BANANAS: Bayesian Optimization with Neural Architectures for Neural Architecture Search

Colin White [1] **Willie Neiswanger** [2 3] **Yash Savani** [1]

Abstract

Neural Architecture Search (NAS) has seen an explosion of research in the past few years, with techniques spanning reinforcement learning, evolutionary search, Gaussian process (GP) Bayesian optimization (BO), and gradient descent. While BO with GPs has seen great success in hyperparameter optimization, there are many challenges applying BO to NAS, such as the requirement of a distance function between neural networks. In this work, we develop a suite of techniques for high-performance BO applied to NAS that allows us to achieve state-of-the-art NAS results. We develop a BO procedure that leverages a novel architecture representation (which we term the path encoding) and a neural network-based predictive uncertainty model on this representation.

On popular search spaces, we can predict the validation accuracy of a new architecture to within one percent of its true value using only 200 training points. This may be of independent interest beyond NAS. We also show experimentally and theoretically that our method scales far better than existing techniques. We test our algorithm on the NASBench (Ying et al. 2019) and DARTS (Liu et al. 2018) search spaces and show that our algorithm outperforms a variety of NAS methods including regularized evolution, reinforcement learning, BOHB, and DARTS. Our method achieves state-of-the-art performance on the NASBench dataset and is over 100x more efficient than random search. We adhere to the recent NAS research checklist (Lindauer and Hutter 2019) to facilitate NAS research. In particular, our implementation is publicly available [1] and includes all details needed to fully reproduce our results.

[1]RealityEngines.AI [2]Carnegie Mellon University [3]Petuum Inc. Correspondence to: Colin White <colin@realityengines.ai>, Willie Neiswanger <willie@cs.cmu.edu>, Yash Savani <yash@realityengines.ai>.

[1]https://www.github.com/naszilla/bananas.

1. Introduction

Since the deep learning revolution in 2012, neural networks have been growing increasingly more specialized and more complex (Krizhevsky et al., 2012; Huang et al., 2017; Szegedy et al., 2017). Developing new state-of-the-art architectures often takes a vast amount of engineering and domain knowledge. A new area of research, neural architecture search (NAS), seeks to automate this process. Since the popular work by Zoph and Lee (Zoph & Le, 2017), there has been a flurry of research on NAS (Liu et al., 2018a; Pham et al., 2018; Liu et al., 2018b; Kandasamy et al., 2018b; Elsken et al., 2018; Jin et al., 2018).

Many methods have been proposed for NAS, including random search, evolutionary search, reinforcement learning, Bayesian optimization (BO), and gradient descent. In certain settings, zeroth-order (non-differentiable) algorithms such as BO are of particular interest over first-order techniques, due to advantages such as simple parallelism, joint optimization with other hyperparameters, easy implementation, portability to diverse architecture spaces, and optimization of other/multiple non-differentiable objectives.

BO with Gaussian processes (GPs) has had success in deep learning hyperparameter optimization (Golovin et al., 2017; Falkner et al., 2018), and is a leading method for efficient zeroth order optimization of expensive-to-evaluate functions in Euclidean spaces. However, applying BO to NAS comes with challenges that have so far limited its ability to achieve state-of-the-art results. For example, current approaches require specifying a distance function between architectures in order to define a surrogate GP model. This is often a cumbersome task involving tuning hyperparameters of the distance function (Kandasamy et al., 2018b; Jin et al., 2018). Furthermore, it can be quite challenging to achieve highly accurate prediction performance with GPs, given the potentially high dimensional input architectures.

In this work, we develop a suite of techniques for high-performance BO applied to NAS that allows us to achieve state of the art NAS results. We develop a BO procedure that leverages a novel architecture representation (which we term a *path encoding*) and a neural network-based predictive uncertainty model (which we term a *meta neural network*)

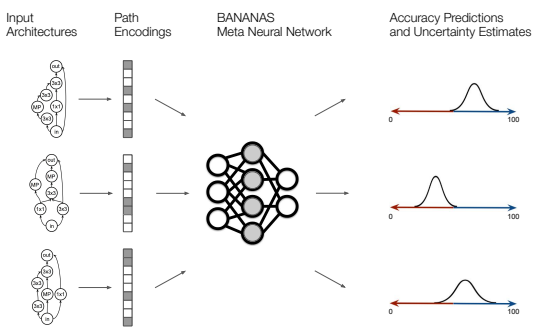

Figure 1.1: Illustration of the meta neural network in the BANANAS algorithm.

defined on this representation. In every iteration of BO, we use our model to estimate accuracies and uncertainty estimates for unseen neural architectures in the search space. This procedure avoids the aforementioned problems with BO in NAS: the model is powerful enough to provide high accuracy predictions of neural architecture accuracies, and there is no need to construct a distance function between architectures. Furthermore, our meta neural network scales far better than a GP model, as it avoids computationally intensive matrix inversions. We call our algorithm BANANAS: Bayesian optimization with neural architectures for NAS.

Training a meta neural network to predict with high accuracy is a challenging task. The majority of popular NAS algorithms are deployed over a directed acyclic graph (DAG) search space – the set of possible architectures is comprised of the set of all DAGs of a certain size, together with all possible combinations of operations on each node (Zoph & Le, 2017; Pham et al., 2018; Liu et al., 2018b). This poses a roadblock when predicting architecture accuracies, since graph structures can be difficult for neural networks when the adjacency matrix is given directly as input (Zhou et al., 2018). By contrast, our path encoding scheme provides a representation for architectures that drastically improves the predictive accuracy of our meta neural network. Each binary feature represents a unique path through the DAG from the input layer to the output layer. With just 200 training points, our meta neural network is able to predict the accuracy of unseen neural networks to within one percent on the popular NASBench dataset (Ying et al., 2019). This is an improvement over all previously reported results, and may be of interest beyond NAS. We also show experimentally and theoretically that our method scales to larger search spaces far better than the standard adjacency matrix encoding.

In BANANAS, we train an ensemble of neural networks to predict the mean and variance of validation error for candidate neural architectures, from which we compute an acquisition function. We define a new variant of the Thompson sampling acquisition function (Thompson, 1933), called independent Thompson sampling, and empirically show that it is well-suited for parallel Bayesian optimization. Finally, we use a mutation algorithm to optimize the acquisition function. We compare our NAS algorithm against a host of NAS algorithms including regularized evolution (Real et al., 2019), REINFORCE (Williams, 1992), Bayesian optimization with a GP model (Snoek et al., 2012), AlphaX (Wang et al., 2018), ASHA (Li & Talwalkar, 2019), DARTS (Liu et al., 2018b), TPE (Bergstra et al., 2011), DNGO (Snoek et al., 2015), and NASBOT (Kandasamy et al., 2018b). On the NASBench dataset, our method achieves state-of-the-art performance and beats random search by a factor of over 100. On the search space from DARTS, when given a budget of 100 neural architecture queries for 50 epochs each, our algorithm achieves a best of 2.57% and average of 2.64% test error, which beats all NAS algorithms with which we could fairly compare (same search space used, and same hyperparameters for the final training). We also show that our algorithm outperforms other methods at optimizing functions of the model accuracy and the number of model parameters.

BANANAS has several moving parts, including the meta neural network, the path-based feature encoding of neural architectures, the acquisition function, and the acquisition optimization strategy. We run a thorough ablation study by removing each piece of the algorithm separately. We show all components are necessary to achieve the best performance. Finally, we check all items on the NAS research checklist (Lindauer & Hutter, 2019), due to recent claims that NAS research is in need of more fair and reproducible empirical evaluations (Ying et al., 2019; Li & Talwalkar, 2019; Lindauer & Hutter, 2019). In particular, we experiment on well-known search spaces and NAS pipelines, we run enough trials to reach statistical significance, and our implementation, including all details needed to reproduce our results, is available at `https://www.github.com/naszilla/bananas`.

Our contributions. We summarize our main contributions.

- We propose a novel path-based encoding for architectures. Using this featurization to predict the validation accuracy of architectures reduces the error by a factor of four, compared to the adjacency matrix encoding. We give theoretical and empirical results that the path encoding scales better than the adjacency matrix encoding.
- We develop BANANAS, a BO-based NAS algorithm which uses a meta neural network predictive uncertainty model defined on this path encoding. Our algorithm outperforms other state-of-the-art NAS methods on two search spaces.

2. Related Work

NAS has been studied since at least the 1990s and has gained significant attention in the past few years (Kitano, 1990; Stanley & Miikkulainen, 2002; Zoph & Le, 2017). Some of the most popular recent techniques for NAS include evolutionary algorithms (Shah et al., 2018; Maziarz et al., 2018), reinforcement learning (Zoph & Le, 2017; Pham et al., 2018; Liu et al., 2018a; Tan & Le, 2019), Bayesian optimization (BO) (Kandasamy et al., 2018b; Jin et al., 2018), and gradient descent (Liu et al., 2018b). Recent papers have highlighted the need for fair and reproducible NAS comparisons (Li & Talwalkar, 2019; Ying et al., 2019; Lindauer & Hutter, 2019). There are several works which predict the validation accuracy of neural networks (Deng et al., 2017; Istrate et al., 2019; Zhang et al., 2018), or the curve of validation accuracy with respect to training time (Klein et al., 2017; Domhan et al., 2015; Baker et al., 2017). A recent algorithm, AlphaX, uses a meta neural network to perform NAS (Wang et al., 2018). The search is progressive, and each iteration makes a small change to the current neural network, rather than choosing a completely new neural network. A few recent papers use graph neural networks (GNNs) to encode neural architectures in NAS (Shi et al., 2019; Zhang et al., 2019). Unlike the path encoding, these algorithms require re-training a GNN for each new dataset. For a survey of neural architecture search, see (Elsken et al., 2018). There is also prior work on using neural network models in BO for hyperparameter optimization (Snoek et al., 2015; Springenberg et al., 2016), The explicit goal of these papers is to improve the efficiency of Gaussian process-based BO from cubic to linear time, not to develop a different type of prediction model in order to improve the performance of BO with respect to the number of iterations.

Ensembles of neural networks is a popular approach for uncertainty estimates, shown in many settings to be more effective than all other methods such as Bayesian neural networks even for an ensemble of size five (Lakshminarayanan et al., 2017; Beluch et al., 2018; Choi et al., 2016; Snoek et al., 2019). We provide additional details and related work on NAS, BO, and architecture prediction in the appendix.

3. Preliminaries

In this section, we give a background on BO. In applications of BO for deep learning, the typical goal is to find a neural architecture and/or set of hyperparameters that lead to an optimal validation error. Formally, BO seeks to compute $a^* = \arg\min_{a \in A} f(a)$, where A is the search space, and $f(a)$ denotes the validation error of architecture a after training on a fixed dataset for a fixed number of epochs. In the standard BO setting, over a sequence of iterations, the results from all previous iterations are used to model the topology of $\{f(a)\}_{a \in A}$ using the posterior distribution of the model (often a GP). The next architecture is then chosen by optimizing an acquisition function such as expected improvement (EI) (Močkus, 1975), upper confidence bound (UCB) (Srinivas et al., 2009), or Thompson sampling (TS) (Thompson, 1933). These functions balance exploration with exploitation during the iterative search. The chosen architecture is then trained and used to update the model of $\{f(a)\}_{a \in A}$. Evaluating $f(a)$ in each iteration is the bottleneck of BO (since a neural network must be trained). To mitigate this, parallel BO methods typically output k architectures to train in each iteration instead of just one, so that the k architectures can be trained in parallel.

4. Methodology

In this section, we discuss all of the components of our NAS algorithm. First we describe our featurization of neural architectures. Next, we describle how to predict mean and uncertainty estimates using an ensemble of meta neural networks. Then, we describe our acquisition function and acquisition optimization strategy, and show how these pieces come together to form our final NAS algorithm. We finish by presenting theoretical results showing that our architecture encoding scales well.

Architecture featurization via the path encoding. Prior work aiming to encode or featurize neural networks have proposed using an adjacency matrix-based approach. The adjacency matrix encoding gives an arbitrary ordering to the nodes, and then gives a binary feature for an edge between node i and node j, for all i, j. Then a list of the operations at each node must also be included in the encoding. This is a challenging data structure for a NAS algorithm to interpret, because it relies on an arbitrary indexing of the nodes, and features are highly dependent on one another. For example, an edge from the input to node 2 is useless if there is no path from node 2 to the output. Also, even if there is an edge from node 2 to the output, it matters a lot whether node 2 is a convolution or a pooling operation. Ying et al. (2019) tested another encoding that is similar to the adjacency matrix encoding, but the features of each edge are continuous.

We introduce a novel encoding which we term a *path encoding*, and we show that it substantially increases the performance of our predictive uncertainty model. Each feature of the path encoding corresponds with a directed path from the input to the output of an architecture cell (for example: input→conv_1x1→conv_3x3→pool_3x3→output). To encode an architecture, we simply check which of the possible paths it contains (i.e. which paths are present in the cell), and write this as a binary vector. Therefore, we do not need to arrange the nodes in any particular order. Furthermore, the features are not nearly as dependent on one another as they are in the adjacency matrix encoding. The total number

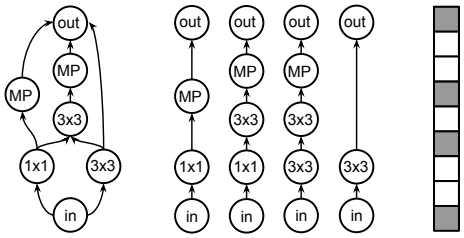

Figure 4.1: Example of our path encoding.

of paths is $\sum_{i=0}^{n} q^i$ where n denotes the number of nodes in the cell, and q denotes the number of operations for each node. See Figure 4.1. For example, the search space from NASBench (Ying et al., 2019) has $\sum_{i=0}^{5} 3^i = 364$ possible paths, and the search space from DARTS (Liu et al., 2018b) has $4 \cdot \sum_{i=0}^{4} 8^i = 18724$ possible paths. A downside of this encoding is that it scales exponentially in n, while the adjacency matrix encoding scales quadratically. At the end of this section, we give theoretical results giving evidence that simply truncating the path encoding allows it to scale *linearly* with a negligible decrease in performance. We back this up with experimental results in the next section.

Acquisition function and optimization. Many acquisition functions used in BO can be approximately computed using a mean and uncertainty estimate for each input datapoint. We train an ensemble of five meta neural networks in order to predict mean and uncertainty estimates for new neural architectures. Concretely, we predict the validation error, as well as confidence intervals around the predicted validation error, of neural architectures that we have not yet observed. Suppose we have an ensemble of M neural networks, $\{\hat{f}_m\}_{m=1}^{M}$, where $\hat{f}_m : A \to \mathbb{R}$ for all m. We optimize the following acquisition function, which we call *independent Thompson sampling* (ITS), which at each time step t is defined to be

$$\phi_{\text{ITS}}(a) = f_a(a), \quad \text{where} \quad f_a \sim p(f|\mathcal{D}_t). \tag{4.1}$$

Here, $\mathcal{D}_t = \{(a_0, f(a_0)), \ldots, (a_t, f(a_t))\}$ is the dataset of architecture evaluations at time t, and $p(f|\mathcal{D}_t)$ is our posterior belief about meta neural networks $f \in \mathcal{F}$ at time t given $\mathcal{D}_t$. Equation 4.1 can be viewed as defining the acquisition function equal to the output of a sample from the posterior distribution of our model. This is similar to classic Thompson sampling (TS) (Thompson, 1933), which has advantages when running parallel experiments in batch BO (Kandasamy et al., 2018a). However, in contrast with TS, the ITS acquisition function returns a unique posterior function sample f_a for each input architecture a. We choose not to use classic TS here for a few reasons: based on our ensemble model for predictive uncertainty, it is unclear how to draw exact posterior samples of functions f that can be evaluated on multiple architectures a, while we can develop procedures to sample from the posterior conditioned on a single input architecture a. One potential strategy to carry out classic TS is to use elements of our ensemble as approximate posterior samples, but this has not shown to perform well in practice. In the next section, we show empirically that ITS performs better than this strategy for TS as well as other acquisition functions.

In practice, in order to compute $\phi_{\text{ITS}}(a)$ given our ensemble of M meta neural networks, $\{\hat{f}_m\}_{m=1}^{M}$, we assume that the posterior conditioned on a given input architecture a follows a Gaussian distribution. In particular, we assume $p(f(a)|\mathcal{D}_t) = \mathcal{N}\left(\hat{f}(a), \hat{\sigma}(a)^2\right)$, with parameters $\hat{f}(a) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}_m(a)$ and $\hat{\sigma}(a)^2 = \sqrt{\frac{\sum_{m=1}^{M}(\hat{f}_m(a) - \hat{f}(a))^2}{M-1}}$. In each iteration of BO, our goal is to find the neural network from the search space which minimizes the acquisition function. Evaluating the acquisition function for every neural network in the search space is computationally infeasible. Instead, we optimize the acquisition function via a mutation procedure, in which we randomly mutate the best architectures that we have trained so far and then select the architecture from this set which minimizes the acquisition function. A neural architecture is mutated by either adding an edge, removing an edge, or changing one of the operations with some probability.

BANANAS: Bayesian optimization with neural architectures for NAS. Now we present our full NAS algorithm. At the start, we draw t_0 random architectures from the search space. Then we begin an iterative process, where in iteration t, we train an ensemble of meta neural networks on architectures $a_0, \ldots, a_{t-1}$. Each meta neural network is a feedforward network with fully-connected layers, and each is given a different random initialization of the weights and a different random ordering of the training set. We use a slight variant of mean absolute percentage error (MAPE),

$$\mathcal{L}(y_{\text{pred}}, y_{\text{true}}) = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_{\text{pred}}^{(i)} - y_{\text{LB}}}{y_{\text{true}}^{(i)} - y_{\text{LB}}} - 1\right|,$$

where $y_{\text{pred}}^{(i)}$ and $y_{\text{true}}^{(i)}$ are the predicted and true values of the validation error for architecture i, and y_{LB} is a global lower bound on the minimum true validation error. This loss function gives a higher weight to losses for architectures with smaller values of y_{true}.

We create a candidate set of architectures by mutating the best architectures we have seen so far, and we choose to train the candidate architecture which minimizes the ITS acquisition function. See Algorithm 1. To parallelize Algorithm 1, in step iv. we simply choose the k architectures with the smallest values of the acquisition function and evaluate the architectures in parallel.

Algorithm 1 BANANAS

Input: Search space A, dataset D, parameters t_0, T, M, c, x, acquisition function ϕ, function $f(a)$ returning validation error of a after training.

1. Draw t_0 architectures $a_0, \ldots, a_{t_0}$ uniformly at random from A and train them on D.
2. For t from t_0 to T,
   i. Train an ensemble of meta neural networks on $\{(a_0, f(a_0)), \ldots, (a_t, f(a_t))\}$.
   ii. Generate a set of c candidate architectures from A by randomly mutating the x architectures a from $\{a_0, \ldots, a_t\}$ that have the lowest value of $f(a)$.
   iii. For each candidate architecture a, evaluate the acquisition function $\phi(a)$.
   iv. Denote a_{t+1} as the candidate architecture with minimum $\phi(a)$, and evaluate $f(a_{t+1})$.

Output: $a^* = \text{argmin}_{t=0,\ldots,T} f(a_t)$.

Truncating the path encoding. In this section, we give theoretical results for a truncated version of the path encoding. One of the downsides of the path encoding is that it scales exponentially in n. However, the vast majority of paths rarely appear in any neural architecture during a full run of a NAS algorithm. This is because many NAS algorithms sample architectures from a random procedure (and/or mutate these samples). We show that the vast majority of paths have a very low probability of occurring in a cell returned from `random_spec()`, a popular random procedure (as in Ying et al. (2019) and used by BANANAS). Our results show that by simply truncating the least-likely paths, our encoding scales *linearly* in the size of the cell, with an arbitrarily small amount of information loss. We back this up with experimental evidence in the next section. For the full proofs, see Appendix C. We start by defining a random graph model corresponding to `random_spec()`.

Definition 4.1. 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.
(2) Label each node randomly with one of r operations.
(3) For all $i < j$, add edge (i, j) with probability $\frac{2k}{n(n-1)}$.
(4) If there is no path from node 1 to node n, `goto (1)`.

The probability value in step (3) is chosen so that the expected number of edges after this step is exactly k. In this section, we use 'path' to mean a path from node 1 to node n. We prove the following theorem.

Theorem 4.2 (informal)**.** *Given integers $r, c > 0$, there exists an N such that for all $n > N$, there exists a set of n paths $\mathcal{P}'$ such that the probability that $G_{n,n+c,r}$ contains a path not in $\mathcal{P}'$ is less than $\frac{1}{n^2}$.*

This theorem says that when $k = n + c$, and when n is large enough compared to c and r, then we can truncate the path encoding to a set $\mathcal{P}'$ of size n, because the probability that `random_spec()` returns a graph $G_{n,k,r}$ with a path outside of $\mathcal{P}'$ is very small.

We choose $\mathcal{P}'$ to be the set of paths with length less than $\log_r n$. Our argument relies on a simple concept: the probability of a long path (length $> \log_r n$) from node 1 to node n is much lower than the probability of a short path. For example, the probability that $G_{n,k,r}$ contains a path of length $n-1$ is on the order of $\frac{1}{n^{n-1}}$, because $n-1$ edges must be chosen, each with a probability of roughly $\frac{1}{n}$.

Proof sketch of Thorem 4.2. We set $\mathcal{P}'$ as the set of paths of length less than $\log_r n$, therefore,

$$|\mathcal{P}'| = 1 + r^1 + r^2 + \cdots + r^{\lceil \log_r n \rceil - 1} < n.$$

In Definition 4.1, the probability that step (3) returns a graph with a path from node 1 to node n is at least $\frac{1}{n}$, because the probability of a path of length 1 is the probability of having edge $(1, n)$, which is $\frac{2k}{n(n-1)} > \frac{1}{n}$ for large enough n.

In general, denote $a_{n,k,\ell}$ as the expected value of the number of paths of length ℓ after step (3). Then

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

We can prove that

$$\sum_{\ell=\log_r n}^{n-1} a_{n,k,\ell} \leq \left(\frac{1}{n}\right)^{\log_r \log_r n - 4} \leq \frac{1}{n^3},$$

using the well-known inequality $\binom{n}{\ell} \leq \left(\frac{en}{\ell}\right)^{\ell}$ (e.g. (Stanica, 2001)) and the fact that $(\log n)^{\log n} = n^{\log \log n}$. Then the probability that $G_{n,k,r}$ contains a path outside of $\mathcal{P}'$ is

$$P(\exists\ p \notin \mathcal{P}') = \frac{P(\exists \text{ path} \notin \mathcal{P}' \text{ after step (3)})}{P(\exists \text{ path after step (3)})} \leq \left(\frac{1}{n^3}\right) / \left(\frac{1}{n}\right) \leq \frac{1}{n^2}. \qquad \square$$

5. Experiments

In this section, we discuss our experimental setup and results. We give experimental results on the performance of the meta neural network itself, as well as the full NAS algorithm compared to several NAS algorithms.

We check every box in the NAS research checklist (Lindauer & Hutter, 2019). In particular, our code is publicly available, we used a tabular NAS dataset, and we ran many trials of each algorithm. In Appendix E, we give the full details of our answers to the NAS research checklist.

Search space from NASBench. The NASBench dataset is a tabular dataset designed to facilitate NAS research and fair comparisons between NAS algorithms (Ying et al., 2019). It consists of over 423,000 unique neural architectures from a cell-based search space, and each architecture comes with precomputed training, validation, and test accuracies for 108 epochs on CIFAR-10.

The search space consists of a cell with 7 nodes. The first node is the input, and the last node is the output. The remaining five nodes can be either 1×1 convolution, 3×3 convolution, or 3×3 max pooling. The cell can take on any DAG structure from the input to the output with at most 9 edges. The NASBench search space was chosen to contain ResNet-like and Inception-like cells (He et al., 2016; Szegedy et al., 2016). The hyper-architecture consists of nine cells stacked sequentially, with each set of three cells separated by downsampling layers. The first layer before the first cell is a convolutional layer, and the hyper-architecture ends with a global average pooling layer and dense layer.

Search space from DARTS. One of the most popular convolutional cell-based search spaces is the one from DARTS (Liu et al., 2018b), used for CIFAR-10. The search space consists of two cells with 6 nodes each: a convolutional cell and a reduction cell, and the hyper-architecture stacks the convolutional and reduction cells. For each cell, the first two nodes are the inputs from the previous two cells in the hyper-architecture. The next four nodes each contain exactly two edges as input, such that the cell forms a connected DAG. Each *edge* can take one of seven operations: 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max pooling, 3×3 average pooling, identity, and *zero* (this is in contrast to NASBench, where the *nodes* take on operations).

5.1. NASBench Experiments

First we evaluate the performance of the meta neural network on the NASBench dataset. The meta neural network consists of a sequential fully-connected neural network. The number of layers is set to 10, and each layer has width 20. We use the Adam optimizer with a learning rate of 0.01. For this set of experiments, we use mean absolute error as the loss function (MAE), as we are evaluating the performance of the meta neural network without running the full BANANAS algorithm. We test the standard adjacency matrix encoding as well as the path encoding discussed in Section 4. We draw architectures using the `random_spec()` method described in Section 4. The path encoding outperforms the adjacency matrix encoding by up to a factor of 4 with respect to mean absolute error. See Figure 5.1. In order to compare our meta neural network to similar work, we trained on 1100 NASBench architectures and computed the correlation between the predicted and true validation accuracies on a test set of size 1000. See Table 1. In Appendix D, we give a more detailed experimental study on the performance of our meta neural network.

Next, we evaluate the performance of BANANAS. We use the same meta neural network parameters as in the previous paragraph, but for the loss function we use MAPE as defined in Section 4. We baseline against random search (which multiple papers have concluded is a competitive baseline for NAS (Li & Talwalkar, 2019; Sciuto et al., 2019)). We compare to several state-of-the-art methods, representing the most common zeroth-order paradigms. Regularized Evolution is a popular evolutionary algorithm for NAS (Real et al., 2019). We compared to two reinforcement learning (RL) algorithms: REINFORCE (Williams, 1992), which prior work showed to be more effective than other RL algorithms (Ying et al., 2019), and AlphaX, which uses a neural network to select the best action in each round, such as making a small change to, or growing, the current architecture (Wang et al., 2018). We compared to several Bayesian approaches: vanilla Bayesian optimization with a Gaussian process prior; Bayesian optimization Hyperband (BOHB) (Falkner et al., 2018), which combines multi-fidelity Bayesian optimization with principled early-stopping; Deep Networks for Global Optimization (DNGO) (Snoek et al., 2015), which is an implementation of Bayesian optimization using adaptive basis regression using neural networks instead of Gaussian processes to avoid the cubic scaling; and neural architecture search with Bayesian optimization and optimal transport (NASBOT) (Kandasamy et al., 2018b), which uses an optimal transport-based distance function in BO. We also compared to Tree-structured Parzen estimator (TPE) (Bergstra et al., 2011), which is a hyperparameter optimization algorithm based on adaptive Parzen windows. See Appendix D for more details on the implementation of these algorithms.

Experimental setup and results. Each NAS algorithm is given a budget of 47 TPU hours, or about 150 queries. That is, each algorithm can train and output the validation error of at most 150 architectures. We chose this number of queries as it is a realistic setting in practice (on other search spaces such as DARTS, 150 queries takes 15 GPU days), though we also ran experiments to 156.7 TPU hours or 500 queries in Appendix D, which resulted in similar trends. Every 10 iterations, each algorithm returns the architecture with the best validation error so far. After all NAS algorithms have completed, we output the test error for each returned architecture. We ran 200 trials for each algorithm. See Figure 5.2. BANANAS significantly outperforms all other baselines, and to the best of our knowledge, BANANAS achieves state-of-the-art error on NASBench in the 100-150 queries setting. The standard deviations among 200 trials for all NAS algorithms were between 0.16 and 0.22. BANANAS had the lowest standard deviation, and

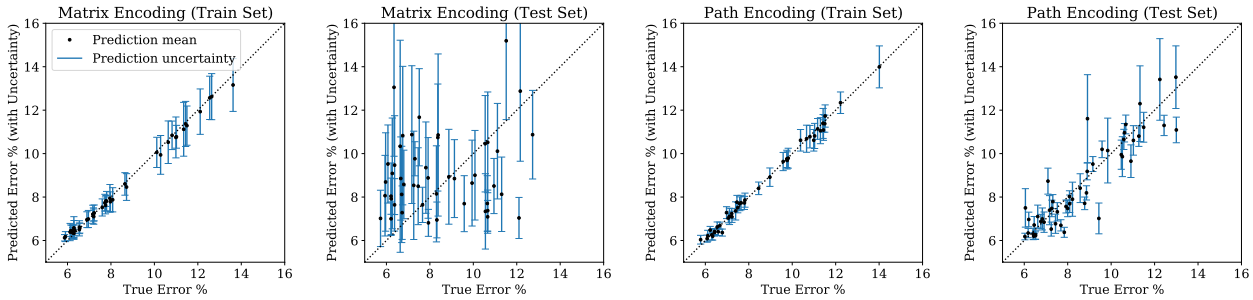

Figure 5.1: Predictive uncertainty estimates for architecture validation error under our ensemble model. We train this model on 200 architectures drawn from `random_spec()`, and test the adjacency matrix encoding and path encoding on a held-out set of architectures as well as on a subset of the training set.

Table 1: Performance of neural architecture predictors trained on NASBench. The truncated path encoding is truncated to a length of $3^0 + 3^1 + 3^2 + 3^3 = 40$.

Architecture	Source	Correlation	No. of parameters
Multilayer Perceptron	(Wang et al., 2018)	0.400	6,326,000
Long Short-Term Memory (LSTM)	(Wang et al., 2018)	0.460	92,000
Graph Convolutional Network	(Shi et al., 2019)	0.607	14,000
Meta NN with Adjacency Encoding	Ours	0.212	**4,400**
Meta NN with Truncated Path Encoding	Ours	0.668	4,700
Meta NN with Full Path Encoding	Ours	**0.699**	11,000

regularized evolution had the highest.

Ablation study. Our NAS algorithm has several moving parts, including the meta neural network model, the path-based feature encoding of neural architectures, the acquisition function, and the acquisition optimization strategy. We run a thorough ablation study by removing each piece of the algorithm separately. In particular, we compare against the following algorithms: *(1)* BANANAS in which the acquisition funtion is optimized by drawing 1000 random architectures instead of a mutation algorithm; BANANAS in which the featurization of each architecture is not the path-based encoding, but instead *(2)* the adjacency matrix encoding, or *(3)* the continuous adjacency matrix encoding from (Ying et al., 2019); BANANAS with a GP model instead of a neural network model, where the distance function in the GP is computed as the Hamming distance between *(4)* the path encoding, or *(5)* adjacency matrix encoding. We found that BANANAS distinctly outperformed all variants, with the meta neural network having the greatest effect on performance. See Figure 5.2 (middle). In Appendix D, we show a separate study testing five different acquisition functions.

Dual-objective experiments. One of the benefits of a zeroth order NAS algorithm is that it allows for optimization with respect to additional non-differentiable objectives. For example, we define and optimize a dual-objective function of both validation loss and number of model parameters. Specifically, we use

$$\mathcal{L}(a) = (\text{val\_loss}(a) - 4.8) \cdot (\#\text{\_params}(a))^{0.5}, \quad (5.1)$$

which is similar to prior work (Cai et al., 2019; Tan et al., 2019), where 4.8 is a lower bound on the minimum validation loss. We run 200 trials of random search, Regularized Evolution, and BANANAS on this objective. We plot the average test loss and the number of parameters of the returned models after 10 to 150 queries. See Figure 5.2 (right). The gray lines are contour lines of Equation 5.1.

In Appendix D, we present experiments for BANANAS in other settings: 500 queries instead of 150 queries, and using random validation error instead of mean validation error. We show the trends are largely the same and BANANAS performs the best. We also study the effect of the length of the path encoding on the performance of both the meta neural network, and BANANAS as a whole. That is, we truncate the path enoding vector as in Section 4. Surprisingly, the length of the path encoding can be reduced by an order of magnitude with a negligible decrease in performance.

5.2. DARTS search space experiments.

We test the BANANAS algorithm on the search space from DARTS. We give BANANAS a budget of 100 queries. In each query, a neural network is trained for 50 epochs and the average validation error of the last 5 epochs is recorded. As

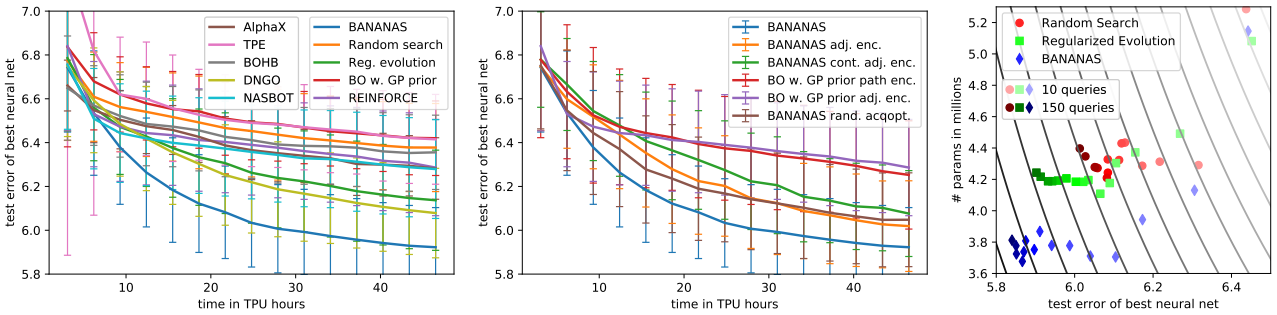

Figure 5.2: Performance of BANANAS on NASBench compared to other algorithms (left). Ablation study (middle). NAS dual-objective experiments minimizing a function of validation loss and number of model parameters (right).

Table 2: Comparison of the mean test error of the best architectures returned by three NAS algorithms. The runtime is in total GPU-days on a Tesla V100. Note that ASHA queries use varying numbers of epochs.

NAS Algorithm	Source	Test error		Queries	Runtime	Method
		Avg	Best			
SNAS	(Xie et al., 2018)	2.85			1.5	Gradient based
ENAS	(Pham et al., 2018)		2.89		0.5	RL
Random search	(Liu et al., 2018b)	3.29			4	Random
DARTS	(Liu et al., 2018b)	2.76			5	Gradient-based
ASHA	(Li & Talwalkar, 2019)	3.03	2.85	700	9	Successive halving
Random search WS	(Li & Talwalkar, 2019)	2.85	2.71	1000	9.7	Random
DARTS	Ours	2.68	**2.57**		5	Gradient-based
ASHA	Ours	3.08	2.98	700	9	Successive halving
BANANAS	Ours	**2.64**	**2.57**	100	11.8	Neural BayesOpt

in the NASBench experiments, we parallelize the algorithm by choosing 10 neural architectures to train in each iteration of Bayesian optimization using ITS. We use the same meta neural net architecture as in the NASBench experiment, but we change the learning rate to 10^{-4} and the number of epochs to 10^4. In Appendix D, we show the best architecture found by BANANAS. The algorithm requires 11.8 GPU days of computation to run.

To ensure a fair comparison by controlling all hyperparameter settings and hardware, we re-trained the architectures from papers which used the DARTS search space and reported the final architecture. We report the mean test error over five random seeds of the best architectures found by BANANAS, DARTS, and ASHA (Li & Talwalkar, 2019). We trained each architecture using the default hyperparameters from Li & Talwalkar (2019).

The BANANAS architecture achieved an average of 2.64% error, which is state-of-the-art for this search-space and final training parameter settings. See Table 2. We also compare to other NAS algorithms using the DARTS search space, however, the comparison is not perfect due to differences in the pytorch version of the final training code (which appears to change the final percent error by $\sim$0.1%). We cannot fairly compare our method to recent NAS algorithms which use a larger search space than DARTS, or which train the final architecture for significantly more than 600 epochs (Laube & Zell, 2019; Liang et al., 2019; Zhou et al., 2019). Our algorithm significantly beats ASHA, a multi-fidelity zeroth order NAS algorithm, and is on par with DARTS, a first-order method. We emphasize that since BANANAS is a zeroth order method, it allows for easy parallelism, integration with optimizing other hyperparameters, easy implementation, and optimization with respect to other non-differentiable objectives. [2]

6. Conclusion and Future Work

In this work, we propose a new method for NAS. Using novel techniques such as an architecture path encoding and meta neural network for predictive uncertainty estimates, our Bayesian optimization-based NAS algorithm achieves

[2] We also ran four trails of BANANAS on an older version (UCB instead of ITS and MAE instead of MAPE) and the average final error was within 0.1% of DARTS). The difference in final error between different runs of BANANAS was also within 0.1%.

state-of-the-art results. We present results on the NASBench and DARTS search spaces. Our path encoding may be of independent interest, as it substantially improves the predictive power of our meta neural network, and we give theoretical and empirical results showing that it scales better than existing encoding techniques. An interesting follow-up idea is to develop a multi-fidelity version of BANANAS. For example, incorporating a successive-halving approach to BANANAS could result in a significant decrease in the runtime without substantially sacrificing accuracy.

Acknowledgments

We thank Naveen Sundar Govindarajulu, Liam Li, Jeff Schneider, Sam Nolen, and Mark Rogers for their help with this project.

References

Baker, B., Gupta, O., Raskar, R., and Naik, N. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

Beluch, W. H., Genewein, T., Nürnberger, A., and Köhler, J. M. The power of ensembles for active learning in image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2011.

Cai, H., Zhu, L., and Han, S. Proxylessnas: Direct neural architecture search on target task and hardware. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

Choi, Y., Kwon, Y., Lee, H., Kim, B. J., Paik, M. C., and Won, J.-H. Ensemble of deep convolutional neural networks for prognosis of ischemic stroke. In *International Workshop on Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*, 2016.

Deng, B., Yan, J., and Lin, D. Peephole: Predicting network performance before training. *arXiv preprint arXiv:1712.03351*, 2017.

Domhan, T., Springenberg, J. T., and Hutter, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

Floreano, D., Dürr, P., and Mattiussi, C. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1(1):47–62, 2008.

Frazier, P. I. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1487–1495. ACM, 2017.

González, J., Dai, Z., Hennig, P., and Lawrence, N. Batch bayesian optimization via local penalization. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Istrate, R., Scheidegger, F., Mariani, G., Nikolopoulos, D., Bekas, C., and Malossi, A. C. I. Tapas: Train-less accuracy predictor for architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

Jin, H., Song, Q., and Hu, X. Auto-keras: Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282*, 2018.

Kandasamy, K., Krishnamurthy, A., Schneider, J., and Póczos, B. Parallelised bayesian optimisation via thompson sampling. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018a.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. P. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pp. 2016–2025, 2018b.

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

Klein, A., Falkner, S., Springenberg, J. T., and Hutter, F. Learning curve prediction with bayesian neural networks. *ICLR 2017*, 2017.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.

Kushner, H. J. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1):97–106, 1964.

Lakshminarayanan, B., Pritzel, A., and Blundell, C. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, pp. 6402–6413, 2017.

Laube, K. A. and Zell, A. Prune and replace nas. *arXiv preprint arXiv:1906.07528*, 2019.

Li, L. and Talwalkar, A. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Liang, H., Zhang, S., Sun, J., He, X., Huang, W., Zhuang, K., and Li, Z. Darts+: Improved differentiable architecture search with early stopping. *arXiv preprint arXiv:1909.06035*, 2019.

Lindauer, M. and Hutter, F. Best practices for scientific research on neural architecture search. *arXiv preprint arXiv:1909.02453*, 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 *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 19–34, 2018a.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018b.

Maziarz, K., Khorlin, A., de Laroussilhe, Q., and Gesmundo, A. Evolutionary-neural hybrid agents for architecture search. *arXiv preprint arXiv:1811.09828*, 2018.

Močkus, J. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pp. 400–404. Springer, 1975.

Neiswanger, W., Kandasamy, K., Poczos, B., Schneider, J., and Xing, E. Probo: a framework for using probabilistic programming in bayesian optimization. *arXiv preprint arXiv:1901.11515*, 2019.

Očenášek, J. and Schwarz, J. The parallel bayesian optimization algorithm. In *The State of the Art in Computational Intelligence*. Springer, 2000.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Rasmussen, C. E. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pp. 63–71. Springer, 2003.

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

Sciuto, C., Yu, K., Jaggi, M., Musat, C., and Salzmann, M. Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*, 2019.

Shah, S. A. R., Wu, W., Lu, Q., Zhang, L., Sasidharan, S., DeMar, P., Guok, C., Macauley, J., Pouyoul, E., Kim, J., et al. Amoebanet: An sdn-enabled network service for big data science. *Journal of Network and Computer Applications*, 119:70–82, 2018.

Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J. T., and Zhang, T. Multi-objective neural architecture search via predictive network performance optimization. *arXiv preprint arXiv:1911.09336*, 2019.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.

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 *International conference on machine learning*, pp. 2171–2180, 2015.

Snoek, J., Ovadia, Y., Fertig, E., Lakshminarayanan, B., Nowozin, S., Sculley, D., Dillon, J., Ren, J., and Nado, Z. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2019.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. Bayesian optimization with robust bayesian neural networks. In *Advances in Neural Information Processing Systems*, pp. 4134–4142, 2016.

Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

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

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

Thompson, W. R. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Wang, L., Zhao, Y., Jinnai, Y., and Fonseca, R. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1805.07440*, 2018.

Wang, L., Xie, S., Li, T., Fonseca, R., and Tian, Y. Sample-efficient neural architecture search by learning action space. *arXiv preprint arXiv:1906.06832*, 2019.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, pp. 229–256, 1992.

Xie, S., Zheng, H., Liu, C., and Lin, L. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

Ying, C., Klein, A., Real, E., Christiansen, E., Murphy, K., and Hutter, F. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

Zhang, C., Ren, M., and Urtasun, R. Graph hypernetworks for neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

Zhang, M., Jiang, S., Cui, Z., Garnett, R., and Chen, Y. D-vae: A variational autoencoder for directed acyclic graphs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2019.

Zhou, H., Yang, M., Wang, J., and Pan, W. Bayesnas: A bayesian approach for neural architecture search. *arXiv preprint arXiv:1905.04919*, 2019.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

A. Related Work Continued

Neural architecture search. Neural architecture search has been studied since at least the 1990s (Floreano et al., 2008; Kitano, 1990; Stanley & Miikkulainen, 2002), but the field was revitalized in 2017 when the work of Zoph & Le (2017) gained significant attention. Some of the most popular techniques for NAS include evolutionary algorithms (Shah et al., 2018; Maziarz et al., 2018), reinforcement learning (Zoph & Le, 2017; Pham et al., 2018; Liu et al., 2018a; Tan & Le, 2019; Wang et al., 2019), Bayesian optimization (Kandasamy et al., 2018b; Jin et al., 2018; Zhou et al., 2019), and gradient descent (Liu et al., 2018b; Liang et al., 2019; Laube & Zell, 2019). See Elsken et al. (2018) for a survey on NAS.

Recent papers have called for fair and reproducible experiments in the future (Li & Talwalkar, 2019; Ying et al., 2019). In this vein, the NASBench dataset was created, which contains over 400k neural architectures with precomputed training, validation, and test accuracy (Ying et al., 2019). A recent algorithm, AlphaX, uses a meta neural network to perform NAS (Wang et al., 2018), where the meta neural network is trained to make small changes to the current architecture, given adjacency matrix featurizations of neural architectures. The search is progressive, and each iteration makes a small change to the current neural network, rather than choosing a completely new neural network.

A few recent papers use graph neural networks (GNNs) to encode neural architectures in NAS (Shi et al., 2019; Zhang et al., 2019). However, GNNs require re-training for each new dataset/search space, unlike the path encoding which is a fixed encoding for every search space. Shi et al. (2019) has experiments on NASBench, however, they only report the number of queries needed to find the optimal architecture (which is roughly 1500), and no code is provided, so it is not clear how well their algorithm performs with a budget of fewer queries.

Bayesian optimization. Bayesian optimization is a leading technique for zeroth order optimization when function queries are expensive (Rasmussen, 2003; Frazier, 2018), and it has seen great success in hyperparameter optimization for deep learning (Rasmussen, 2003; Golovin et al., 2017; Li et al., 2016). The majority of Bayesian optimization literature has focused on Euclidean or categorical input domains, and has used a GP model (Rasmussen, 2003; Golovin et al., 2017; Frazier, 2018; Snoek et al., 2012). There are techniques for parallelizing Bayesian optimization (González et al., 2016; Kandasamy et al., 2018a; Očenášek & Schwarz, 2000). There is also prior work on using neural network models in Bayesian optimization for hyperparameter optimization (Snoek et al., 2015; Springenberg et al., 2016), The goal of these papers is to improve the efficiency of Gaussian Process-based Bayesian optimization from cubic to linear time, not to develop a different type of prediction model in order to improve the performance of BO with respect to the number of iterations. In our work, we present new techniques which deviate from Gaussian Process-based Bayesian optimization and see a large performance boost with respect to the number of iterations.

Predicting neural network accuracy. There are several approaches for predicting the validation accuracy of neural networks, such as a layer-wise encoding of neural networks with an LSTM algorithm (Deng et al., 2017), and a layer-wise encoding and dataset features to predict the accuracy for neural network and dataset pairs (Istrate et al., 2019). There is also work in predicting the learning curve of neural networks for hyperparameter optimization (Klein et al., 2017; Domhan et al., 2015) or NAS (Baker et al., 2017) using Bayesian techniques. None of these methods have predicted the accuracy of neural networks drawn from a cell-based DAG search space such as NASBench or the DARTS search space. Another recent work uses a hypernetwork for neural network prediction in NAS (Zhang et al., 2018).

Ensembling of neural networks is a popular approach for uncertainty estimates, shown in many settings to be more effective than all other methods such as Bayesian neural networks even for an ensemble of size five (Lakshminarayanan et al., 2017; Beluch et al., 2018; Choi et al., 2016; Snoek et al., 2019).

B. Preliminaries Continued

We give background information on three key ingredients of NAS algorithms.

Search space. Before deploying a NAS algorithm, we must define the space of neural networks that the algorithm can search through. Perhaps the most common type of search space for NAS is a *cell-based search space* (Zoph & Le, 2017; Pham et al., 2018; Liu et al., 2018b; Li & Talwalkar, 2019; Sciuto et al., 2019; Ying et al., 2019). A *cell* consists of a relatively small section of a neural network, usually 6-12 nodes forming a directed acyclic graph (DAG). A neural architecture is then built by repeatedly stacking one or two different cells on top of each other sequentially, possibly separated by specialized layers. The layout of cells and specialized layers is called a *hyper-architecture*, and this is fixed, while the NAS algorithm searches for the best cells. The search space over cells consists of all possible DAGs of a certain size, where each node can be one of several operations such as 1×1 convolution, 3×3 convolution, or 3×3 max pooling. It is also common to set a restriction on the number of total edges or the in-degree of each node (Ying et al., 2019; Liu et al., 2018b). In this work, we focus

on NAS over convolutional cell-based search spaces, though our method can be applied more broadly.

Search strategy. The search strategy is the optimization method that the algorithm uses to find the optimal or near-optimal neural architecture from the search space. There are many varied search strategies, such as Bayesian optimization, evolutionary search, reinforcement learning, and gradient descent. In Section 4, we introduced a novel search strategy based on Bayesian optimization with a neural network model using a path-based encoding.

Evaluation method. Many types of NAS algorithms consist of an iterative framework in which the algorithm chooses a neural network to train, computes its validation error, and uses this result to guide the choice of neural network in the next iteration. The simplest instantiation of this approach is to train each neural network in a fixed way, i.e., the algorithm has black-box access to a function that trains a neural network for x epochs and then returns the validation error. Algorithms with black-box evaluation methods can be compared by returning the architecture with the lowest validation error after a certain number of queries to the black-box function. There are also multi-fidelity methods, for example, when a NAS algorithm chooses the number of training epochs in addition to the architecture.

C. Details from Section 4

In this section, we give the full details from Section 4.

Recall that the size of the path encoding is equal to the number of unique paths, which is $\sum_{i=0^n} r^i$, where n is the number of nodes in the cell, and r is the number of operations to choose from at each node. This is at least r^n. By contrast, the adjacency matrix encoding scales quadratically in n.

However, the vast majority of the paths rarely show up in any neural architecture throughout a full run of a NAS algorithm. This is because many NAS algorithms can only sample architectures from a random procedure or mutate architectures drawn from the random procedure. Now we will give the full details of Theorem 4.2, showing that the vast majority of paths have a very low probability of occurring in a cell outputted from `random_spec()`, a popular random procedure (as in Ying et al. (2019) and used by BANANAS). We backed this up with experimental evidence in Section 5. Our results show that by simply truncating the least-likely paths, our encoding scales *linearly* in the size of the cell, with an arbitrarily small amount of information loss.

For convenience, we restate the random graph model defined in Section 4.

Definition 4.1. 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.
(2) Label each node randomly with one of r operations.
(3) For all $i < j$, add edge (i, j) with probability $\frac{2k}{n(n-1)}$.
(4) If there is no path from node 1 to node n, `goto (1)`.

The probability value in step (3) is chosen so that the expected number of edges after this step is exactly k. This definition is slightly different from `random_spec()`. In `random_spec()`, edges are added with probability $1/2$, and a cell is rejected if there are greater than 9 edges (in addition to being rejected if there is no path from node 1 to node n).

Recall that we use 'path' to mean a path from node 1 to node n. We restate the theorem formally. Denote $\mathcal{P}$ as the set of all possible paths from node 1 to node n that could occur in $G_{n,k,r}$.

Theorem 4.2 (formal). *Given integers $r, c > 0$, there exists N such that for all $n > N$, there exists a set of n paths $\mathcal{P}' \subseteq \mathcal{P}$ such that*

$$P(\exists p \in G_{n,n+c,r} \cap \mathcal{P} \setminus \mathcal{P}') \leq \frac{1}{n^2}.$$

This theorem says that when $k = n + c$, and when n is large enough compared to c and r, then we can truncate the path encoding to a set $\mathcal{P}'$ of size n, because the probability that `random_spec()` outputs a graph $G_{n,k,r}$ with a path outside of $\mathcal{P}'$ is very small.

Note that there are two caveats to this theorem. First, BANANAS may mutate architectures drawn from Definition 4.1, and Theorem 4.2 does not show the probability of paths from mutated architectures is small. However, our experiments in the next section give evidence that the mutated architectures do not change the distribution of paths too much. Second, the most common paths in Definition 4.1 are not necessarily the paths whose existence or non-existence give the most entropy in predicting the validation accuracy of a neural architecture. Again, while this is technically true, our experiments back up Theorem 4.2 as a reasonable argument that truncating the path encoding does not sacrifice performance.

Denote by $G'_{n,k,r}$ the random graph outputted by Definition 4.1 without step (4). In other words, $G'_{n,k,r}$ is a random graph that could have no path from node 1 to node n. Since there are $\frac{n(n-1)}{2}$ pairs (i, j) such that $i < j$, the expected number of edges of $G'_{n,k,r}$ is k. For reference, in the NAS-Bench dataset, there are $n = 7$ nodes and $r = 3$ operations, and the maximum number of edges is 9.

We choose $\mathcal{P}'$ as the n shortest paths from node 1 to node n. The argument for Theorem 4.2 relies on a simple concept: the probability that $G_{n,k,r}$ contains a long path (length >

$\log_r n$) is much lower than the probability that it contains a short path. For example, the probability that $G'_{n,k,r}$ contains a path of length $n-1$ is very low, because there are $\Theta(n^2)$ potential edges but the expected number of edges is $n+O(1)$. We start by upper bounding the length of the n shortest paths.

Lemma C.1. *Given a graph with n nodes and r node labels, there are fewer than n paths of length less than or equal to $\log_r n - 1$.*

Proof. The number of paths of length ℓ is r^ℓ, since there are r choices of labels for each node. Then

$$1 + r + \cdots + r^{\lceil \log_r n \rceil - 1} = \frac{r^{\lceil \log_r n \rceil} - 1}{r-1} = \frac{n-1}{r-1} < n.$$

□

To continue our argument, we will need the following well-known bounds on binomial coefficients, e.g. (Stanica, 2001).

Theorem C.2. *Given $0 \le \ell \le n$, we have*

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

Now we 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,

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

The following lemma, which is the driving force behind Theorem 4.2, shows that the value of $a_{n,k,\ell}$ for small ℓ is much larger than the value of $a_{n,k,\ell}$ for large ℓ.

Lemma C.3. *Given integers $r, c > 0$, then there exists n such that for $k = n + c$, we have*

$$\sum_{\ell=\log_r n}^{n-1} a_{n,k,\ell} < \frac{1}{n^3} \text{ and } a_{n,k,1} > \frac{1}{n}.$$

Proof. We have that

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

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 $\frac{2}{n(n-1)} \cdot k$.

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

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

for sufficiently large n. Now we will derive an upper bound for $a_{n,k,\ell}$ using Theorem C.2.

$$\begin{aligned}
a_{n,k,\ell} &= \binom{n-2}{\ell-1}\left(\frac{2k}{n(n-1)}\right)^\ell \\
&\le \left(\frac{e(n-2)}{\ell-1}\right)^{\ell-1}\left(\frac{2k}{n(n-1)}\right)^\ell \\
&\le \left(\frac{2k}{n(n-1)}\right)\left(\frac{2ek(n-2)}{(\ell-1)n(n-1)}\right)^{\ell-1} \\
&\le \left(\frac{4}{n}\right)\left(\frac{4e}{\ell-1}\right)^{\ell-1}
\end{aligned}$$

The last inequality is true because $k/(n-1) = (n+c)/(n-1) \le 2$ for sufficiently large n. Now we have

$$\begin{aligned}
\sum_{\ell=\log_r n}^{n-1} a_{n,k,\ell} &\le \sum_{\ell=\log_r n}^{n-1}\left(\frac{4}{n}\right)\left(\frac{4e}{\ell-1}\right)^{\ell-1} \\
&\le \sum_{\ell=\log_r n}^{n-1}\left(\frac{4e}{\ell-1}\right)^{\ell-1} \\
&\le \sum_{\ell=\log_r n}^{n-1}\left(\frac{4e}{\log_r n}\right)^{\ell-1} \\
&\le \left(\frac{4e}{\log_r n}\right)^{\log_r n}\sum_{\ell=0}^{n-\log_r n}\left(\frac{4e}{\log_r n}\right)^{\ell} \\
&\le (e)^{3\log_r n}\left(\frac{1}{\log_r n}\right)^{\log_r n} \cdot 2 \qquad \text{(C.1)} \\
&\le 2\,(n)^3\left(\frac{1}{n}\right)^{\log_r \log_r n} \qquad \text{(C.2)} \\
&\le \left(\frac{1}{n}\right)^{\log_r \log_r n - 4} \\
&\le \frac{1}{n^3}.
\end{aligned}$$

In inequality C.1, we use the fact that for large enough n, $\frac{4e}{\log_r n} < \frac{1}{2}$, therefore,

$$\sum_{\ell=0}^{n-\log_r n}\left(\frac{4e}{\log_r n}\right)^{\ell} \le \sum_{\ell=0}^{n-\log_r n}\left(\frac{1}{2}\right)^{\ell} \le 2$$

In inequality C.2, we use the fact that

$$\begin{aligned}
(\log n)^{\log n} &= \left(e^{\log\log n}\right)^{\log n} = \left(e^{\log n}\right)^{\log\log n} \\
&= n^{\log\log n}
\end{aligned}$$

□

Now we can prove Theorem 4.2.

Proof of Theorem 4.2. Recall that $\mathcal{P}$ denotes the set of all possible paths from node 1 to node n that could be present in $G_{n,k,r}$, and let $\mathcal{P}' = \{p \mid |p| < \log_r n - 1\}$. Then by Lemma C.1, $|\mathcal{P}| < n$. In Definition 4.1, the probability that we return a graph in step (4) is at least the probability that there exists an edge from node 1 to node n. This probability is $\geq \frac{1}{n}$ from Lemma C.3. Now we will compute the probability that there exists a path in $\mathcal{P} \setminus \mathcal{P}'$ in $G_{n,k,r}$ by conditioning on returning a graph in step (4). The penultimate inequality is due to Lemma C.3.

$$
\begin{aligned}
&P(\exists p \in G_{n,k,r} \cap \mathcal{P} \setminus \mathcal{P}') \\
&= P(\exists p \in G'_{n,k,r} \cap \mathcal{P} \setminus \mathcal{P}' \mid \exists q \in G'_{n,k,r} \cap \mathcal{P}) \\
&= \frac{P(\exists p \in G'_{n,k,r} \cap \mathcal{P} \setminus \mathcal{P}')}{P(\exists q \in G'_{n,k,r} \cap \mathcal{P})} \\
&\leq \left(\frac{1}{n^3}\right) / \left(\frac{1}{n}\right) \leq \frac{1}{n^2}
\end{aligned}
$$

□

D. Additional Experiments and Details

In this section, we present details from Section 5 and more experiments for BANANAS. In Section D.2, we evaluate the meta neural network used by BANANAS with different training set sizes. In Section D.3, we evaluate BANANAS on NASBench with five different acquisition functions. In Sections D.4 and D.5, we evaluate BANANAS on NASBench in settings different from Section 5: 500 queries instead of 150 queries, and using random validation error instead of mean validation error. Finally, in Section D.6, we study the effect of the length of the path encoding on the performance of both the meta neural network, and BANANAS as a whole.

D.1. Details from Section 5

Here, we give more details of the NAS algorithms we compared in Section 5.

Random search. The simplest baseline, random search, draws n architectures at random and outputs the architecture with the lowest validation error. Despite its simplicity, multiple papers have concluded that random search is a competitive baseline for NAS algorithms (Li & Talwalkar, 2019; Sciuto et al., 2019).

Regularized evolution. This algorithm consists of iteratively mutating the best achitectures out of a sample of all architectures evaluated so far (Real et al., 2019). We used the same hyperparameters as in the NASBench implementation, but changed the population size from 50 to 30 to account for fewer total queries.

Reinforcement Learning. We use the NASBench implementation of reinforcement learning for NAS based on the REINFORCE algorithm (Williams, 1992). We used this algorithm because prior work has shown that a 1-layer LSTM controller trained with PPO is not effective on the NASBench dataset (Ying et al., 2019).

Bayesian optimization with a GP model. We set up Bayesian optimization with a Gaussian process model and UCB acquisition. In the Gaussian process, we set the distance function between two neural networks as the sum of the Hamming distances between the adjacency matrices and the list of operations. Note that we also used a path-based distance function in our ablation study. We use the ProBO implementation (Neiswanger et al., 2019).

AlphaX. AlphaX casts NAS as a reinforcement learning problem, using a neural network to guide the search (Wang et al., 2018). Each iteration, a neural network is trained to select the best action, such as making a small change to, or growing, the current architecture.

TPE. Tree-structured Parzen estimator (TPE) is a hyperparameter optimization algorithm based on adaptive Parzen windows. We use the NASBench implementation.

BOHB. Bayesian Optimization HyperBand (BOHB) combines multi-fidelity Bayesian optimization with principled early-stopping from Hyperband (Falkner et al., 2018). We use the NASBench implementation.

DNGO. Deep Networks for Global Optimization (DNGO) is an implementation of Bayesian optimization using adaptive basis regression using neural networks instead of Gaussian processes to avoid the cubic scaling (Snoek et al., 2015).

NASBOT. Neural architecture search with Bayesian optimization and optimal transport (NASBOT) (Kandasamy et al., 2018b) works by defining a distance function between neural networks by computing the similarities between layers and then running an optimal transport algorithm to find the minimum earth-mover's distance between the two architectures. Then Bayesian optimization is run using this distance function. The NASBOT algorithm is specific to macro NAS, and we put in a good-faith effort to implement it in the cell-based setting. Specifically, we compute the distance between two cells by taking the earth-mover's distance between the set of row-sums, column-sums, and node operations. This is a version of the OTMANN distance defined in Section 3 and Table 1 in Kandasamy et al. (2018b), defined for the cell-based setting.

Additional notes from Section 5. In the main NASBench experiments and the ablation study, Figure 5.2, we added an isomorphism-removing subroutine to any algorithm that uses the adjacency matrix encoding. This is because multiple adjacency matrices can map to the same architecture. With the path encoding, this is not necessary. Note that without the isomorphism-removing subroutine, some algorithms

perform significantly worse, e.g. BANANAS with the adjacency matrix encoding performs almost as bad as random search. This is another strength of the path encoding.

In the main plot, DNGO was run with the adjacency matrix encoding. We also ran DNGO with the path encoding, and it performed similarly to DNGO with the adjacency matrix encoding. We also ran DNGO with the objective in Equation 5.1. It performed similar to regularized evolution (worse than BANANAS). We also ran NASBOT (Kandasamy et al., 2018b), which performed better than BO with the adjacency encoding but worse than BO with the path encoding.

In Section 5, we described the details of running BANANAS on the DARTS search space, which resulted in an architecture. We show this architecture in Figure D.1.

D.2. Meta Neural Network Experiments

We plot the meta neural network performance with both featurizations for training sets of size 20, 100, and 500. We use the same experimental setup as described in Section 5. See Figure D.2. We note that a training set of size 20 neural networks is realistic at the start of a NAS algorithm, and size 200 is realistic near the middle or end of a NAS algorithm.

D.3. Acquisition functions.

We tested BANANAS with four other common acquisition functions in addition to ITS: expected improvement (EI) (Močkus, 1975), probability of improvement (PI) (Kushner, 1964), Thompson sampling (TS) (Thompson, 1933), and upper confidence bound (UCB) (Srinivas et al., 2009). First we give the formal definitions of each acquisition function.

Suppose we have trained a collection of M predictive models, $\{f_m\}_{m=1}^M$, where $f_m : A \to \mathbb{R}$. Following previous work (Neiswanger et al., 2019), we use the following acquisition function estimates for an input architecture $a \in A$:

$$\phi_{\text{EI}}(a) = \mathbb{E}\left[\mathbb{1}\left[f_m(a) > y_{\text{min}}\right](y_{\text{min}} - f_m(a))\right] \quad \text{(D.1)}$$
$$= \int_{-\infty}^{y_{\text{min}}} (y_{\text{min}} - y)\,\mathcal{N}\left(\hat{f}, \hat{\sigma}^2\right) dy$$
$$\phi_{\text{PI}}(x) = \mathbb{E}\left[\mathbb{1}\left[f_m(x) > y_{\text{min}}\right]\right] \quad \text{(D.2)}$$
$$= \int_{-\infty}^{y_{\text{min}}} \mathcal{N}\left(\hat{f}, \hat{\sigma}^2\right) dy$$
$$\phi_{\text{UCB}}(x) = \text{UCB}\left(\{f_m(x)\}_{m=1}^M\right) = \hat{f} - \beta\hat{\sigma} \quad \text{(D.3)}$$
$$\phi_{\text{TS}}(x) = f_{\tilde{m}}(x), \quad \tilde{m} \sim \text{Unif}\,(1, M) \quad \text{(D.4)}$$
$$\phi_{\text{ITS}}(x) = \tilde{f}_x(x), \quad \tilde{f}_x(x) \sim \mathcal{N}(\hat{f}, \hat{\sigma}^2) \quad \text{(D.5)}$$

In these acquisition function definitions, $\mathbb{1}(x) = 1$ if x is true and 0 otherwise, and we are making a normal approximation for our model's posterior predictive density, where we estimate parameters

$$\hat{f} = \frac{1}{M}\sum_{m=1}^{M} f_m(x), \text{ and } \hat{\sigma}^2 = \sqrt{\frac{\sum_{m=1}^{M}(f_m(x) - \hat{f})^2}{M-1}}.$$

In the UCB acquisition function, we use UCB to denote some estimate of a lower confidence bound for the posterior predictive density (note that we use a lower confidence bound rather than an upper confidence bound since we are performing minimization), and β is a tradeoff parameter. In experiments we set $\beta = 0.5$.

See Figure D.3 (left). We see that the acquisition function has a small effect on the performance of BANANAS on NASBench, and UCB and ITS perform the best. We note that since the DARTS search space is much larger than the NASBench search space, in the parallel setting, the 10 neural architectures chosen by UCB in each round will have less diversity on the DARTS search space. Since ITS inherently gives a more randomized batch of 10 neural architectures, we expect more diversity and therefore ITS to perform better than UCB on DARTS. Due to the extreme computational cost of running a single BANANAS trial on the DARTS search space, we were unable to give an ablation study.

D.4. NAS algorithms for 500 queries.

In our main set of experiments, we gave each NAS algorithm a budget of 150 queries. That is, each NAS algorithm can only choose 150 architectures to train and evaluate. Now we give each NAS algorithm a budget of 500 queries. See Figure D.3 (right). We largely see the same trends, although the gap between BANANAS and regularized evolution becomes smaller after query 400.

D.5. Mean vs. random validation error.

In the NASBench dataset, each architecture was trained to 108 epochs three separate times with different random seeds. The NASBench paper conducted experiments by choosing a random validation error when querying each architecture, and then reporting the mean test error at the conclusion of the NAS algorithm. We found that the mismatch (choosing random validation error, but mean test error) added extra noise that would not be present during a real-life NAS experiment. Another way to conduct experiments on NASBench is to use mean validation error and mean test error. This is the method we used in Section 5 and the previous experiments from this section. Perhaps the most realistic experiment would be to use a random validation error and the test error corresponding to that validation error, however, the NASBench dataset does not explicitly give this functionality. In Figure D.4, we give results in the setting of the NASBench paper, using random validation error and mean test error. We ran each algorithm for 500 trials. Note that the

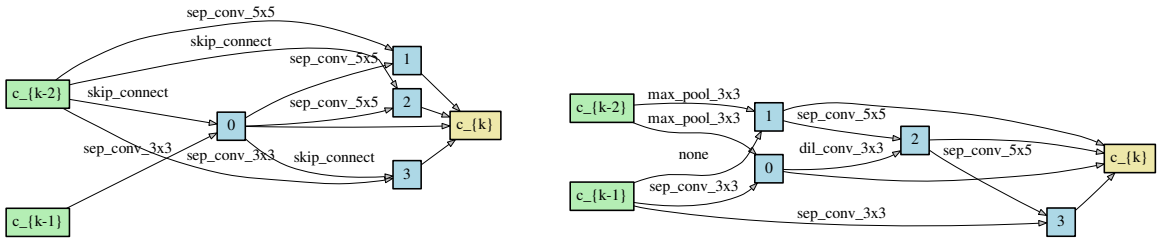

Figure D.1: The best neural architecture found by BANANAS on CIFAR-10. Normal cell (left) and reduction cell (right).

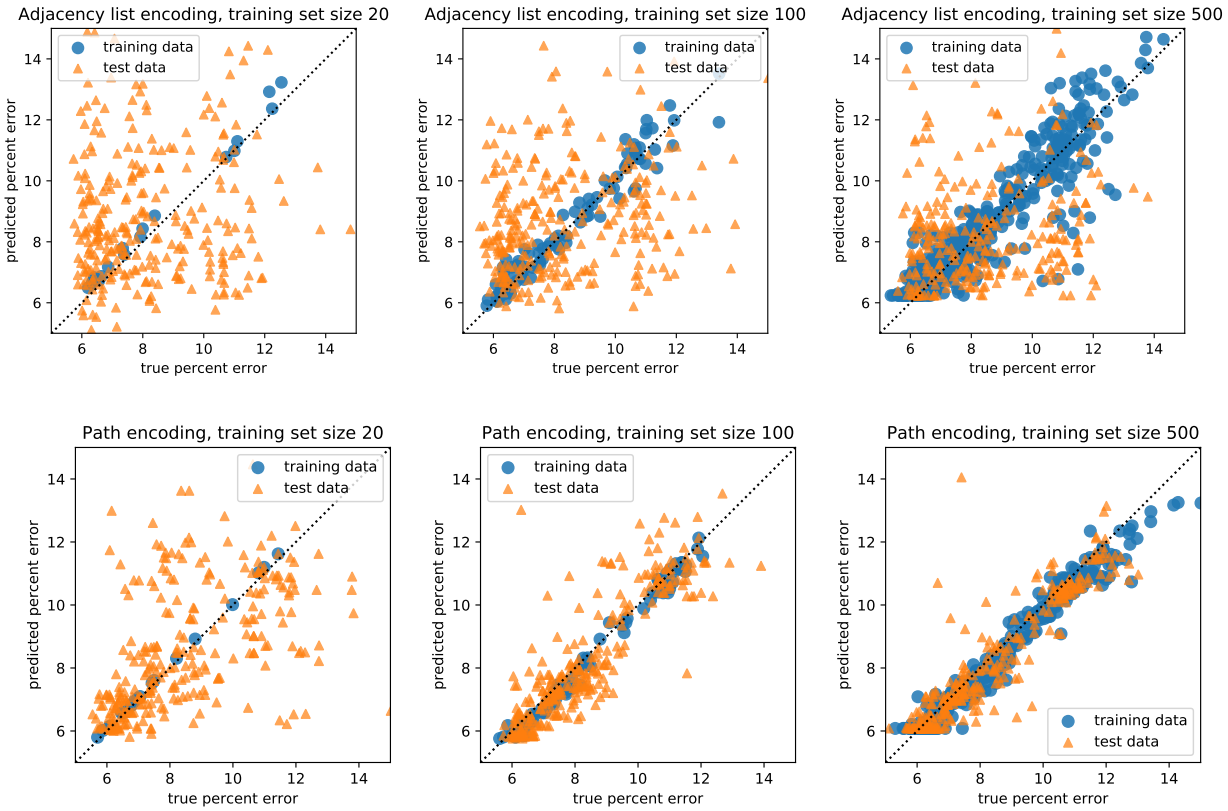

Figure D.2: Performance of the meta neural network with adjacency matrix encoding (row 1) and path encoding (row 2).

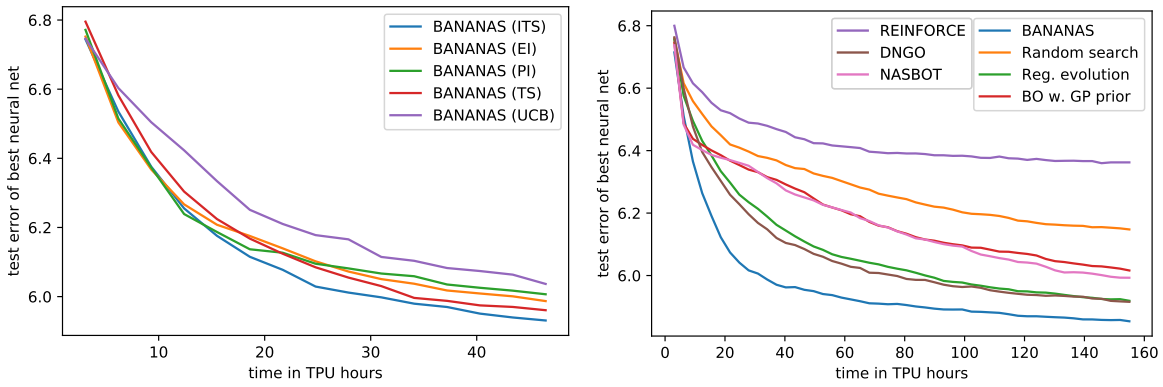

Figure D.3: Comparison of different acquisition functions (left). NAS algorithms with a budget of 500 queries (right).

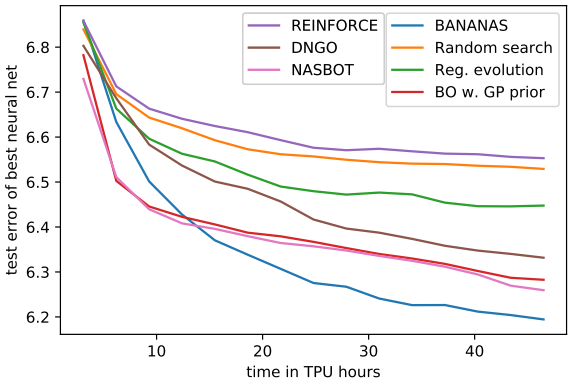

Figure D.4: NAS experiments with random validation error.

test error across the board is higher because the correlation between validation and test error is lower (i.e., validation error is noisier than normal). BANANAS still performs the best out of all algorithms that we tried. We see the trends are largely unchanged, although Bayesian optimization with a GP model performs better than regularized evolution. A possible explanation is that Bayesian optimization with a GP model is better-suited to handle noisy data than regularized evolution.

D.6. Path encoding length

In this section, we give experimental results which back up the theoretical claims from Section 4. In particular, we show that truncating the path encoding, even by an order of magnitude, has minimal effects on the performance of both the meta neural network and BANANAS.

We sort the paths from highest probability to lowest probability, and then we truncate the path encoding to length k. Note that there is a direct correlation between the probability of a path occurrring in a random spec, and the length of a path, so this is equivalent to truncating all but the smallest paths.

First, we give a table of the probabilities of paths by length from NASBench generated from `random_spec()`. These probabilities were computed experimentally by making 100000 calls to `random_spec()`. See Table 3. This table gives experimental evidence to support Theorem 4.2.

In our first experiment on the NASBench dataset, we draw 200 random training architectures, and 500 random test architectures, train a meta neural network, and plot the train and test errors. We perform this experiment with values of k of $3^0, 3^0+3^1, 3^0+3^1+3^2, \ldots, \sum_{i=0}^{5} = 364$. Note that these are natural cutoffs as they correspond to all paths of length ≤ 1, length $\leq 2, \ldots$ length ≤ 6. We ran 100 trials for each value of k. See Figure D.5 (left).

Then we compare the path length to the error of an outputted architecture from a full run of BANANAS. We average 500 trials for each value of k. See Figure D.5 (right). Our experiments show that the path encoding can be truncated by an order of magnitude, with minimal effects on the performance of the meta neural network and BANANAS.

E. Best practices checklist for NAS research

The area of NAS has seen problems with reproducibility, as well as fair empirical comparisons, even more so than other areas of machine learning. Following calls for fair and reproducible NAS research (Li & Talwalkar, 2019; Ying et al., 2019), a best practices checklist was recently created (Lindauer & Hutter, 2019). In order to promote fair and reproducible NAS research, we address all points on the checklist, and we encourage future papers to do the same.

- *Code for the training pipeline used to evaluate the final architectures.* We used two of the most popular search spaces in NAS research, the NASBench search space, and the DARTS search space. For NASBench, the accuracy of all architectures were precomputed. For

Table 3: Probabilities of path lengths in NASBench using `random_spec()`.

Path Length	Probability	Total num. paths	Expected num. paths
1	0.200	1	0.200
2	0.127	3	0.380
3	3.36×10^{-2}	9	0.303
4	3.92×10^{-3}	27	0.106
5	1.50×10^{-4}	81	1.22×10^{-2}
6	6.37×10^{-7}	243	1.55×10^{-4}

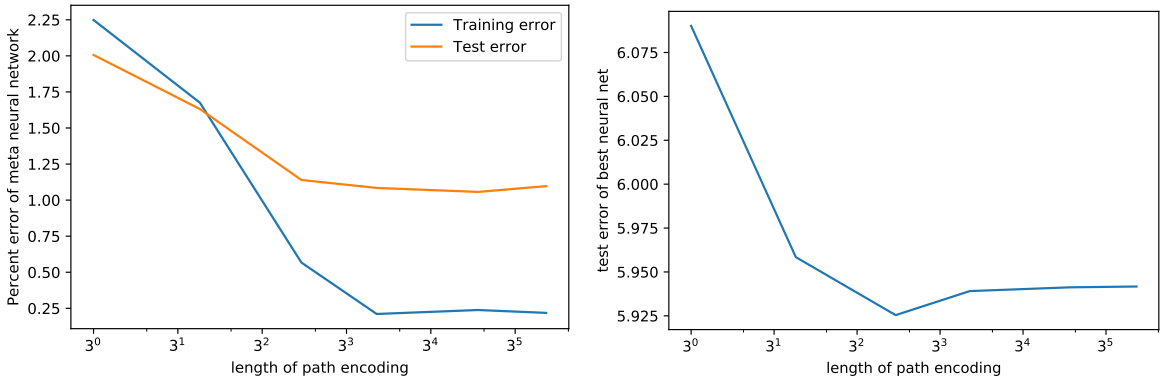

Figure D.5: Experiments on the path length versus the error of the meta neural network, for NASBench (left). Experiments on the path length versus the performance of BANANAS (right).

the DARTS search space, we released our fork of the DARTS repo, which is forked from the DARTS repo designed specifically for reproducible experiments (Li & Talwalkar, 2019), making trivial changes to account for pytorch 1.2.0.

- *Code for the search space.* We used the popular and publicly avaliable NASBench and DARTS search spaces with no changes.
- *Hyperparameters used for the final evaluation pipeline, as well as random seeds.* We left all hyperparameters unchanged. We trained the architectures found by BANANAS, ASHA, and DARTS five times each, using random seeds 0, 1, 2, 3, 4.
- *For all NAS methods you compare, did you use exactly the same NAS benchmark, including the same dataset, search space, and code for training the architectures and hyperparameters for that code?* Yes, we did this by virtue of the NASBench dataset. For the DARTS experiments, we used the reported architectures (found using the same search space and dataset as our method), and then we trained the final architectures using the same code, including hyperparameters. We compared different NAS methods using exactly the same NAS benchmark.
- *Did you control for confounding factors?* Yes, we used the same setup for all of our NASBench experiments. For the DARTS search space, we compared our algorithm to two other algorithms using the same setup (pytorch version, CUDA version, etc). Across training over 5 seeds for each algorithm, we used different GPUs, which we found to have no greater effect than using a different random seed.
- *Did you run ablation studies?* Yes, we ran a thorough ablation study.
- *Did you use the same evaluation protocol for the methods being compared?* Yes, we used the same evaluation protocol for all methods and we tried multiple evaluation protocols.
- *Did you compare performance over time?* In our main NASBench plots, we compared performance to number of queries, since all of our comparisons were to black-box optimization algorithms. Note that the number of queries is almost perfectly correlated with runtime. We computed the run time and found all algorithms were between 46.5 and 47.0 TPU hours.
- *Did you compare to random search?* Yes.
- *Did you perform multiple runs of your experiments and report seeds?* We ran 200 trials of our NASBench

experiments. Since we ran so many trials, we did not report random seeds. We ran four total trials of BANANAS on the DARTS search space. Currently we do not have a fully deterministic version of BANANAS on the DARTS search space (which would be harder to implement as the algorithm runs on 10 GPUs). However, the average final error across trials was within 0.1%.

- *Did you use tabular or surrogate benchmarks for in-depth evaluations* Yes, we used NASBench.
- *Did you report how you tuned hyperparameters, and what time and resources this required?* We performed light hyperparameter tuning for the meta neural network to choose the number of layers, layer size, learning rate, and number of epochs. In general, we found our algorithm to work well without hyperparameter tuning.
- *Did you report the time for the entire end-to-end NAS method?* We reported time for the entire end-to-end NAS method.
- *Did you report all details of your experimental setup?* We reported all details of our experimental setup.