A Study on Encodings for Neural Architecture Search

Anonymous Author(s)
Affiliation
Address
email

Abstract

Neural architecture search (NAS) has been extensively studied in the past few years. A popular approach is to represent each neural architecture in the search space as a directed acyclic graph (DAG), and then search over all DAGs by encoding the adjacency matrix and list of operations as a set of hyperparameters. Recent work has demonstrated that even small changes to the way each architecture is encoded can have a significant effect on the performance of NAS algorithms [24, 27].

In this work, we present the first formal study on the effect of architecture encodings for NAS, including a theoretical grounding and an empirical study. First we formally define architecture encodings and give a theoretical characterization on the scalability of the encodings we study. Then we identify the main encoding-dependent subroutines which NAS algorithms employ, running experiments to show which encodings work best with each subroutine for many popular algorithms. The experiments act as an ablation study for prior work, disentangling the algorithmic and encoding-based contributions, as well as a guideline for future work. Our results demonstrate that NAS encodings are an important design decision which can have a significant impact on overall performance. We release our code and all materials needed to reproduce our results.

1 Introduction

In the past few years, the field of neural architecture search (NAS) has seen a meteoric rise in interest [2], due to the promise of automatically designing specialized neural architectures for any given problem. Techniques for NAS span evolutionary search, reinforcement learning, gradient-based methods, and neural predictor methods. Many NAS instantiations can be described by the optimization problem $\min_{a \in A} f(a)$, where A denotes a large set of neural architectures, and $f(a)$ denotes the objective function of interest for a, which is usually a combination of validation accuracy, latency, or number of parameters. A popular approach is to describe each neural architecture a as a labeled directed acyclic graph (DAG), where each node or edge represents an operation.

Due to the complexity of DAG structures and the large size of the space, neural architecture search is typically a highly non-convex, challenging optimization problem. A natural consideration when designing a NAS algorithm is therefore, *how should we encode the neural architectures to maximize performance?* For example, NAS algorithms may involve manipulating or perturbing architectures, or training a model to predict the accuracy of a given architecture; as a consequence, the representation of the DAG-based architectures may significantly change the outcome of these subroutines. The majority of prior work has not explicitly considered this question, opting to use a standard encoding consisting of the adjacency matrix of the DAG along with a list of the operations. Two recent papers have shown that even small changes to the architecture encoding can make a substantial difference in the final performance of the NAS algorithm [24, 27]. It is not obvious how to formally define an encoding for NAS, as prior work defines encodings in different ways, inadvertently using encodings which are incompatible with other NAS algorithms.

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

In this work, we provide the first formal study on NAS encoding schemes, including a theoretical grounding as well as a thorough set of experimental results. We define an encoding as a multi-function from an architecture to a real-valued tensor. We define a number of common encodings from prior work, identifying adjacency matrix-based encodings [29, 27, 23] and path-based encodings [24, 22, 20] as two main paradigms. Adjacency matrix approaches represent the architecture as a list of edges and operations, while path-based approaches represent the architecture as a set of paths from the input to the output. We theoretically characterize the scalability of each encoding by quantifying the information loss from truncation. This characterization is particularly interesting for path-based encodings, which we find to exhibit a sharp phase change at $r^{k/n}$, where r is the number of possible operations, n is the number of nodes, and k is the expected number of edges. In particular, we show that when the size of the path encoding is greater than $r^{2k/n}$, barely any information is lost, but below $r^{k/(2n)}$, nearly all information is lost. We empirically verify these findings.

Next, we identify three major encoding-dependent subroutines used in NAS algorithms: *sample random architecture*, *perturb architecture*, and *train predictor model*. We show which of the encodings perform best for each subroutine by testing each encoding within each subroutine for many popular NAS algorithms. Our experiments retroactively provide an ablation study for prior work by disentangling the algorithmic contributions from the encoding-based contributions. We also test the ability of a neural predictor to generalize to new search spaces, using a given encoding. Finally, for encodings in which multiple architectures can map to the same encoding, we evaluate the average standard deviation of accuracies for the equivalence class of architectures defined by each encoding.

Overall, our results show that NAS encodings are an important design decision which must be taken into account not only at the algorithmic level, but at the subroutine level, and which can have a significant impact on the final performance. Based on our results, we lay out recommendations for which encodings to use within each NAS subroutine. Our experimental results follow the guidelines in the recently released NAS research checklist [9]. In particular, we experiment on two popular NAS benchmark datasets, and we release our code.

Our contributions. We summarize our main contributions below.

- We demonstrate that the choice of encoding is an important, nontrivial question that should be considered not only at the algorithmic level, but at the subroutine level.
- We give a theoretical grounding for NAS encodings, including a characterization of the scalability of each encoding, backed by experimental observations.
- We give an experimental study of architecture encodings for NAS algorithms, disentangling the algorithmic contributions from the encoding-based contributions of prior work, and laying out recommendations for best encodings to use in different settings as guidance for future work.

2 Related Work

Neural architecture search. NAS has been studied for at least two decades and has received significant attention in recent years [7, 17, 29]. Some of the most popular techniques for NAS include evolutionary algorithms [11], reinforcement learning [13, 21], Bayesian optimization [6], gradient descent [10], neural predictors [23], and local search [25]. Recent papers have highlighted the need for fair and reproducible NAS comparisons [8, 27, 9]. See the recent survey on NAS [2] for more information on NAS research.

Encoding schemes. Most prior NAS work has used the adjacency matrix encoding, [29, 27, 10], which consists of the adjacency matrix together with a list of the operations on each node. A continuous-valued variant has been shown to be more effective for some NAS algorithms [27]. The path encoding is a popular choice for neural predictor methods [24, 22, 20], and it was shown that truncating the path encoding leads to a small information loss [24].

Some prior work uses graph convolutional networks (GCN) as a subroutine in NAS [15, 28], which requires retraining for each new dataset or search space. Other work has used intermediate encodings to reduce the complexity of the DAG [18, 4], or added summary statistics to the encoding of feedforward networks [19]. To the best of our knowledge, no paper has conducted a formal study of encodings involving more than two encodings.

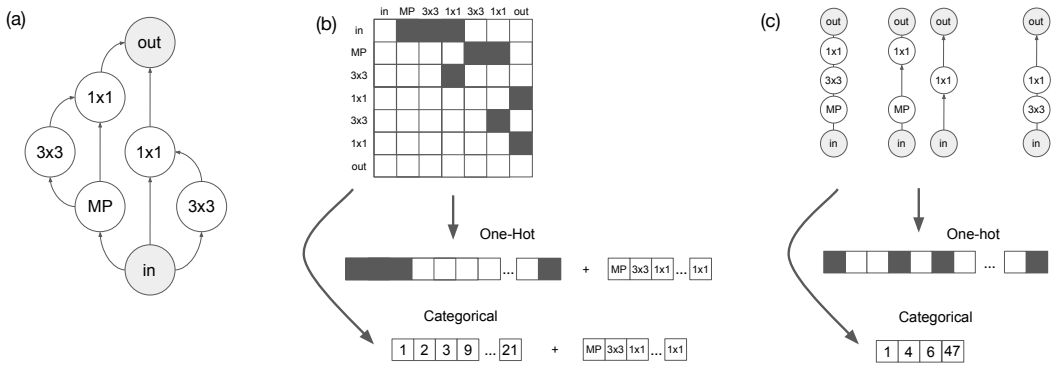

Figure 2.1: (a) An example neural architecture a. (b) A path-based representation of a, showing two encodings. (c) An adjacency matrix representation of a, showing two encodings.

3 Encodings for NAS

We denote a set of neural architectures a by A (called a search space), and we define an objective function $\ell : A \to \mathbb{R}$, where $\ell(a)$ is typically a combination of the accuracy and the model complexity. We define a neural architecture encoding as an integer d and a multifunction $e : A \to \mathbb{R}^d$ from a set of neural architectures A to a d-dimensional Euclidean space $\mathbb{R}^d$, and we define a NAS algorithm $\mathcal{A}$ as a procedure which takes as input a triple (A, ℓ, e), and outputs an architecture a, with the goal that $\ell(a)$ is as close to $\max_{a \in A} \ell(a)$ as possible. Based on this definition, we consider an encoding e to be a fixed transformation, independent of ℓ. In particular, NAS components that use ℓ to learn a transformation of an input architecture (such as graph convolutional networks or autoencoders), are considered part of the NAS algorithm rather than the encoding. This is consistent with prior definitions of encodings [20, 27].

We define eight encodings split into two popular paradigms: adjacency matrix-based and path-based encodings. We assume that each architecture is represented by a DAG with at most n nodes, at most k edges, at most P paths from input to output, and q choices of operations on each node. We focus on the case where nodes represent operations, though our analysis extends similarly to formulations where edges represent operations. Most of the following encodings have been defined in prior work, and we will see in the next section that each encoding is useful for some part of the NAS pipeline.

Adjacency matrix encodings. We first consider a class of encodings that are based on representations of the adjacency matrix. These are the most common types of encodings used in current NAS research. The *one-hot adjacency matrix encoding* is created by row-major vectorizing (i.e. flattening) the architecture adjacency matrix and concatenating it with a list of node operation labels. Each position in the operation list is a single integer-valued feature, where each operation is denoted by a different integer. The total dimension is $n(n-1)/2+n$. See Figure 2.1. In the *categorical adjacency matrix encoding*, the adjacency matrix is first flattened into a one-hot vector, and the encoding is then defined as a list of the indices each of which specifies one of the $n(n-1)/2$ possible edges in the adjacency matrix. To ensure a fixed length encoding, we allocate k features to each encoding (where k is the maximum number of possible edges). We again concatenate this representation with a list of operations, yielding a total dimensionality of $k+n$. Finally, the *continuous adjacency matrix encoding* is similar to the one-hot encoding, but each of the features for each edge can take on any real value in $[0, 1]$, rather than just $\{0, 1\}$. We also add a feature representing the number of edges, $1 \leq K \leq k$. The list of operations is encoded the same way as before. The architecture is created by choosing the K edges with the largest continuous features. The dimension is $n(n-1)/2+n+1$. The disadvantage of adjacency matrix-based encodings is that nodes are arbitrarily assigned indices in the matrix, which means one architecture can have many different representations (in other words, e^{-1} is not onto). See Figure 3.1 (b).

Path-based encodings. Path-based encodings are representations of a neural architecture that are based on the set of paths from input to output that are present within the architecture DAG. The *one-hot path encoding* is created by giving a binary feature to each possible path from the input

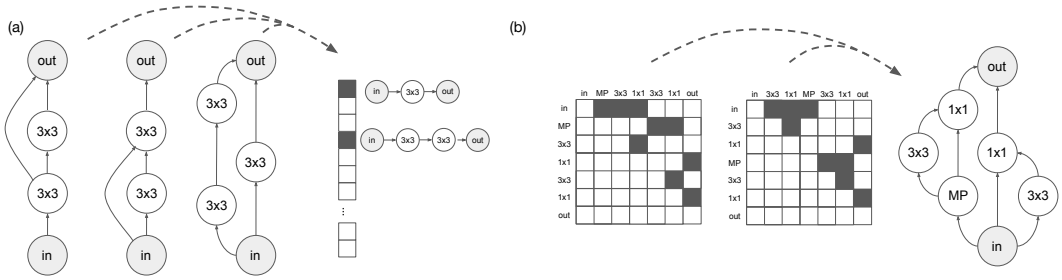

Figure 3.1: (a) An example of three architectures that map to the same path encoding. (b) An example of two adjacency matrices that map to the same architecture.

node to the output node in the DAG (for example: `input–conv1x1–maxpool3x3–output`). See Figure 2.1. The total dimension is $\sum_{i=0}^{n} q^i = (q^{n+1} - 1)/(q - 1)$. The *truncated one-hot path encoding*, simply truncates this encoding to only include paths of length x. The new dimension is $\sum_{i=0}^{x} q^i$. The *categorical path encoding*, is defined as a list of indices each of which specifies one of the $\sum_{i=0}^{n} q^i$ possible paths. The *continuous path encoding* consists of a real-valued feature $[0, 1]$ for each potential path, as well as a feature representing the number of paths. Just like the one-hot path encoding, the continuous path encoding can be truncated. Path-based encodings have the advantage that nodes are not arbitrarily assigned indices, and also that isomorphisms are automatically mapped to the same encoding. Path-based encodings have the disadvantage that different architectures can map to the same encoding (e is not onto). See Figure 3.1 (c).

3.1 The scalability of encodings

In this section, we discuss the scalability of the NAS encodings with respect to architecture size. We focus on the one-hot variants of the encodings, but our analysis extends to all encodings. We show that the path encoding can be truncated significantly while maintaining its performance, while the adjacency matrix cannot be truncated at all without sacrificing performance, and we back up our theoretical results with experimental observations in the next section. In prior work, the one-hot path encoding has been shown to be effective on smaller benchmark NAS datasets [22, 24], but it has been questioned whether its exponential $\Theta(q^n)$ length allows it to perform well on very large search spaces [20]. However, a counter-arguement is as follows. The vast majority of features correspond to single line paths using the full set of nodes. This type of architecture is not common during NAS algorithms, nor is it likely to be effective in real applications. Prior work has made the first steps in showing that truncating the path encoding does not harm the performance of NAS algorithms [24].

Consider the popular *sample random architecture* method: given n, r, and $k \leq \frac{n(n-1)}{2}$, *(1)* choose one of r operations for each node from 1 to n; *(2)* for all $i < j$, add an edge from node i to node j with probability $\frac{2k}{n(n-1)}$; *(3)* if there is no path from node 1 to node n, `goto(1)`. Given a random graph $G_{n,k,r}$ outputted by this method, let $a_{n,k,\ell}$ denote the expected number of paths from node 1 to node n of length ℓ in $G_{n,k,r}$. We define

$$b(k, x) = \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{n} a_{n,k,\ell}}.$$

Given $n < k < n(n-1)/2$ and $0 < x < n$, $b(k, x)$ represents the expected fraction of paths of length at most x in $G_{n,k,r}$ in expectation. Say that we truncate the path encoding to only include paths of length at most x. If $b(k, x)$ is very close to one, then the truncation will result in very little information loss because nearly all paths in a randomly drawn architecture are length at most x with high probability. However, if $b(k, x)$ is bounded away from 1 by some constant, there may not be enough information in the truncated path encoding to effectively run a NAS algorithm.

Prior work has shown that $b(k, x) > 1 - 1/n^2$ when $k < n + O(1)$ and $x > \log n$ [24]. However, no bounds for $b(k, x)$ are known when k is larger than a constant added to n. Now we present our main result for the path encoding, which gives a full characterization of $b(k, x)$ up to constant factors. Interestingly, we show that $b(k, x)$ exhibits a phase transition at $x = k/n$. What this means is, for the purposes of NAS, truncating the path encoding to length $r^{k/n}$ contains almost exactly the

same information as the full path encoding, and it cannot be truncated any smaller. In particular, if $k \leq n \log n$, the truncated path encoding can be length n, which is smaller than the one-hot adjacency matrix encoding. We give the full details of the proofs from this section in Appendix A.

Theorem 3.1. Given $n \leq k \leq \frac{n(n-1)}{2}$, and $c > 1$, for $x > 2ec \cdot \frac{k}{n}$, $b(k,x) > 1 - c^{-x-1}$, and for $x < \frac{1}{2ec} \cdot \frac{k}{n}$, $b(k,x) < -2^{\frac{k}{2n}}$.

Proof sketch. Let $G'_{n,k,r}$ denote a random graph after step *(2)* of *sample random architecture*. Then $G'_{n,k,r}$ may not contain a path from node 1 to node n. Let $a'_{n,k,\ell}$ denote the expected number of paths of length ℓ in $G'_{n,k,r}$. Say that a graph is *valid* if it contains a path from node 1 to node n. Then

$$a'_{n,k,\ell} = 0 \cdot (1 - P(G'_{n,k,r} \text{ is valid})) + a_{n,k,\ell} \cdot P(G'_{n,k,r} \text{ is valid}),$$

so $a_{n,k,\ell} = a'_{n,k,\ell} / P(G'_{n,k,r} \text{ is valid})$. Then

$$b(k,x) = \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{n} a_{n,k,\ell}} = \frac{\sum_{\ell=1}^{x} a'_{n,k,\ell} / P(G'_{n,k,r} \text{ is valid})}{\sum_{\ell=1}^{n} a'_{n,k,\ell} / P(G'_{n,k,r} \text{ is valid})} = \frac{\sum_{\ell=1}^{x} a'_{n,k,\ell}}{\sum_{\ell=1}^{n} a'_{n,k,\ell}}.$$

$$\text{Now we claim that } \frac{2k}{n^2} \left(\frac{2k(1-\epsilon)}{(\ell-1)n} \right)^{\ell-1} \leq a_{n,k,\ell} \leq \frac{2k}{n^2} \left(\frac{2ek(1-\epsilon)}{(\ell-1)n} \right)^{\ell-1}$$

for a small constant $0 < \epsilon < 1$. This is because on a path from node 1 to n of length ℓ, there are $\binom{n-2}{\ell-1}$ choices of intermediate nodes from 1 to n. Once the nodes are chosen, we need all ℓ edges between the nodes to exist, and each edge exists independently with probability $\frac{2}{n(n-1)} \cdot k$. Then we use the well-known binomial inequalities $\left(\frac{n}{\ell}\right)^{\ell} \leq \binom{n}{\ell} \leq \left(\frac{en}{\ell}\right)^{\ell}$ to finish the claim.

To prove the first part of Theorem 3.1, given $x > 2ec \cdot \frac{k}{n}$, we must upper bound $\sum_{\ell=x+1}^{n} a'_{n,k,\ell}$ and lower bound $\sum_{\ell=1}^{x} a'_{n,k,\ell}$. To lower bound $\sum_{\ell=1}^{x} a'_{n,k,\ell}$, we use $x > 2ec \cdot \frac{k}{n}$ with the claim:

$$\sum_{\ell=x+1}^{n} a_{n,k,\ell} \leq \sum_{\ell=x+1}^{n} \frac{2k}{n^2} \left(\frac{2ek(1-\epsilon)}{(\ell-1)n} \right)^{\ell-1} \leq \frac{2k}{n^2} \sum_{\ell=x+1}^{n} \left(\frac{1}{c} \right)^{\ell-1} \leq \left(\frac{2k}{n^2} \right) \left(\frac{1}{c} \right)^{x-1}$$

We also have $a_{n,k,1} = \frac{2k}{n^2}$ because there is just one path of length 1: the edge from the input node to the output node. Therefore, we have

$$b(k,x) = \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{n} a_{n,k,\ell}} \geq \frac{a_{n,k,1}}{a_{n,k,1} + \sum_{\ell=x+1}^{n} a_{n,k,\ell}} \geq \frac{\frac{2k}{n^2}}{\frac{2k}{n^2} + \left(\frac{2k}{n^2}\right)\left(\frac{1}{c}\right)^{x-1}} \geq 1 - c^{-x-1}.$$

The proof of the second part of Theorem 3.1 uses similar techniques. □

In Figure 4.2, we plot $b(k,x)$ for NASBench-101, which supports Theorem 3.1. Next, we may ask whether the one-hot adjacency matrix encoding can be truncated. However, even removing one bit from the adjacency matrix encoding can be very costly, because each single edge makes the difference between a path from the input node to the output node vs. no path from the input node to the output node. In the next theorem, we show that the probability of a random graph containing any individual edge is at least $2k/(n(n-1))$. Therefore, truncating the adjacency matrix encoding even by a single bit results in significant information loss. In the following theorem, let $E_{n,k,r}$ denote the edge set of $G_{n,k,r}$. Given $1 \leq z \leq n(n-1)/2$, we slightly abuse notation by writing $z \in E_{n,k,r}$ if the edge with index z in the adjacency matrix is in $E_{n,k,r}$.

Theorem 3.2. Given $n \leq k \leq \frac{n(n-1)}{2}$ and $1 \leq z \leq n(n-1)/2$, we have $P(z \in E_{n,k,r}) > \frac{2k}{n(n-1)}$.

Proof. Recall that *sample random architecture* adds each edge with probability $2k/(n(n-1))$ and rejects in step *(3)* if there is no path from the input to the output. Define $G'_{n,k,r}$ and *valid* as in the proof of Theorem 3.1. Then

$$\frac{P(G'_{n,k,r} \text{ is valid} \mid z \in E'_{n,k,r})}{P(G'_{n,k,r} \text{ is valid})} = \frac{P(z \in E'_{n,k,r} \mid G'_{n,k,r} \text{ is valid})}{P(z \in E'_{n,k,r})} > 1,$$

because there is a natural bijection ϕ from graphs with z to graphs without z given by removing z, where G is valid if $\phi(G)$ is valid but the reverse does not hold. Therefore,

$$P(z \in E_{n,k,r}) = P(z \in E'_{n,k,r} \mid G'_{n,k,r} \text{ is valid}) = \frac{P(G'_{n,k,r} \text{ is valid} \mid z \in E'_{n,k,r}) P(z \in E'_{n,k,r})}{P(G'_{n,k,r} \text{ is valid})}$$

$$> P(z \in E'_{n,k,r}) = \frac{2k}{n(n-1)}. \qquad \square$$

Our theoretical results show that the path encoding can be heavily truncated, while the adjacency matrix cannot be truncated. In the next section, we verify this experimentally (Figure 4.2).

4 Experiments

In this section, we present our experimental results. All of our experiments follow the Best Practices for NAS checklist [9]. We discuss our adherence to these practices in Appendix B. We run experiments on two NAS benchmark datasets which we describe below.

The NASBench-101 dataset [27] consists of approximately 423,000 neural architectures pretrained on CIFAR-10. The search space is a cell consisting of 7 nodes. The first node is the input, and the last node is the output. The middle five nodes can take one of three choices of operations, and there can be at most 9 edges between the 7 nodes. The NASBench-201 dataset [1] consists of 15625 neural architectures separately trained on each of CIFAR-10, CIFAR-100, and ImageNet16-120. The search space consists of a cell which is a complete directed acyclic graph with 4 nodes. Each edge takes an operation, and there are five possible operations.

We split up our first set of experiments based on the three encoding-dependent subroutines: *sample random architecture, perturb architecture*, and *train predictor model*. These three subroutines are the only encoding-dependent building blocks necessary for many NAS algorithms.

Sample random architecture. Most NAS algorithms use a subroutine to draw an architecture randomly from the search space. Although this operation is more generally parameterized by a distribution over the search space, it is often instantiated with the choice of architecture encoding. Given an encoding, we define a subroutine by sampling each feature uniformly at random. We also compare to sampling each *architecture* uniformly at random from the search space (which does not correspond to any encoding). Note that sampling architectures uniformly at random can be very computationally intensive. It is much easier to sample *features* uniformly at random.

Perturb architecture. Another common subroutine in NAS algorithms is to make a small change to a given architecture. The type of modification depends on the encoding. For example, a perturbation might be to change an operation, add or remove an edge, or add or remove a path. Given an encoding and a mutation factor m, we define a perturbation subroutine by resampling each feature of the encoding uniformly at random with a fixed probability, so that m features are modified on average.

Train predictor model. Many families of NAS algorithms use a subroutine which learns a model based on previously queried architectures. For example, this can take the form of a Gaussian process within Bayesian optimization (BO), or, more recently, a neural predictor model [15, 23, 24]. In the case of a Gaussian process model, the algorithm uses a distance metric defined on pairs of neural architectures, which is typically chosen as the edit distance between architecture encodings [6, 5]. In the case of a neural predictor, the encodings of the queried architectures are used as training data, and the goal is typically to predict the accuracy of unseen architectures.

We run multiple experiments for each encoding-dependent subroutine listed above. Many NAS algorithms use more than one subroutine, so in each experiment, we fix the encodings for all subroutines except for the one we are testing. For each NAS subroutine, we experiment on algorithms that depend on the subroutine. In particular, for *random sampling*, we run experiments on the Random Search algorithm. For *perturb architecture*, we run experiments on regularized evolution [14] and local search [25]. For *train predictor model*, we run experiments on BO, testing five encodings that define unique distance functions, as well as NASBOT [6] (which does not correspond to an encoding). We also train a neural predictor model using six different encodings. Since this runs in every iterations

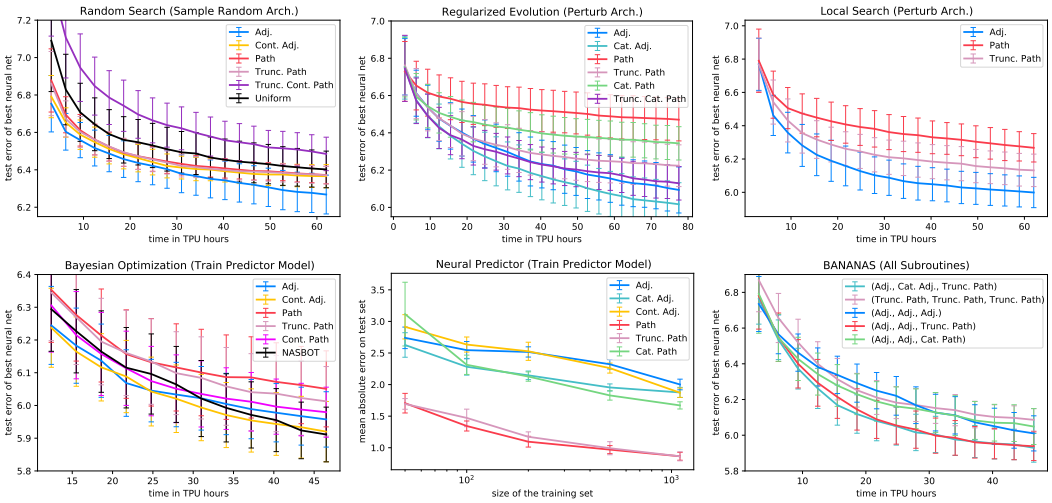

Figure 4.1: Experiments on NASBench-101 with different encodings, keeping all but one subroutine fixed: *random sampling* (top left), *perturb architecture* (top middle, top right), *train predictor model* (bottom left, bottom middle), or varying all three subroutines (bottom right).

of a NAS algorithm [15, 24, 23], we plot the mean absolute error on the test set for different sizes of training data. Finally, we run experiments on BANANAS [24], varying all three subroutines at once. We directly used the open source code for each algorithm. Details on the implementations for each algorithm are discussed in Appendix B. In each experiment, we report the test error of the neural network with the best validation error after time t, for t up to 130 TPU hours. We run at least 200 trials for each algorithm. See Figure 4.1 for the results on NASBench-101. We present more experiments for NASBench-201 in Appendix B, seeing largely the same trends.

Depending on the subroutine, two encodings might be functionally equivalent, which is why not all encodings appear in each experiment (for example, in local search, there is no difference between one-hot and categorical encodings). There is no overall best encoding; instead, each encoding has varied performance for each subroutine, and the results in Figure 4.1 act as a guideline for which encodings to use in which subroutines. For example, the one-hot adjacency matrix encoding performs well in most settings, but is quite poor in the neural predictor subroutine. Categorical, one-hot, adjacency-based, path-based, and continuous encodings are all best in certain settings. Some of our findings explain the success of prior algorithms, e.g., regularized evolution using the categorical adjacency encoding, and BANANAS using the path encoding in the meta neural network. Some of our results show new discoveries, for example, the continuous adjacency encoding has previously never been used for NAS with BO, and it outperforms all other encodings. We also show that combining the best encodings for each subroutine in BANANAS yields the best performance.

In Figure 4.1, *Trunc. Path* denotes the path encoding truncated from $\sum_{i=0}^{5} 3^i = 364$ to $\sum_{i=0}^{3} 3^i = 40$. As predicted by Theorem 3.1, this does not decrease performance. In fact, in regularized evolution, the truncation improves performance significantly because perturbing with the full path encoding is more likely to add uncommon paths that do not improve accuracy. We also evaluate the effect of truncating the one-hot adjacency matrix encoding on regularized evolution, from the full 31 bits (on NASBench-101) to 0 bits, and the path encoding from 31 bits (out of 364) to 0 bits. See Figure 4.2. The path encoding is much more robust to truncation, consistent with Theorems 3.1 and 3.2.

Outside search space experiment. In the set of experiments above, we tested the effect of encodings on a neural predictor model by computing the mean absolute error between the predicted vs. actual errors on the test set, and also by evaluating the performance of BANANAS when changing the encoding of its neural predictor model. The latter experiment tests the predictor model's ability to predict the *best* architectures, not just all architectures on average. We take this one step further and test the ability of the neural predictor to generalize beyond the search space on which it was trained. We set up the experiment as follows. We define the training search space as a subset of NASBench-101: architectures with at most 6 nodes and 7 edges. We define the disjoint test

Table 1: Ability of neural predictor with different encodings to generalize beyond the search space.

Encoding	Validation error		Test error	
	Top 10 avg.	Top 1 avg.	Top 10 avg.	Top 1 avg.
Adjacency	**5.888**	**5.505**	**6.454**	**6.056**
Categorical Adjacency	7.589	6.191	8.155	7.086
Path	5.967	5.606	6.616	6.335
Truncated Path	6.082	5.644	6.712	6.452
Categorical Path	6.357	5.703	6.939	6.489
Truncated Categorical Path	6.339	5.895	6.918	6.766

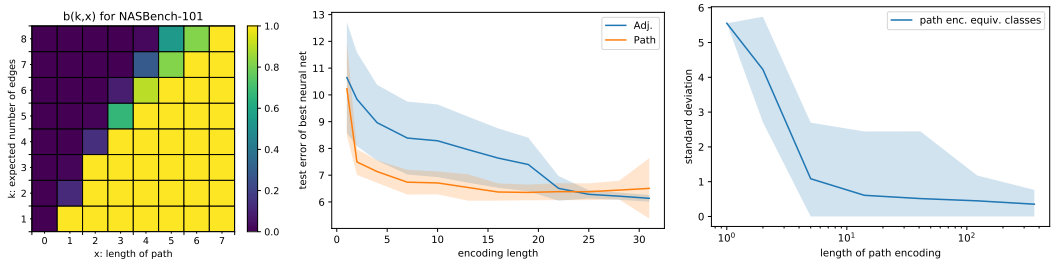

Figure 4.2: Plot of $b(k, x)$ on NASBench-101 (left). Truncation of encodings for regularized evolution on NASBench-101 (middle). Average standard deviation of accuracies within each equivalence class defined by the path encoding at different levels of truncation on NASBench-101 (right).

search space as architectures with 6 nodes and 7 to 9 edges. The neural predictor is trained on 1000 architectures and predicts the validation loss of the 5000 architectures from the test search space. We evaluate the losses of the ten architectures with the highest predicted validation loss. We run 200 trials for each encoding and average the results. See Table 1. The adjacency encoding performed the best. An explanation is that for the path encoding, there are features (paths) in architectures from the test set that do not exist in the training set. This is not the case for the adjacency encoding: all features (edges) from architectures in the test set have shown up in the training set.

Equivalence class experiments. Recall that the path encoding function e is not onto (see Figure 3.1). In general, this is not desirable because information is lost when two architectures map to the same encoding. However, if the encoding function only maps architectures with similar accuracies to the same encoding, then the behavior is beneficial. On the NASBench-101 dataset, we compute the path encoding of all 423k architectures, and then we compute the average standard deviation of accuracies among architectures with the same encoding (i.e., we look at the standard deviations within the equivalence classes defined by the encoding). The result is an average standard deviation of 0.353%, compared to the 5.55% standard deviation over the entire set of architectures. See Figure 4.2.

5 Conclusion

In this paper, we give the first formal study of encoding schemes for neural architecture search. We define eight different encodings and characterize the scalability of each one. We then identify three encoding-dependent subroutines used by NAS algorithms, *sample random architecture*, *perturb architecture*, and *train predictor model*, and we run experiments to find the best encoding for each subroutine in many popular algorithms. We also conduct experiments on the ability of a neural predictor to generalize beyond the training search space, given each encoding. Our experimental results allow us to disentangle the algorithmic and encoding-based contributions of prior work, and act as a guideline for the encodings to use in future work. Overall, we show that encodings are an important, nontrivial design decision in the field of NAS. Designing and testing new encodings is an exciting next step.

6 Broader Impact

Our work gives a study on encodings for neural architecture search, with the goal of helping future researchers improve their NAS algorithms. Therefore, this work will not directly impact society, since it is two levels of abstraction from real applications, but it can indirectly impact society. As an example, our work may inspire the creation of a new state-of-the-art NAS algorithm, which is then used to improve the performance of several deep learning algorithms, ranging from optimizers that reduce CO2 emissions, to deep fake generators.

Since our work is two levels up the stack, we have much less control over the net impact of our work on society. Due to the recent push for the AI community to be more conscious and clairvoyant about the societal impact of its work, [3] we are cautiously optimistic that our work will have a net positive impact.

References

[1] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[2] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[3] Brent Hecht, Lauren Wilcox, Jeffrey P Bigham, Johannes Schöning, Ehsan Hoque, Jason Ernst, Yonatan Bisk, Luigi De Russis, Lana Yarosh, Bushra Anjum, Danish Contractor, and Cathy Wu. It's time to do something: Mitigating the negative impacts of computing through a change to the peer review process. *ACM Future of Computing Blog*, 2018.

[4] William Irwin-Harris, Yanan Sun, Bing Xue, and Mengjie Zhang. A graph-based encoding for evolutionary convolutional neural network architecture design. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 546–553. IEEE, 2019.

[5] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282*, 2018.

[6] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018.

[7] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4(4):461–476, 1990.

[8] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[9] Marius Lindauer and Frank Hutter. Best practices for scientific research on neural architecture search. *arXiv preprint arXiv:1909.02453*, 2019.

[10] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[11] Krzysztof Maziarz, Andrey Khorlin, Quentin de Laroussilhe, and Andrea Gesmundo. Evolutionary-neural hybrid agents for architecture search. *arXiv preprint arXiv:1811.09828*, 2018.

[12] Willie Neiswanger, Kirthevasan Kandasamy, Barnabas Poczos, Jeff Schneider, and Eric Xing. Probo: a framework for using probabilistic programming in bayesian optimization. *arXiv preprint arXiv:1901.11515*, 2019.

[13] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[14] 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, pages 4780–4789, 2019.

[15] Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James T Kwok, and Tong Zhang. Multi-objective neural architecture search via predictive network performance optimization. *arXiv preprint arXiv:1911.09336*, 2019.

[16] Pantelimon Stanica. Good lower and upper bounds on binomial coefficients. *Journal of Inequalities in Pure and Applied Mathematics*, 2001.

[17] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[18] KO Stanley, DB D'Ambrosio, and J Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. In *Artificial Life*, volume 15(2), pages 185–212, 2009.

[19] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 2019.

[20] El-Ghazali Talbi. Optimization of deep neural networks: a survey and unified taxonomy. 2020.

[21] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[22] Chen Wei, Chuang Niu, Yiping Tang, and Jimin Liang. Npenas: Neural predictor guided evolution for neural architecture search. *arXiv preprint arXiv:2003.12857*, 2020.

[23] Wei Wen, Hanxiao Liu, Hai Li, Yiran Chen, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. *arXiv preprint arXiv:1912.00848*, 2019.

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

[25] Colin White, Sam Nolen, and Yash Savani. Local search is state of the art for nas benchmarks. *arXiv preprint arXiv:2005.02960*, 2020.

[26] Antoine Yang, Pedro M Esperança, and Fabio M Carlucci. Nas evaluation is frustratingly hard. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[27] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[28] Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. D-vae: A variational autoencoder for directed acyclic graphs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2019.

[29] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

A Details from Section 3 (Encodings for NAS)

We give the details from Section 3. We restate the random graph model here more formally.

Definition A.1. Given nonzero integers n, r, and $k < {}^{n(n-1)}/_2$, a random graph $G_{n,k,r}$ is generated as follows:
(1) Denote n nodes by 1 to n and label each node randomly with one of r operations.
(2) For all $i < j$, add edge (i, j) with probability $\frac{2k}{n(n-1)}$.
(3) If there is no path from node 1 to node n, `goto (1)`.

Let $G'_{n,k,r}$ denote the random graph outputted by the above procedure without step (3). Since the number of pairs (i, j) such that $i < j$ is $\frac{n(n-1)}{2}$, the expected number of edges of $G'_{n,k,r}$ is k. Define $a_{n,k,\ell}$ as the expected number of paths from node 1 to node n of length ℓ in $G'_{n,k,r}$. Formally, we set $\mathcal{P} = \{\text{paths from node 1 to } n \text{ in } G'_{n,k,r}\}$, and define

$$a_{n,k,\ell} = \mathbb{E}\left[|p \in \mathcal{P}| \mid |p| = \ell\right].$$

Recall that

$$b(k, x) = \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{n} a_{n,k,\ell}}.$$

In the next theorem, we give a full characterization of $b(k, x)$ in terms of k and n, up to constant factors. We prove there exists a phase transition for $b(k, x)$ at $x = \frac{k}{n}$. As noted by prior work [24], there are two caveats when applying this type of theorem to NAS performance. The theorem considers the distribution from Definition A.1, not the distribution of architectures encountered in a real search, and the most common paths in the distribution are not necessarily the ones with the most entropy in predicting whether an architecture has a high accuracy. However, two prior works have experimentally showed that truncating the path encoding does not decrease performance [22, 24], and we gave even more experimental evidence in Section 4.

Theorem 3.1 (restated). *Given $n \le k \le \frac{n(n-1)}{2}$, and $c > 3$, for $x > 2ec \cdot \frac{k}{n}$, $b(k, x) > 1 - c^{-x+1}$, and for $x < \frac{1}{2ec} \cdot \frac{k}{n}$, $b(k, x) < -2^{\frac{k}{2n}}$.*

To prove Theorem 3.1, we use the well-known bounds on binomial coefficients, e.g. [16].

Theorem A.2. *Given $0 \le \ell \le n$,*

$$\left(\frac{n}{\ell}\right)^{\ell} \le \binom{n}{\ell} \le \left(\frac{en}{\ell}\right)^{\ell}.$$

Now we give upper and lower bounds on $a_{n,k,\ell}$ which will be used for the rest of the proofs. The next fact is similar to Lemma C.3 from BANANAS [24].

Fact A.3. *Given $n \le k \le \frac{n(n-1)}{2}$, and $0 < x < n$, we have*

$$\frac{2k}{n^2}\left(\frac{2k(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} \le a_{n,k,\ell} \le \frac{2k}{n^2}\left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1}$$

Proof. First, we have

$$a_{n,k,\ell} = \binom{n-2}{\ell-1}\left(\frac{2k}{n(n-1)}\right)^{\ell}$$

because on a path from node 1 to node n with length ℓ, there are $\binom{n-2}{\ell-1}$ choices of intermediate nodes from 1 to n. Once the nodes are chosen, we need all ℓ edges between the nodes to exist, and each edge exists independently with probability $\frac{2}{n(n-1)} \cdot k$. Then we achieve the desired result by applying Theorem A.2. □

Now we prove the upper bound of Theorem 3.1.

Lemma A.4. *Given $n \le k \le \frac{n(n-1)}{2}$ and $c > 2$, for $x > \frac{2eck}{n}$, $b(k, x) > 1 - c^{-x+1}$.*

Proof. Given $n \le k \le \frac{n(n-1)}{2}$ and $x > \frac{2eck}{n}$, we give a lower bound for $\sum_{\ell=1}^{x} a_{n,k,\ell}$ and an upper bound for $\sum_{\ell=x+1}^{n} a_{n,k,\ell}$.

When $\ell = 1$, we have $\binom{n-2}{\ell-1} = 1$. Therefore,

$$\sum_{\ell=1}^{x} a_{n,k,\ell} \ge a_{n,k,1} = \left(\frac{2k}{n(n-1)}\right) \ge \frac{2k}{n^2}.$$

Now we upper bound $\sum_{\ell=x}^{n} a_{n,k,\ell}$.

$$\begin{aligned}
\sum_{\ell=x+1}^{n} a_{n,k,\ell} &\le \sum_{\ell=x+1}^{n} \frac{2k}{n^2} \left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} \\
&= \frac{2k}{n^2} \sum_{\ell=x+1}^{n} \left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} \\
&\le \frac{2k}{n^2} \sum_{\ell=x+1}^{n} \left(\frac{1}{c}\right)^{\ell-1} \\
&= \left(\frac{2k}{n^2}\right)\left(\frac{1}{c}\right)^{x} \sum_{\ell=0}^{\infty} \left(\frac{1}{c}\right)^{\ell} \qquad \text{(A.1)} \\
&= \left(\frac{2k}{n^2}\right)\left(\frac{1}{c}\right)^{x} \left(\frac{c}{c-1}\right) \\
&= \left(\frac{2k}{n^2}\right)\left(\frac{1}{c}\right)^{x-1}
\end{aligned}$$

In inequality A.1, we use the fact that for all $\ell \ge x$,

$$\ell \ge x > \frac{2eck}{n} \implies \frac{2ek(1-\epsilon)}{n\ell} \le \frac{1}{c}$$

when $c > 2$, since $1 - \epsilon > \frac{n-2}{n-1}$ in Fact A.3.

Therefore, we have

$$\begin{aligned}
b(k,x) &= \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{n} a_{n,k,\ell}} \\
&= \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{x} a_{n,k,\ell} + \sum_{\ell=x+1}^{n} a_{n,k,\ell}} \\
&\ge \frac{\frac{2k}{n^2}}{\frac{2k}{n^2} + \left(\frac{2k}{n^2}\right)\left(\frac{1}{c}\right)^{x-1}} \\
&= \frac{1}{1 + \left(\frac{1}{c}\right)^{x-1}} \\
&\ge 1 - c^{-x+1}.
\end{aligned}$$

□

Now we prove the lower bound for Theorem 3.1.

Lemma A.5. *Given $n \le k \le \frac{n(n-1)}{2}$ and $c > 3$, for $x < \frac{k}{2ecn}$, $b(k,x) < 2^{-\frac{k}{2n}}$.*

Proof. Given $n \le k \le \frac{n(n-1)}{2}$ and $x < \frac{k}{2ecn}$, now we give an upper bound for $\sum_{\ell=1}^{x} a_{n,k,\ell}$ and a lower bound for $\sum_{\ell=x+1}^{n} a_{n,k,\ell}$.

First we make the following claim. For all $1 \leq \ell \leq x < \frac{k}{2ecn}$, we have

$$\left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell} < \left(4e^2c(1-\epsilon)\right)^{\frac{k}{2ecn}}. \tag{A.2}$$

To prove the claim, we have

$$\begin{aligned}
\left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell} &= e^{\log\left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)\ell} && \text{(A.3)}\\
&= e^{\ell\log\frac{1}{\ell}+\ell\log\left(\frac{2ek(1-\epsilon)}{n}\right)} \\
&\leq e^{\frac{k}{2ecn}\log\left(\frac{2ecn}{k}\right)+\frac{k}{2ecn}\log\left(\frac{2ek(1-\epsilon)}{n}\right)} && \text{(A.4)}\\
&= e^{\log\left(4e^2c(1-\epsilon)\right)\frac{k}{2ecn}} && \text{(A.5)}\\
&= \left(4e^2c(1-\epsilon)\right)^{\frac{k}{2ecn}}
\end{aligned}$$

In inequality A.5, we use the fact that $\frac{k}{(\ell-1)n} < 1$ and $y > \log y$ for all $y > 1$.

Then we have

$$\begin{aligned}
\sum_{\ell=1}^{x} a_{n,k,\ell} &\leq \sum_{\ell=1}^{x} \frac{2k}{n^2}\left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} \\
&= \frac{2k}{n^2}\sum_{\ell=1}^{x}\left(\frac{2ek(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} \\
&\leq \left(\frac{2k}{n^2}\right)\cdot x\cdot\left(4e^2c(1-\epsilon)\right)^{\frac{k}{2ecn}}.
\end{aligned}$$

Now we give the lower bound for the other side of the summation.

$$\sum_{\ell=x}^{n} a_{n,k,\ell} = \sum_{\ell=x}^{n}\frac{2k}{n^2}\left(\frac{2k(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} \geq \frac{2k}{n^2}\sum_{\ell=\frac{k}{n}}^{\frac{k}{n}}\left(\frac{2k(1-\epsilon)}{(\ell-1)n}\right)^{\ell-1} = \frac{2k}{n^2}\left(2(1-\epsilon)\right)^{\frac{k}{n}}$$

Therefore,

$$\begin{aligned}
b(k,x) &= \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=1}^{n} a_{n,k,\ell}} \\
&\leq \frac{\sum_{\ell=1}^{x} a_{n,k,\ell}}{\sum_{\ell=x+1}^{n} a_{n,k,\ell}} \\
&\leq \frac{\left(\frac{2k}{n^2}\right)\cdot x\cdot\left(4e^2c(1-\epsilon)\right)^{\frac{k}{2ecn}}}{\frac{2k}{n^2}\left(2(1-\epsilon)\right)^{\frac{k}{n}}} \\
&\leq x\cdot(2e)^{\frac{k}{ecn}}\,(c)^{\frac{k}{2ecn}}\,(2)^{-\frac{k}{n}} \\
&\leq x\cdot 2^{-\frac{k}{n}\left(1-\frac{1}{c}-\frac{\log c}{2ec}\right)} \\
&\leq 2^{-\frac{k}{2n}}
\end{aligned}$$

□

The proof of Theorem 3.1 follows immediately from combining Lemmas A.4 and A.5.

B Details from Section 4 (Experiments)

In this section, we give details from Section 4, and more experiments. First we describe the algorithms used in our experiments.

- Random Search consists of randomly choosing architectures and then training them, until the runtime budget is exceeded.
- Regularized evolution [14] consists of maintaining a population of neural architectures. In each iteration, a subset is selected and the best architecture from the subset is mutated. The mutation replaces the oldest architecture from the population. We used a population size of 30. We also found that replacing the worst architecture (not the oldest) performed better, so we used this version.
- Local search [25] is a simple greedy algorithm that has only recently been applied to NAS. We use the simplest instantiation (often called the hill-climbing algorithm).
- Bayesian optimization (BO) is a strong method for zeroth order optimization. We use the ProBO [12] implementation, which uses a Gaussian process kernel and expected improvement as the acquisition function.
- NASBOT [6] is a BO-based NAS algorithm. It was not defined for cell-based search spaces, so we use a variant that works for cell-based spaces [24].
- BANANAS [24] is a BO-based method which uses a neural predictor model.

B.1 Experiments on NASBench-201

In this section, we give similar experiments to Figure 4.1, but with NASBench-201 instead of NASBench-101. Note that NASBench-201 is not as good for encoding experiments because every single architecture has the same graph structure - a clique of size 4. The only differences are the operations. Therefore, many encodings are functionally equivalent. For example, the one-hot, categorical, and continuous adjacency matrix encodings are all identical because the only difference is the way they encode the adjacency matrix. I.e., these encodings will all look like a set of operations, plus some adjacency matrix encoding that is the same for every architecture in the search space. The one-hot adjacency matrix encoding, path encoding, and truncated path encoding are all distinct from one another, so we run experiments with these encodings. See Figure B.1. We see largely the same trends as in NASBench-101 (Figure 4.1). Note that on the ImageNet-16-120 dataset, some algorithms such as NASBOT overfit to the training set, causing performance to decline over time.

B.2 Best practices for NAS

Many authors have called for improving the reproducibility and fairness in experimental comparisons in NAS research [8, 27, 26], which has led to the release of a NAS best practices checklist [9]. We address each section and we encourage future work to do the same.

- **Best practices for releasing code.** We included our code in the supplementary material. We will release our code publicly after this reviewing cycle. We used the NASBench-101 and NASBench-201 datasets, so questions about training pipeline, evaluation, and hyperparameters for the final evaluation do not apply.
- **Best practices for comparing NAS methods.** We made fair comparisons due to our use of NASBench-101 and NASBench-201. We did run ablation studies and ran random search. We performed 300 trials of each experiment on NASBench-101 and NASBench-201.
- **Best practices for reporting important details.** We used the hyperparamters straight from the open source repositories, with a few exceptions listed above.

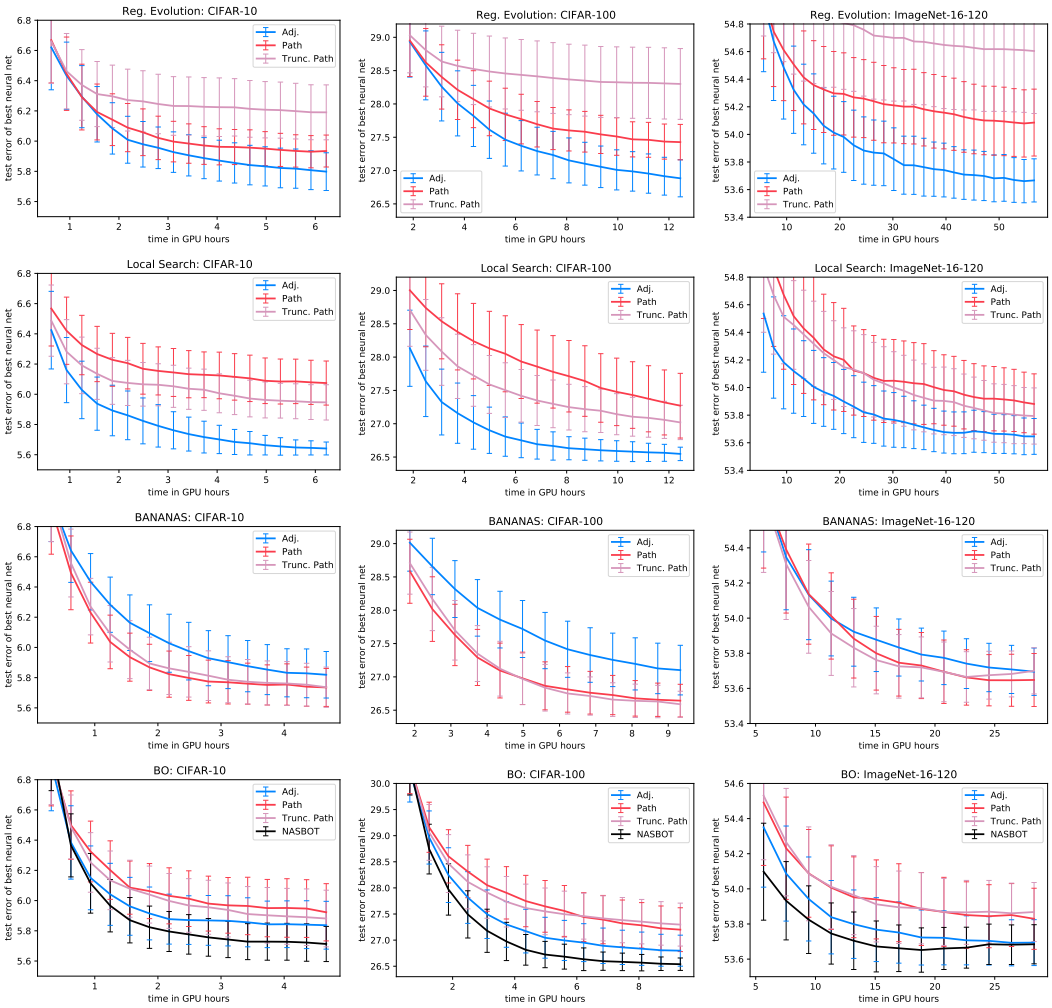

Figure B.1: Experiments on NASBench-201 with different encodings, keeping all but one subroutine fixed: *perturb architecture* (Reg. evolution (top row), local search (second row)), *train predictor model* (BANANAS (third row), Bayesian optimization (bottom row)).