge that directly connects them (*local dependency*) or a path made of a series of such edges that indirectly connects them (*long-range dependency*). We create dependency masks for such pairs of nodes for both direct and indirect cases and use these dependency masks as the attention masks in the Transformer. Specifically, the direct dependency mask $\mathbf{M}^{Direct}$ and the indirect dependency mask $\mathbf{M}^{Indirect}$ can be created as follows:

$$\mathbf{M}^{Direct}_{i,j} = \begin{cases} 0, & \textit{if} \quad A_{i,j} = 1 \\ -\infty, & \textit{if} \quad A_{i,j} = 0 \end{cases}$$

$$\mathbf{M}^{Indirect}_{i,j} = \begin{cases} 0, & \textit{if} \quad \tilde{A}_{i,j} = 1 \\ -\infty, & \textit{if} \quad \tilde{A}_{i,j} = 0 \end{cases}$$

where $\mathbf{A}$ is the adjacency matrix and $\tilde{\mathbf{A}} = \textit{Floyed}(\mathbf{A})$ is derived using Floyd algorithm in Algorithm 1.

Algorithm 1 Floyd Algorithm

Input: the node set $\mathcal{V}$, the adjacent matrix $\mathbf{A}$
$\tilde{\mathbf{A}} \leftarrow \mathbf{A}$
for $k \in \mathcal{V}$ **do**
  for $i \in \mathcal{V}$ **do**
    for $j \in \mathcal{V}$ **do**
      $\tilde{\mathbf{A}}_{i,j} \mid = \tilde{\mathbf{A}}_{i,k} \quad \& \quad \tilde{\mathbf{A}}_{k,j}$
Output: $\tilde{\mathbf{A}}$

Uni/Bidirectional Encoding. Finally, the final hidden vector $\mathbf{H}^l_N$ is used as the unidirectional encoding for the architecture. We also considered encoding the architecture in a bidirectional manner, where both the output node hidden vector from the original DAG and the input node hidden vector from the reversed one are extracted and then concatenated together. However, our experiments show that bidirectional encoding performs worse than unidirectional encoding. We include this result in Appendix A.

3.3. Pre-training CATE

Architecture Pair Sampling. We split the dataset into 95% training and 5% held-out test sets for our pairwise pre-training. To ensure that it does not scale with quadratic time complexity, we first sort the architectures based on their computational attributes $\mathbf{P}$ (*e.g.* number of parameters, FLOPs). We then employ a sliding window for each architecture x^i and its neighborhood $r(x^i) = \{y : |\mathbf{P}(x^i) - \mathbf{P}(y)| < \delta\}$, where δ is a hyperparameter for the pairwise computation constraint. Finally, we randomly select K distinct architectures $Y = \{y^1, \ldots, y^K\}, x^i \notin Y, Y \subset r(x^i)$ within the neighborhood to compose K architecture pairs $\{(x^i, y^1), \ldots, (x^i, y^K)\}$ for architecture x^i.

Pairwise Pre-training with Cross-Attention. Once the computationally similar architecture pair is composed, we randomly select 20% operations from each architecture within the pair for masking, where 80% of them are re-

placed with a $[MASK]$ token and the remaining 20% are replaced with a random token chosen from the predefined operation set. We apply padding to architectures that have nodes less than the maximum number of nodes N in one batch to handle variable length inputs. The joint representation $\mathbf{H}_{XY}^L$ is derived by concatenating $\mathbf{H}_X^L$ and $\mathbf{H}_Y^L$ followed by the summation of the corresponding segment embedding. Segment embedding acts as an identifier of different architectures during pre-training. We set it to be trainable and randomly initialized. The joint representation $\mathbf{H}_{XY}^L$ is then contextualized with another L_c-layer Transformer with the cross-attention mask $\mathbf{M}_c$ such that segments from the two architectures can attend to each other given the pairwise information. For example, given two architectures X with three nodes and Y with four nodes in Figure 1, X has access to the non-padded nodes of Y and itself, and same for Y. The cross-attention dimension of the encoder is denoted as d_c. The joint representation of the last layer is used for prediction. The model is trained by minimizing the cross-entropy loss computed using the predicted operations and the original operations.

3.4. Encoding-dependent NAS Subroutines

(White et al., 2020a) identifies three major encoding-dependent subroutines included in existing NAS algorithms: *sample random architecture*, *perturb architecture*, and *train predictor model*. The *sample random architecture* subroutine includes random search (Li & Talwalkar, 2019). The *perturb architecture* subroutine includes regularized evolution (REA) (Real et al., 2019) and local search (LS) (White et al., 2020b). The *train predictor model* subroutine includes neural predictor (Wen et al., 2020; Shi et al., 2020; White et al., 2021), Bayesian optimization with Gaussian process (GP) (Rasmussen & Williams, 2006), and Bayesian optimization with neural networks (DNGO) (Snoek et al., 2015) which is much faster to fit compared to GP and scales linearly with large datasets rather than cubically.

Inspired by (Ottelander et al., 2020; White et al., 2020b), we found that LS (*perturb architecture*) can be combined with DNGO (*train predictor model*). We thus propose a DNGO-based computation-aware search using CATE called CATE-DNGO-LS. Specifically, we maintain a pool of sampled architectures and take iterations to add new ones. In each iteration, we pass all architecture encodings to the predictor trained 30 epochs with samples in the current pool. We select new architectures with top-5 predicted accuracy and add them to the pool. Assume there are M new architectures which become the new top-5 in the updated pool. We then select the nearest neighbors of the other (5-M) top-5 architectures in L2 distance in latent space and add them to the pool. Hence, there will be 5 to 10 new architectures added to the pool in each iteration. The search stops when the number of samples reaches a pre-defined budget.

4. Experiments

We describe two NAS benchmarks used in our experiments.

NAS-Bench-101. The NAS-Bench-101 search space (Ying et al., 2019) consists of $423,624$ architectures. Each architecture has its pre-computed validation and test accuracies on CIFAR-10. The cell includes up to 7 nodes and at most 9 edges with the first node as input and the last node as output. The intermediate nodes can be either 1×1 convolution, 3×3 convolution, or 3×3 max pooling. We use the number of network parameters as the computational attribute $\mathbf{P}$ for architecture pair sampling. We set δ to $2,000,000$ and K to 2. The ablation studies on δ and K are summarized in Section 4.4. We split the dataset into 95% training and 5% held-out test sets for pre-training.

NAS-Bench-301. NAS-Bench-301 (Siems et al., 2020) is a new surrogate benchmark on the DARTS (Liu et al., 2019a) search space that is much larger than NAS-Bench-101. It was created by fully training $60,000$ architectures that is *stratified by the NAS methods*[1] with a good coverage and then fitting a surrogate model that can estimate the accuracy (with noise) at epoch 100 and the training time for any of the remaining 10^{18} architectures. To convert the DARTS search space into one with the same input format as NAS-Bench-101, we add a summation node to make nodes represent operations and edges represent data flow. Following (Liu et al., 2018a), we use the same cell for both normal and reduction cell, allowing roughly 10^9 DAGs without considering graph isomorphism. More details about the DARTS/NAS-Bench-301 and a cell transformation example are included in Appendix D. We randomly sample $1,000,000$ architectures in this search space, and use the same data split used in NAS-Bench-101 for pre-training. We use network FLOPs as the computational attribute $\mathbf{P}$ for architecture pair sampling. We set δ to $5,000,000$ and K to 1. Since some NAS methods we compare against use the same GIN (Xu et al., 2019) surrogate model used in NAS-Bench-301, to ensure fair comparison, we thus followed (Siems et al., 2020) to use XGB-v1.0 and LGB-runtime-v1.0 which utilizes gradient boosted trees (Chen & Guestrin, 2016; Ke et al., 2017) as the regression model.

Model and Training. We use a $L = 12$ layer Transformer encoder and a $L_c = 24$ layer cross-attention Transformer encoder, each has 8 attention heads. The hidden state size is $d_h = d_c = 64$ for all the encoders. The hidden dimension is $d_{ff} = 64$ for all the feed-forward layers. We employ AdamW (Loshchilov & Hutter, 2019) as our optimizer. The initial learning rate is 1e-3. The momentum parameters are set to 0.9 and 0.999. The weight decay is 0.01 for regular layer and 0 for dropout and layer normalization. We trained

[1]We suggest referring to C.2 in (Siems et al., 2020) for a detailed description on the data collection.

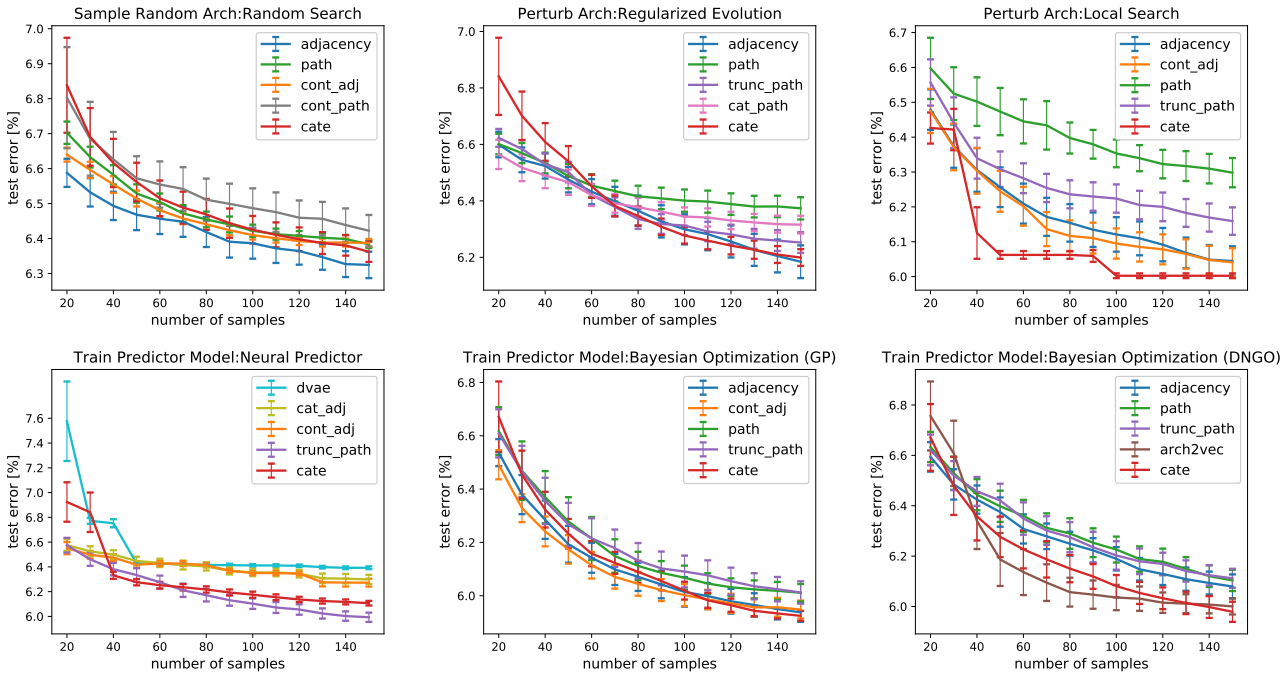

Figure 2. Comparison between CATE and other architecture encoding schemes under different subroutines on NAS-Bench-101: *sample random architecture* (top left), *perturb architecture* (top middle, top right), and *train predictor model* (bottom left, bottom middle, bottom right). It reports the test error of 200 independent runs given 150 queried architectures.

our model with batch size of 1024 on NVIDIA Quadro RTX 8000 GPUs. It takes around 4GB GPU memory for NAS-Bench-101 and 9GB GPU memory for NAS-Bench-301. The validation loss converges well after 10 epochs of pretraining, which takes 1.2 hours on NAS-Bench-101 and 7.5 hours on NAS-Bench-301.

4.1. Comparison with Different Encoding Schemes

In our first experiment, we compare CATE with eleven architecture encoding schemes under three major encoding-dependent subroutines described in Section 3.4 on NAS-Bench-101. These encoding schemes include (1-3) one-hot/categorical/continuous adjacency matrix encoding (Ying et al., 2019), (4-6) one-hot/categorical/continuous path encoding and (7-9) their corresponding truncated counterparts (White et al., 2021), (10) D-VAE (Zhang et al., 2019), and (11) arch2vec (Yan et al., 2020). For continuous encodings, we use L2 distance as the distance metric. To examine the effectiveness of the encoding schemes themselves, we compare different encoding schemes under the same search subroutine.

Figure 2 illustrates our results. For each subroutine, we show the top-five best-performing encoding schemes. Overall, despite there is no overall best encoding, we found that CATE is among the top five across all the subroutines.

Specifically, for *sample random architecture* subroutine, random search using adjacency matrix encoding performs the best. The random search using continuous encodings performs slightly worse than the adjacency encodings possibly due to the discretization loss from vector space into a fixed number of bins of same size before the random sampling.

For *perturb architecture* subroutine, CATE is on par with or outperforms adjacency encoding and path encoding because it is pre-trained to preserve strong computation locality information. This advantage allows the evolution or local search to find architectures with similar performance in local neighborhood more easily. Interestingly, we observe very small deviation using local search with CATE. This indicates that it always converges to some certain local minimums across different initial seeds. Since NAS-Bench-101 already exhibits locality in edit distance, encoding computation makes architectures even closer in terms of accuracy and thus benefits the local search.

For *train predictor model* subroutine, we have four observations: 1) Adjacency matrix encodings perform less effective with neural predictor and DNGO. It is possibly that edit distance cannot fully reflect the closeness of architectures w.r.t their actual performance. 2) Path encoding performs well

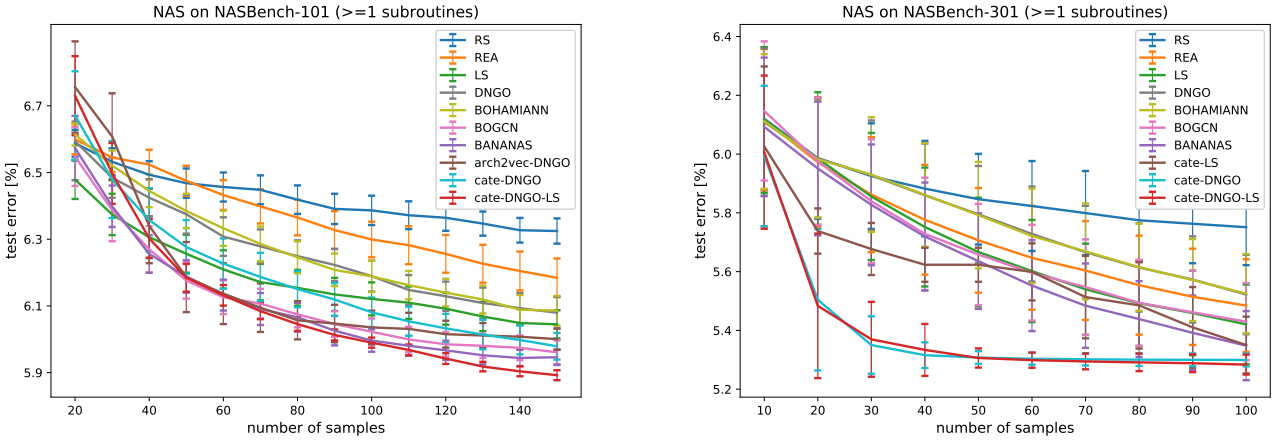

Figure 3. Comparison between CATE and SOTA NAS methods on NAS-Bench-101 (left) and NAS-Bench-301 (right). It reports the test error of 200 independent runs. The error bars denote the variance of the test error. The number of queried architectures is set to 150 for NAS-Bench-101 and 100 for NAS-Bench-301.

with neural predictor but worse than other encodings with Bayesian optimization. 3) D-VAE and arch2vec, two encodings learned via variational autoencoding, perform well only with some certain NAS methods. It could be attributed to their challenging training objective which easily leads to overfitting. 4) CATE is competitive with neural predictor and outperforms all the other encodings with Bayesian optimization. This is because neighboring computation-aware encodings correspond with similar accuracies. Moreover, the training objective in CATE is more efficient compared to the standard VAE loss (Kingma & Welling, 2014) used by D-VAE and arch2vec.

4.2. Comparison with Different NAS Methods

In our second experiment, we compare the neural architecture search performance based on CATE encodings with state-of-the-art NAS algorithms on NAS-Bench-101 and NAS-Bench-301. Existing NAS algorithms contain one or more encoding-dependent subroutines. We consider six NAS algorithms that contain one encoding-dependent subroutine: random search (RS) (Li & Talwalkar, 2019) (*sample random arch.*), regularized evolution (REA) (Real et al., 2019) (*perturb arch.*), local search (LS) (White et al., 2020b) (*perturb arch.*), DNGO (Snoek et al., 2015) (*train predictor*), BOHAMIANN (Springenberg et al., 2016) (*train predictor*), arch2vec-DNGO (Yan et al., 2020) (*train predictor*), and two NAS algorithms that contain more than one encoding-dependent subroutine: BOGCN (Shi et al., 2020) (*perturb arch.*, *train predictor*) and BANANAS (White et al., 2021) (*sample random arch.*, *perturb arch.*, *train predictor*). We compare these eight existing NAS algorithms with CATE-DNGO: a NAS algorithm based on CATE encodings with the DNGO subroutine (*train predictor*), and CATE-DNGO-LS: a NAS algorithm based on CATE encodings with the combination of DNGO and LS subroutines (*train predictor*, *perturb arch.*) as described in Section 3.4.

NAS methods	NAS-Bench-101	NAS-Bench-301
Prev. SOTA (White et al., 2021)	5.92	5.35
CATE-DNGO-LS (ours)	**5.88**	**5.28**

Table 1. Comparison between CATE and state-of-the-arts: Final test error [%] given 150 queried architectures on NAS-Bench-101 and 100 queried architectures on NAS-Bench-301. The result is averaged over 200 independent runs.

Figure 3 and Table 1 summarize our results. We have three major findings from Figure 3: 1) Architecture encoding matters especially in the large search space. The right plot shows that CATE-DNGO and CATE-DNGO-LS in DARTS search space not only converge faster but also lead to better final search performance given the same budgets. 2) Local search (LS) is a strong baseline in both small and large search spaces. As mentioned in Section 4.1, performing LS using CATE leads to better results compared to other encodings. 3) NAS algorithms that use more than one encoding-dependent subroutine in general perform better than NAS algorithms with just one subroutine. Specifically, BOGCN and BANANAS that have multiple subroutines perform better than the single-subroutine NAS algorithms such as REA, DNGO, and BOHAMIANN. Moreover, CATE-DNGO-LS leads to the best performing result in both NAS-Bench-101 and NAS-Bench-301 search spaces. Meanwhile, the improvement of CATE-DNGO-LS versus CATE-DNGO shrinks in larger search space, indicating that the larger search space is more challenging to encode.

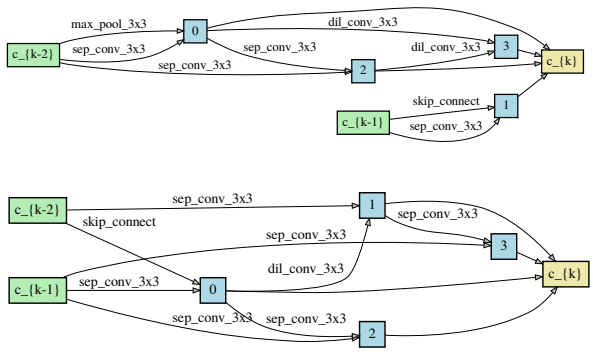

Figure 4. Top: Best found cell from CATE-DNGO-LS given the budget of 100 samples. Bottom: Best found cell from CATE-DNGO-LS given the budget of 300 samples.

NAS Methods	Avg. Test Error (%)	Params (M)	Search Cost (GPU days)
RS (Li & Talwalkar, 2019)	3.29 ± 0.15	**3.2**	4
DARTS (Liu et al., 2019a)	2.76 ± 0.09	3.3	4
BANANAS (White et al., 2021)	2.67 ± 0.07	3.6	11.8
arch2vec-BO (Yan et al., 2020)	2.56 ± 0.05	3.6	9.2
CATE-DNGO-LS (small budget)	2.55 ± 0.08	3.5	**3.3**
CATE-DNGO-LS (large budget)	$\mathbf{2.46 \pm 0.05}$	4.1	10.3

Table 2. NAS results in DARTS search space using CIFAR-10.

NAS Methods	Params (M)	Mult-Adds (M)	Top-1 Test Error (%)
SNAS (Xie et al., 2019b)	4.3	522	27.3
DARTS (Liu et al., 2019a)	4.7	574	26.7
BayesNAS (Zhou et al., 2019)	**4.0**	**440**	26.5
arch2vec-BO (Yan et al., 2020)	5.2	580	25.5
BANANAS (ours)	5.1	576	26.3
CATE-DNGO-LS (small budget)	5.0	556	26.1
CATE-DNGO-LS (large budget)	5.8	642	**25.0**

Table 3. Transfer learning results on ImageNet.

NAS-Bench-301 uses a surrogate model trained on 60k architectures to predict the performance of all the other architectures in the DARTS search space. The performance of the other architectures, however, can be inaccurate. Given that, we further validate the effectiveness of CATE-DNGO-LS in the *actual* DARTS search space by training the queried architectures from scratch. We set the budget to 100 and 300 queries, separately. Each queried architecture is trained for 50 epochs with a batch size of 96, using 32 initial channels and 8 cell layers. The average validation error of the last 5 epochs is computed as the label. These values are chosen to be close to the proxy model used in DARTS. It takes about 3.3 GPU days to finish the search with 100 quries and 10.3 GPU days with 300 queries. See Figure 4 for the best found cells. To ensure fair comparison, we compare CATE-DNGO-LS to methods (Liu et al., 2019a; Li & Talwalkar, 2019; Yan et al., 2020; White et al., 2021) that use the common test evaluation script which is to train for 600 epochs with cutout and auxiliary tower.

Table 2 summarizes our results. As shown, CATE-DNGO-LS (small budget) achieves competitive performance (2.55% avg. test error) with much less search cost and CATE-DNGO-LS (large budget) achieves superior performance (2.46% avg. test error) with similar search cost compared to other sampling-based search methods (Yan et al., 2020; White et al., 2021) in the actual DARTS search space. This is consistent with our observation in NAS-Bench-301. We report the transfer learning results on ImageNet (Deng et al., 2009) in Table 3.

4.3. Generalization to Outside Search Space

In our third experiment, inspired by (White et al., 2020a), we evaluate the generalization ability of CATE beyond the search space on which it was trained. The training search space is designed as a subset of NAS-Bench-101, where each included architecture has 2 to 6 nodes and 1 to 7 edges. The test search space is disjointed from the training search space and includes architectures with 6 nodes and 7 to 9 edges. There are $10,026$ and $60,669$ non-isomorphic graphs in the training and test space respectively. The CATE encodings are pre-trained using the training space and are used to conduct architecture search in the test space. We compare CATE with the adjacency matrix encoding because it was shown in (White et al., 2020a) to have *the best generalization capability* compared to other encodings. A simple 2-layer MLP with hidden size 128 is used as the neural predictor for both encodings.

Figure 5 shows the validation error curve of the test search space given the number of 150 sample budget across 500 independent runs. As shown, CATE outperforms adjacency matrix encoding by a large margin. This indicates that CATE can better contextualize the computation information compared to fixed encodings, which generalizes better when adapting to outside search space. Moreover, the padding scheme in our encoder allows us to handle architectures with different numbers of nodes.

4.4. Ablation Studies

Finally, we conduct ablation studies on different hyperparameters involved in CATE. We use CATE-DNGO as the NAS method and report the final NAS test error [%] given 150 queried architectures on NAS-Bench-101. The result is averaged over 200 independent runs.

Architecture Pair Sampling Hyperparameters. We plot the histogram of model parameters on NAS-Bench-101 in Figure 6. As shown, the architectures are neither nor-

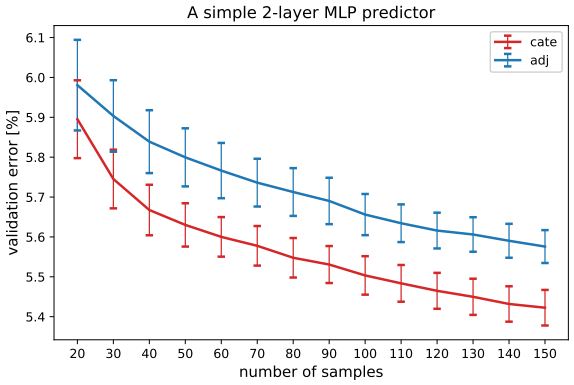

Figure 5. Performance on the out-of-training search space. It reports the validation error of 500 independent runs.

δ \ K	1	2	4	8
1×10^6	6.02	5.95	5.99	5.95
2×10^6	6.02	**5.94**	6.04	5.96
4×10^6	**5.94**	6.03	6.05	5.99
8×10^6	6.05	6.04	6.11	6.04

Table 4. Effects of δ and K on architecture pair sampling.

mally nor uniformly distributed in this search space in terms of model parameters. This motivates us to use a sliding window-based architecture pair selection to avoid the unbalanced sampling as proposed in Section 3.3. The choice of δ and K and their effects on the downstream NAS are summarized in Table 4. We found that strong computation locality (*i.e.* small δ) usually leads to better results. The choice of neighborhood size K does not have a significant effect on NAS performance. Therefore, we choose small K for faster pretraining. For NAS-Bench-301, we use the FLOPs as the computational attributes $\mathbf{P}$ and observe the same trend as in NAS-Bench-101 on the selection of δ and K. We report the results in Appendix B.

Transformer Hyperparameters. We studied the effect of the number of cross-attention Transformer blocks L_c and the hidden dimension of the feed-forward layer d_{ff} on CATE. We fix δ and K for pre-training as mentioned in Section 4. The downstream NAS result is summarized in Table 5. It shows that larger L_c and d_{ff} usually lead to better NAS performance, which indicates that deep contextualized representations are beneficial to downstream NAS.

d_{ff} \ L_c	6	12	24
64	6.07	5.99	5.95
128	6.01	**5.94**	5.95
256	5.97	**5.94**	**5.94**

Table 5. Effects of L_c and d_{ff} on pretraining CATE.

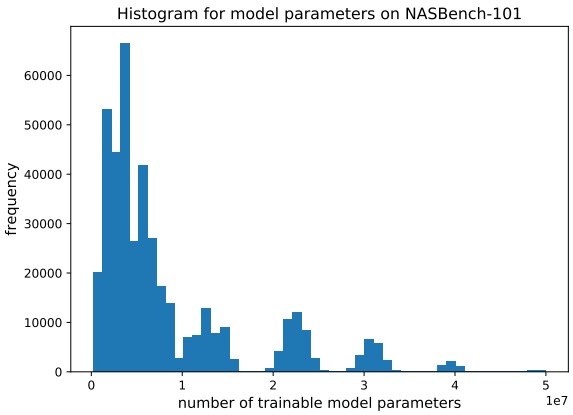

Figure 6. Histogram of model parameters on NAS-Bench-101.

Choice of Mask Type. We studied pretraining CATE with direct/indirect dependency mask and summarize its downstream NAS results in Table 6. CATE trained with indirect dependency mask outperforms the direct one in both benchmarks, indicating that capturing long-range dependency helps preserve computation information in the encodings.

Mask type	NAS-Bench-101	NAS-Bench-301
Direct	6.03	5.35
Indirect	**5.94**	**5.30**

Table 6. Direct/Indirect dependency mask selection.

5. Conclusion

In this paper, we presented CATE, a new computation-aware architecture encoding method based on Transformers. Unlike encodings with fixed transformations, we show that the computation information of neural architectures can be contextualized through a pairwise learning scheme trained with MLM. Our experimental results show its effectiveness and scalability along with three major encoding-dependent NAS subroutines in both small and large search spaces. We also show its superior generalization capability outside the training search space. We anticipate that the methods presented in this work can be extended to encode even larger search spaces (*e.g.* TuNAS (Bender et al., 2020)) to study the effectiveness of different downstream NAS algorithms.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments. We thank Yu Zheng, Colin White, and Frank Hutter for their help with this project. This work was partially supported by NSF Awards CNS-1617627, CNS-1814551, and PFI:BIC-1632051.

References

Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017.

Baker, B., Gupta, O., Raskar, R., and Naik, N. Accelerating neural architecture search using performance prediction. In *ICLR Workshop*, 2018.

Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

Bender, G., Liu, H., Chen, B., Chu, G., Cheng, S., Kindermans, P.-J., and Le, Q. Can weight sharing outperform random architecture search? an investigation with tunas. In *CVPR*, 2020.

Breiman, L. Random forests. In *Machine learning*, 2001.

Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In *SIGKDD*, 2016.

Chen, X. and Hsieh, C.-J. Stabilizing differentiable architecture search via perturbation-based regularization. In *ICML*, 2020.

Choi, K., Choe, M., and Lee, H. Pretraining neural architecture search controllers with locality-based self-supervised learning. In *arXiv preprint arXiv: 2103.08157*, 2021.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *ACL*, 2019.

Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., and Hon, H.-W. Unified language model pre-training for natural language understanding and generation. In *NeurIPS*, 2019.

Dong, X. and Yang, Y. Searching for a robust neural architecture in four gpu hours. In *CVPR*, 2019.

Dong, X. and Yang, Y. NAS-Bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020.

Drucker, H., Burges, C. J. C., Kaufman, L., Smola, A., and Vapnik, V. Support vector regression machines. In *Advances in Neural Information Processing Systems*, 1997.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. In *JMLR*, 2019.

Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. In *ICML*, 2018.

Floyd, R. W. Algorithm 97: Shortest path. In *Communications of the ACM*, 1962.

Hesslow, D. and Poli, I. Contrastive embeddings for neural architectures. In *arXiv preprint arXiv: 2102.04208*, 2021.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. In *Neural computation*, 1997.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. Neural architecture search with bayesian optimisation and optimal transport. In *NeurIPS*, 2018.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. Lightgbm: A highly efficient gradient boosting decision tree. In *NeurIPS*, 2017.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. In *ICLR*, 2014.

Kitano, H. Designing neural networks using genetic algorithms with graph generation system. In *Complex systems*, 1990.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*, 2020.

Li, L. and Talwalkar, A. Random search and reproducibility for neural architecture search. In *UAI*, 2019.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. In *JMLR*, 2018.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Marben, J., Müller, P., and Hutter, F. Boah: A tool suite for multi-fidelity bayesian optimization and analysis of hyperparameters. In *arXiv preprint arXiv: 1908.06756*, 2019.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. Progressive neural architecture search. In *ECCV*, 2018a.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. In *ICLR*, 2018b.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. In *ICLR*, 2019a.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. In *arXiv:1907.11692*, 2019b.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *ICLR*, 2019.

Lu, Z., Deb, K., Goodman, E., Banzhaf, W., and Boddeti, V. N. Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search. In *ECCV*, 2020.

Lukasik, J., Friede, D., Zela, A., Hutter, F., and Keuper, M. Smooth variational graph embeddings for efficient neural architecture search. In *IJCNN*, 2021.

Luo, R., Tian, F., Qin, T., Chen, E., and Liu, T.-Y. Neural architecture optimization. In *NeurIPS*, 2018a.

Luo, R., Tian, F., Qin, T., Chen, E.-H., and Liu, T.-Y. Neural architecture optimization. In *NeurIPS*, 2018b.

Luo, R., Tan, X., Wang, R., Qin, T., Chen, E., and Liu, T.-Y. Semi-supervised neural architecture search. In *NeurIPS*, 2020.

Miller, G. F., Todd, P. M., and Hegde, S. U. Designing neural networks using genetic algorithms. In *ICGA*, 1989.

Negrinho, R. and Gordon, G. Deeparchitect: Automatically designing and training deep architectures. In *arXiv:1704.08792*, 2017.

Ning, X., Li, W., Zhou, Z., Zhao, T., Zheng, Y., Liang, S., Yang, H., and Wang, Y. A surgery of the neural architecture evaluators. *arXiv preprint arXiv:2008.03064*, 2020a.

Ning, X., Zheng, Y., Zhao, T., Wang, Y., and Yang, H. A generic graph-based neural architecture encoding scheme for predictor-based nas. In *ECCV*, 2020b.

Ottelander, T. D., Dushatskiy, A., Virgolin, M., and Bosman, P. A. Local search is a remarkably strong baseline for neural architecture search. In *arXiv:2004.08996*, 2020.

Peng, H., Du, H., Yu, H., Li, Q., Liao, J., and Fu, J. Cream of the crop: Distilling prioritized paths for one-shot neural architecture search. In *NeurIPS*, 2020.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *NAACL*, 2018.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. In *OpenAI Blog*, 2019.

Radosavovic, I., Kosaraju, R. P., Girshick, R., He, K., and Dollár, P. Designing network design spaces. In *CVPR*, 2020.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. In *JMLR*, 2020.

Rasmussen, C. E. and Williams, C. K. I. Gaussian processes for machine learning. In *The MIT Press*, 2006.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. Large-scale evolution of image classifiers. In *ICML*, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

Ru, B., Esperanca, P., and Carlucci, F. Neural architecture generator optimization. In *NeurIPS*, 2020.

Ru, B., Wan, X., Dong, X., and Osborne, M. Interpretable neural architecture search via bayesian optimisation with weisfeiler-lehman kernels. In *ICLR*, 2021.

Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J. T., and Zhang, T. Bridging the gap between sample-based and one-shot neural architecture search with bonas. In *NeurIPS*, 2020.

Siems, J., Zimmer, L., Zela, A., Lukasik, J., Keuper, M., and Hutter, F. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. In *arXiv:2008.09777*, 2020.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., and Adams, R. Scalable bayesian optimization using deep neural networks. In *ICML*, 2015.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. Bayesian optimization with robust bayesian neural networks. In *NeurIPS*, 2016.

Stanley, K. O. and Miikkulainen, R. A. Evolving neural networks through augmenting topologies. In *Evolutionary Computation*, 2002.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

Tang, Y., Wang, Y., Xu, Y., Chen, H., Shi, B., Xu, C., Xu, C., Tian, Q., and Xu, C. A semi-supervised assessor of neural architectures. In *CVPR*, June 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *NeurIPS*, 2017.

Wang, L., Zhao, Y., Jinnai, Y., Tian, Y., and Fonseca, R. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. In *AAAI*, 2020.

Wei, C., Niu, C., Tang, Y., and min Liang, J. Npenas: Neural predictor guided evolution for neural architecture search. In *arXiv:2003.12857*, 2020a.

Wei, C., Tang, Y., Niu, C., Hu, H., Wang, Y., and Liang, J. Self-supervised representation learning for evolutionary neural architecture search. In *arXiv preprint arXiv: 2011.00186*, 2020b.

Wen, W., Liu, H., Li, H., Chen, Y., Bender, G., and Kindermans, P.-J. Neural predictor for neural architecture search. In *ECCV*, 2020.

White, C., Neiswanger, W., Nolen, S., and Savani, Y. A study on encodings for neural architecture search. In *NeurIPS*, 2020a.

White, C., Nolen, S., and Savani, Y. Local search is state of the art for neural architecture search benchmarks. In *arXiv:2005.02960*, 2020b.

White, C., Neiswanger, W., and Savani, Y. Bananas: Bayesian optimization with neural architectures for neural architecture search. In *AAAI*, 2021.

Xie, L., Chen, X., et al. Weight-sharing neural architecture search: A battle to shrink the optimization gap. In *arXiv:2008.01475*, 2020.

Xie, S., Kirillov, A., Girshick, R., and He, K. Exploring randomly wired neural networks for image recognition. In *ICCV*, 2019a.

Xie, S., Zheng, H., Liu, C., and Lin, L. Snas: Stochastic neural architecture search. In *ICLR*, 2019b.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2019.

Yan, S., Fang, B., Zhang, F., Zheng, Y., Zeng, X., Zhang, M., and Xu, H. Hm-nas: Efficient neural architecture search via hierarchical masking. In *ICCVW*, 2019.

Yan, S., Zheng, Y., Ao, W., Zeng, X., and Zhang, M. Does unsupervised architecture representation learning help neural architecture search? In *NeurIPS*, 2020.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*, 2019.

Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., and Hutter, F. NAS-Bench-101: Towards reproducible neural architecture search. In *ICML*, 2019.

You, J., Leskovec, J., He, K., and Xie, S. Graph structure of neural networks. In *ICML*, 2020a.

You, S., Huang, T., Yang, M., Wang, F., Qian, C., and Zhang, C. Greedynas: Towards fast one-shot nas with greedy supernet. In *CVPR*, 2020b.

Zela, A., Elsken, T., Saikia, T., Marrakchi, Y., Brox, T., and Hutter, F. Understanding and robustifying differentiable architecture search. In *ICLR*, 2020.

Zhang, M., Jiang, S., Cui, Z., Garnett, R., and Chen, Y. D-vae: A variational autoencoder for directed acyclic graphs. In *NeurIPS*, 2019.

Zhang, Y., Zhang, J., and Zhong, Z. Autobss: An efficient algorithm for block stacking style search. In *NeurIPS*, 2020.

Zhou, H., Yang, M., Wang, J., and Pan, W. BayesNAS: A Bayesian approach for neural architecture search. In *ICML*, 2019.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.

A. Uni/Bidirectional Encoding

As mentioned in Section 3.2, we also considered encoding the architecture in a bidirectional manner where both the output node hidden vector from the original DAG and the input node hidden vector from the reversed one are extracted and then concatenated together. Note that d_c in the cross-attention Transformer encoder will be doubled due to the concatenation. We compare the results of unidirectional and bidirectional encodings in Table 7. As shown, bidirectional encoding does not necessarily improve the results. Therefore, we keep unidirectional encoding in other experiments due to its simplicity and better performance.

Encoding	NAS-Bench-101	NAS-Bench-301
Unidirectional	**5.88**	**5.28**
Bidirectional	5.89	5.30

Table 7. Unidirectional encoding vs. bidirectional encoding. We report the final NAS test error [%] given 150 queried architectures on NAS-Bench-101 and 100 queried architectures on NAS-Bench-301. The result is averaged over 200 independent runs.

B. Architecture Pair Sampling Hyperparameters

As mentioned in Section 4.4, we randomly sample 1,000,000 architectures in NAS-Bench-301 for pretraining. We use the same proxy model configuration (*i.e.* 100 training epochs, 32 initial channels, 8 cell layers) as used in NAS-Bench-301 to compute the model FLOPs. We plot the histogram of model FLOPs of the sampled architectures in Figure 7. Given that, we experiment different δ and K and summarize the downstream NAS results in Table 8. Similar to our reported results on NAS-Bench-101, we find that strong locality leads to better results.

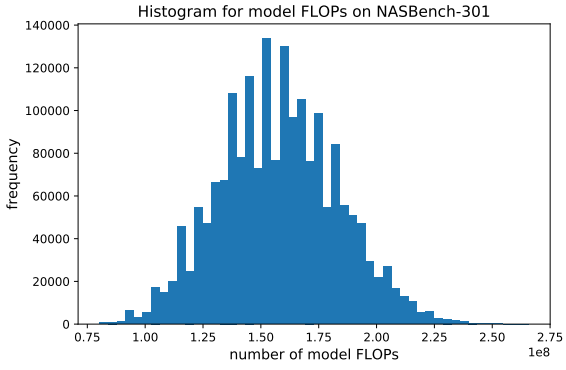

Figure 7. Histogram of model FLOPs on the sampled $1,000,000$ architectures of NAS-Bench-301.

δ \ K	1	2	4	8
5×10^6	**5.28**	5.29	5.30	5.30
1×10^7	5.30	**5.28**	5.29	5.31
2×10^7	5.30	5.30	5.31	5.32

Table 8. Effects of δ and K on architecture pair sampling on NAS-Bench-301. We report the final NAS test error [%] given 100 queried architectures on NAS-Bench-301. The result is averaged over 200 independent runs.

Corruption Rate	NAS-Bench-101	NAS-Bench-301
15%	5.89	**5.28**
20%	**5.88**	**5.28**
30%	5.93	5.29

Table 9. NAS results under different corruption rates.

C. Corruption Rate

By default, we randomly select 20% operations from each architecture within the pair for masking in the pairwise pretraining. We also experimented corruption rates of 15% and 30%. As shown in Table 9, overall, we find that the corruption rate has a limited effect on the NAS performance. Note that the number of nodes in our search space is much smaller compared to the number of tokens in the sequence modeling tasks. Given that, using larger corruption rate may slow down the training convergence and result in degraded performance. Based on these results, we use 20% corruption rate for other experiments.

D. NAS-Bench-301 Benchmark

NAS-Bench-301 (Siems et al., 2020) is the first surrogate NAS benchmark to cover the large-scale DARTS search space (Liu et al., 2019a). The DARTS search space consists of two cells: a convolutional cell and a reduction cell, each with six nodes. For each cell, the first two nodes are the outputs from the previous two cells. The next four nodes contain two edges as input, creating a DAG. In total, there are roughly 10^{18} DAGs without considering graph isomorphism, which is a much larger search space compared to NAS-Bench-101 (Ying et al., 2019) and NAS-Bench-201 (Dong & Yang, 2020).

NAS-Bench-301 is fully trained on around 60k architectures collected by unbiased architecture sampling using random search as well as biased and dense architecture sampling in high-performance regions using different NAS methods and training hyperparameters (including training time, number of parameters, and number of multiply-adds). It trains various regression models such as Random Forest (RF) (Breiman, 2001), Support Vector Regression (SVR) (Drucker et al., 1997), Graph Isomorphism Network (GIN)

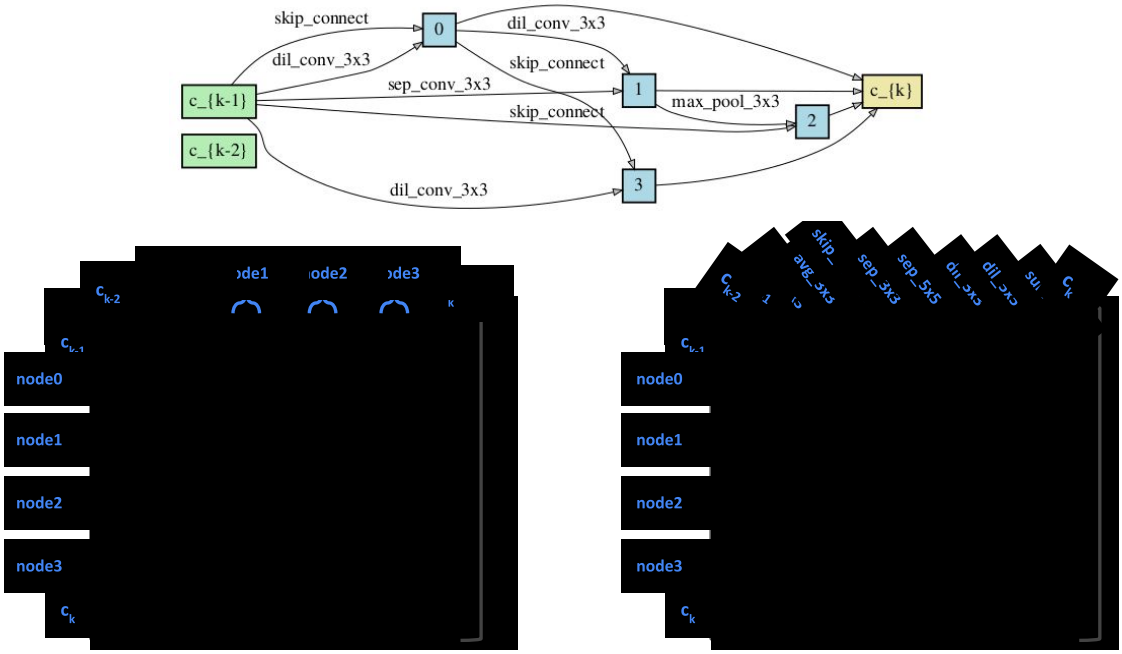

Figure 8. A cell transformation example in DARTS search space. The top panel shows the cell. The bottom-left and bottom-right panels show its corresponding adjacency matrix and operation matrix respectively.

(Xu et al., 2019) and Tree-based gradient boosting model (*e.g.* XGBoost (XGB) (Chen & Guestrin, 2016), LGBoost (LGB) (Ke et al., 2017)) to predict the accuracies of unseen architectures. The three best-performing models (GIN, XGB, LGB) are used to predict the search trajectories in the benchmark API.

D.1. Cell Transformation

To transform the DARTS search space into one with the same input format as NAS-Bench-101, we additionally add a summation node to make nodes to represent operations and edges to represent data flow. For example, if there is an edge from node A to node B with operation O, we create an additional node P, remove the edge $\langle A, B\rangle$, and add 2 edges $\langle A, P\rangle$ and $\langle P, B\rangle$. The operation on node P is set to be O. Given that, a 15×15 upper-triangular binary matrix is used to encode edges and a 15×11 operation matrix is used to encode operations with the order of $\{c_{k-2}$, c_{k-1}, 3×3 max-pool, 3×3 average-pool, skip connect, 3×3 separable conv, 5×5 separable conv, 3×3 dilated conv, 5×5 dilated conv, sum, $c_k\}$. Following NAS-Bench-301 (Siems et al., 2020), we do not include zero operator. Following (Liu et al., 2018a), we use the same cell for both normal and reduction cells. An example of cell transformation is shown in Figure 8.