Improving Neural Networks with Generalizable Performance Predictors and Generative Code Language Models

Improving Neural Networks with Generalizable Performance Predictors and Generative Code Language Models

a thesis presented
by
Aakash Mishra
to
The Department of Computer Science

in partial fulfillment of the requirements
for the degree of
Bachelor of Arts
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts
May 2023

Thesis advisor: Professor Stratos Idreos Aakash Mishra

Improving Neural Networks with Generalizable Performance Predictors and Generative Code Language Models

Abstract

Neural Architecture Search (NAS) is a growing field with many evolving facets of research, from evaluation strategies and search space criterion to architecture optimization strategies and performance prediction. Currently, these spaces are disjoint and constrained due to lack of generalizability. Structured search spaces restrict algorithms to specific architectures, while performance estimators are fixed to given benchmarks without the ability to conduct zero-shot evaluation. Using advances in generative AI, we present a chimera of the aforementioned methods in a tool called NAS-Assistant. Our methodology consists of a new generalizable GNN-based neural architecture encoder and a clustering, attention-based regression network that predicts model performance with high accuracy and transferability. We also propose a unique method for evaluating the contribution of each layer of a network, combined with zero-cost NAS

evaluation. Lastly, we develop a framework for using generative code language models to explore any model search space requested from NAS-Assistant. This thesis aims to demonstrate the first integrated generative AI optimizer for Neural Architecture Search.

Contents

I would like to dedicate this thesis to my family: Rajiv, Shalini and Apurva Mishra for being pillars of support in my life.

Acknowledgments

I would like to thank my thesis advisor Professor Stratos Idreos for his guidance throughout this process. I would also like to thank the Data Systems Laboratory for mentoring my research work. I would like to thank the Harvard College Research Program (HCRP) for their funding of this project. Moreover, I would like to thank my thesis readers: Professor Stratos Idreos, Dr. Pavlos Protopapas and Dr. Utku Sirin for their time and consideration. I would also like to thank my peers Frank D'Agostino '23 and Danielle Nam '24 for their feedback on this project.

0

Introduction

The field of neural architecture search (NAS) is a frontier of machine learning research aimed at making the design of neural networks more accessible. Selecting the optimal design for the given task and dataset at hand is not straightforward. Though designers typically rely on their previous experience, this may in fact bias them away from the optimal design. Therefore the goal of the ideal

NAS program would be to deliver, with high fidelity, the best model design maximizing performance given a set of resource constraints[47].

We structure the NAS problem into three components. First, there is the search space we are considering. This space is a set containing neural architectures represented as one of the following: discrete, continuous, or continuous and differentiable. Then, there is the search strategy: given a search space, how should one effectively look for the best architecture within the space? Last, we have the evaluation strategy: how we choose to allocate resources to configurations. This is important as training large neural network architectures with hundreds of parameters is computationally expensive[62].

State of the art NAS search spaces are very constrained. A global search space in NAS is broad and makes the problem difficult to structure due to the combinatorial explosion of the number of layers and nodes in a given architecture[71,64]. As a result, many papers attempt to apply structure to the problem space by adding modules that act as operator nodes and configuring those rather than all possible combinations of weights, functions and modules[37,73]. The framework of the search spaces then influences the creation of space-adapted evaluation strategies employed by Liu et al.[38], Chen et al.[10], and Hu et al.[26]. For example, many specialized predictors have been trained to fit

only a specific search space; such as BANANAS[61], Once-For-All (OFA)[7] and Semi-NAS[39]. Aside from predictors, there are architecture embedding models that have been custom-fit to the search spaces of NAS-101[69], NAS-201[15] and NAS-301[51], such as Arch2Vec[67].

Neural architecture search and hyper-parameter optimization is computationally expensive. AmoebaNet-A, NasNet, and RandomNAS both require thousands of GPU days in order to come up with the most optimal network configuration[74,46,37]. This is a result of their assumption that the search space is formatted as a discrete problem rather than a continuous problem. Algorithms based on reinforcement learning, sequential model-based optimization, evolutionary algorithms, and Bayesian optimization are inherently limited[47]. This is because these optimization approaches do not take advantage of the problem space. However, efforts to relax this search space by transforming it from discrete to continuous and differentiable using DARTS (first-order) have reduced requirements to only 2-4 GPU days. While this is several orders of magnitude better than the original options taking thousands of days, these solutions are still constricted to very specific search spaces.

Training each model configuration completely is neither computationally feasible nor scalable. Distributed NAS evaluation strategies aim

to develop resource-efficient and robust solutions for sorting out desirable architectures without conceding model validation accuracy or loss, agnostic of the search space. Single-fidelity optimizers use the results obtained after running a full training run. State-of-the-art optimization strategies aim to use the compute-power more efficiently through early-stopping which is the termination of poor performers during training. These methods are known as multi-fidelity gradient-free optimizers. These multi-fidelity optimizers utilize "low fidelity" evaluations throughout the training process in order to optimize the amount of resources allocated to architecture[17,32]. Additionally, we can use performance predictors to lower the amortized cost of evaluating each model through thresholding and model ranking[42]. Last, with the development of high fidelity proxyless metrics, we can quickly evaluate the performance bounds of a model without training it for more than a single epoch.

Pre-tabulated benchmarks are not easily generalizable. Pre-tabulated benchmarks, such as NAS 101[69], NAS Benchmark-201[15], NAS Benchmark-301[52] and NAS Benchmark-360[57], aid in reducing resource requirements for NAS researchers. They do this by pre-tuning, training, and storing model configurations and evaluations allowing researchers to use pre-tabulated results to evaluate the efficacy of the NAS approach. However, this fails to address the in-

herent problem of pre-computed benchmarks: the dependence on fixed problem space parameters like model types and the dataset. NAS developers who aim to develop schedulers that perform well on these pre-tabulated benchmarks risk over-fitting their strategy to the benchmark types and losing generalizability to new problem spaces.

Adding new benchmarks is not a scalable solution. One possible solution is to extend existing pre-tabulated benchmarks with more data or add entirely new benchmarks. This has partially motivated pre-computed benchmarks like NAS-360[56] because previous benchmarks are dominated by vision datasets[5,48]. However, in the long term, this is not a viable solution due to significant increases in time and cost: achieving comprehensive pre-tabulated results for every configuration of the DARTS search space took hundreds of GPU days[52]. With the rapid pace in development of new neural architectures[44,63], it is fundamentally difficult for researchers to rapidly test their scheduling algorithms on new benchmarks because of the holistic cost of training and development time as well access to resources[55].

Solution: NAS needs generalizability. From search space to evaluation strategy, we need a NAS program that can work outside of the search spaces and types of models it was originally designed to train or evaluate. To this

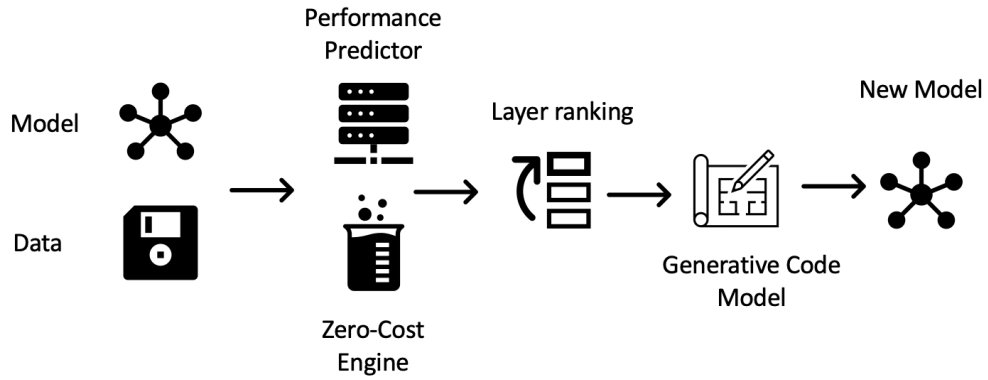


Figure 1: Illustration of the NAS-Assistant workflow from data to output model.

end, GENNAPE[42] processes the computation graphs generated by compiling a model and encodes them into an embedding using a graph neural network encoder. They then feed the embeddings into a soft-clustering algorithm which then assigns the embedding as input to a specialized multi-layer perceptron (MLP) that calculates the accuracy value specifically for convolutional neural networks. We build on this by creating our own generalized performance model as a component of our generalized NAS platform. Specifically, our goal is to take an input model architecture A_I and output a new architecture A_O that is locally optimal for any kind of model. We achieve this by first extending the GENNAPE framework towards understanding both Tensorflow and PyTorch defined models such that it can encode the architecture based on the computational graph (CG) for both. We then train the encoder-regression model to

not rely on clustering anymore, as it is less generalizable to cluster, and then direct a model to be processed by a specialized MLP for a specific cluster. If a point is far away from any cluster, it is unlikely that the MLP regressor would be good at predicting model performance. Instead, we use cross-attention with a larger MLP to overcome this problem. Finally, we add proxyless NAS metrics in order to extend the predictor's capabilities beyond the single dataset whose performance it was trained to predict. We devise a new metric using occlusion to determine the contribution of each weighted model component to the overall performance. Based on this metric and our model evaluation, we then generate new models using a large generative code language model. This process is outlined in figure 1. We call this system the NAS-Assistant. In summary, the contributions of this thesis are as follows:

(1) A new generalizable GNN-based neural architecture encoder and a clustering attention-based regression network that predicts model performance with high accuracy and transferability.

(2) A unique method for evaluating the contribution of each layer of a network, combined with proxyless NAS evalution.

(3) A framework for using generative code language models to explore any model search space requested from NAS-Assistant.

1

Related Works

1.1 Graph Neural Networks and Representation Learning

Graph representation learning entails the modeling of representations of graph nodes and edges by using some low dimensional vector[12,23]. Although the means to accomplish this at first were sparse, the emergence of convolutional neural networks (CNNs) brought a resurgence in geometric machine learning. As a

natural result, the application of CNNs to 2d images was then extended to non-Euclidean inputs[72].

Early Graph Embedding Models. Based on the ideas behind `word2vec` and natural language embedding models that successfully launched representation learning as a viable methodology, graph neural networks shortly followed[21]. The first approach was DeepWalk, which used random walks and depth-first search in conjunction with embedding SkipGrams[45] to create a graph representation[11].

Issues with early methods. Models like `node2vec`, TADW, and LINE suffer from the lack of parameters shared between nodes in the encoding schema and generalization to new graph topologies[68,70]. These approaches were also more inefficient because they were limited to inputs of a specific length. This means that as inputs grew linearly, so too did the number of parameters in the network[23].

Spectral Graph Learning Methods. Rooted in graph signal processing theory[50], spectral methods in GNNs have led to the emergence of the Graph Convolutional Network Operator[29]. Given a graph signal x, let us define Fourier transforms $\mathbb{F}$ as the following: $\mathbb{F}(x) = U^T x$. Note that $\mathbf{U}$ is the matrix of eigenvectors derived from the normalized Laplacian of the graph such that

$\mathbf{L}_{norm} = \mathbf{I_N} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$, given that $\mathbf{A}$ represents our adjacency matrix and $\mathbf{D}$ represents our degree matrix. The Laplacian has unique properties that guarantee its being a real symmetric positive semidefinite matrix[41].

We then derive the convolution operation to be the following:

$$\mathbf{g} \star \mathbf{x} = \mathbf{U}(\mathbf{U}^T g \odot \mathbf{U}^T \mathbf{x})$$

Note that in this equation, the expression $\mathbf{U}^T\mathbf{g}$ is our filter operation. This means that $g = g_w$ can be our learnable kernel weight.

Further improvements to the GCN additionally developed this idea using Chebyshev polynomials[24], which was then further simplified by Kipf & Welling[29] to the following equation:

$$\mathbf{g} \star x = w_0\mathbf{x} - w_1\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{x}$$

If we simplify the following, assuming that $w_0 = -w_1$, and we renormalize to prevent exploding gradients and add a skip connection using the following expression $\mathbf{I}_N + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$, we arrive at the final form of the GCN: where $\mathbf{X}$ is the input for the graph signal in matrix form, $\mathbf{W}$ is the learnable parameter and the result is the convolution operation.

$$\mathbf{g} \star \mathbf{X} = \mathbf{H} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{X} \mathbf{W}$$

The benefits of this operation are that the learnable parameters are input agnostic in terms of scale. The GCN is able to read inputs of different types and the Fourier transformation brings the representation benefits of the Fourier series for sparse graph representation[13].

Unsupervised Graph Encoder Methodologies. The first encoder methodology utilized GCNs in order to create a Graph Auto-Encoder (GAE) model. This encoder-decoder structure has the goal of trying to decipher the adjacency matrix from the encoding and uses the similarity between the original and reconstructed in order to provide a criterion for the model to train[43]. Using contrastive learning, this methodology can be expanded to further unsupervised learning. There are several examples such as Deep Graph Infomax (DGI)[58], Infograph[53] and Multi-view[66] that all utilize various contrasted forms in order to achieve high-fidelity representations. The basic premise lies in trying to create a metric of distance between two representations and the choice of the representations themselves. For example, in this thesis (similarly to GENNAPE[42]), we use a specific form of graph edit distance to develop a metric of distance and then contrast the learned embeddings using cosine similarity to get the performance

criterion.

Additionally, in order to make the GCN model layers more discriminative, we incorporate the Weisfelier-Leman test which test for graph topologies that are isomorphic[29]. This is important to learning the encoding structure of a neural network because there are many ways to code and compile and architecture that may perform the same in terms of performance. The evolution of this test integration led to the proposition of Graph Isomorphic Networks (GINs) which can be used in conjunction with the convolutional operator $\mathbf{g} \star \mathbf{x}$ mentioned earlier[65].

1.2 Architecture Performance Predictors

In order to maintain consistency in the descriptors for the following methods, let us define the **setup time** as the computational time taken for any generalized training or initialization for the method. The **query time** is defined as the time it takes the method to compute an architecture specific prediction[62].

Supervised Learning Methods. The first type of performance predictor is based on supervised training, where the setup time is defined as the time it takes to train on a large set of architectures using some sort of representation to predict a ranking metric or some sort of performance indicator, such as a loss

value or end accuracy value[40]. The actual query time for such an indicator is very short, which enables high inference loads. For example, GENNAPE is a supervised type model where the neural network is trained on 40,000 architectures from NAS-101. These predictors can further be used by other statistical frameworks such as Bayesian optimization or RL techniques to maximize their efficacy. Other examples include boosting or random forest, Gaussian processes or specialized encoding based regressors[40,49,60].

Zero-Cost Methodologies. Zero-cost methods such as Synflow[1], RE-Measure and others[36] are examples of predictors that do not require any setup and small query time intervals in order to compute certain statistics from running a single minibatch of data. These statistics can be calculated by doing a forward or backward pass of the model[1]. Additionally, other metrics include measuring metrics related to in-network attributes such as activation weights. These methods can be attributed to generally ranking models, but may lack high value performance prediction capabilities. For this reason, it may be best to use metrics like these to generally filter bad configurations early on in a NAS search.

Learning Curve Modelling. Some predictors adopt the task of forecasting the learning curves of a model given some number of hyperparameters

and the last given loss value. In order to model the loss curve methods adopt Bayesian, parametric or neural network models to structure the learning problem. Domhan et al.[14] describes a set of 11 model types, such as pow_4: $f(t) = c - (at + b)^{-\alpha}$. In this equation, a, b, c, α are some set of parameters that can be fitted to a curve. The goal was to use a set of $K = 11$ parametric functions $\phi_i(\theta_i, t)$ to extrapolate loss curves to future time steps based on the previous n time steps. Domhan et al.[14] opted to perform Markov Chain Monte Carlo (MCMC) inference to predict future values. Later, Klein et al.[30] built on this work and coupled Bayesian neural networks with these parametric models to forecast loss curves. Depending on the loss landscape, the performance of these methods can have high variability[20].

Hybrid Approaches. Often, the model-based supervised predictor can be combined with evaluation based methods that use early stopping. Examples of this include median rule[22], asynchronous successive halving[34], Population Based Training[27], and Hyperband[33]. Additionally, we can incorporate loss curve modelling and time series forecasting in to these same algorithms. Zero-cost proxies can then serve as generalized filters to kick start the NAS search. There is currently many open questions and possible ensemble models that can be constructed[62].

1.3 Large Language Code Models

Large language models are a suite of models aimed to learn relationships in natural language data. They are typically trained on vast text datasets, ranging from books, web pages, and code repositories. Many state of the art large language models are high parameter transformer models that aim to complete two main tasks in natural language processing: infilling and synthesis. Infilling is the task of filling in sentences or code given the surrounding context. Synthesis is the task of generating new text/code or responding to a given prompt in a contextually-aware way. Significant breakthroughs in the LLM space include GPT-3 by OpenAI and Incoder by Meta[19].

The Incoder transformer model uses 6.7 billion parameters and is trained on a large corpus of Python code. They perform many experiments to test the model's performance for code synthesis and code line infilling. They use a zero-shot evaluation on HumanEval[9] and MBPP[3], which are benchmarks that contain docstrings and aim to produce a correct matching Python function.

We focus mainly on decoder-only models since we only care about the resulting code. However, other works include encoder-only masked language models which allow embeddings to be generated for text and code[28,18] and encoder-decoder models that allow both capabilities[35,2,59].

1.4 Architecture Families

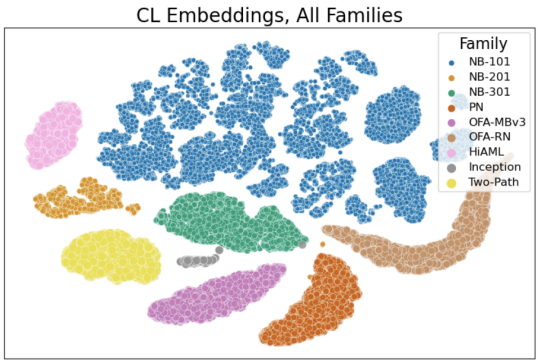


Figure 1.1: TSNE Breakdown of GENNAPE Generated Latent Embedding Space from Mills et al.[42].

NAS-Bench-101 (NB101)[51], NAS-Bench-201 (NB201)[15] and NAS-Bench-301 (NB301)[52] are cell based search space records where NB201 and NB301 are DARTS configured spaces that are built on top of AutoDL genotype configurations. Note that NB301 is a surrogate benchmark and hence the models derived from it are estimated for accuracy terms. ProxylessNAS (OFA-PN)[8] and Once for-All-MobileNetV3 (OFA-MBv3)[31] are based on variations of the original MobileNet configuration and cell based topology. The Once-for-All ResNet family is a residual block based ResNet group of families from He et al.[25]. HiAML architectures are based on GA-NAS and were original designed to solve facial recognition problems[60]. Inception and Two-Path are variations of the Inception family from Szegedy et al.[54] where Inception is the single path backbone type

and Two-Path has two running backbones for weight efficiency. The different families and their similarity in latent space are shown in figure 1.1.

2

Methods

2.1 Creating a Generalizable Performance Predictor

2.1.1 Creating a Usable Computational Graph Representation

We train a graph neural network to first learn the structure of the model architecture. We choose to represent this architecture by using the computation

graph of the compiled model. In order to get the right dimensions for the model, we also require that the user of NAS-Assistant also include the dataloader to the API. From this input, we construct a Tensorflow style computational graph. While PyTorch has no individual method of accessing the entirety of a model's computational graph for visualization or traversal, we use the `TorchView` package to construct similar graphs for the model to load in.

In Tensorflow, the computational graph is stored in a `tf.Graph` object which contains both `Tensor` and `Operation` objects. We use the `Tensor` objects to construct the weighted edges of our graph and the `Operation` objects to represent our nodes. Given a static graph object, we can then traverse through the graph and create our own representation which we store in the `ComputationalGraph` class. In this case we construct two types of nodes. One type of node is a `WeightedNode` which has a weight tensor associated with it, like a projection tensor or convolutional kernel. All other operations are stored as `RegularNode` objects.

In PyTorch, the computational graph has to be constructed using the dynamic graph interface using `Autograd`. We then apply `TorchView` to this gradient object. The module uses the parsing interface to create a `DOT`-formatted `DiGraph` object which is similar to Tensorflow's `tf.Graph()` object API. From this phase, we implement the same parsing to load the graph into our `ComputationalGraph`

object format and split the representation into weighted nodes and regular nodes. The resulting model is not exactly the same as the Tensorflow graphs because of representation differences and differing node definitions. However, we find that we can generate extremely similar embeddings between models that are written using Tensorflow and Pytorch by using Open Neural Network Exchange (ONNX)[4] to ensure similar graph encodings for the same checkpoint objects.

2.1.2 Computational Graph Embedding Modeling

In order to represent each node in our `ComputationalGraph` object, we create a node embedding model that creates an embedding layer for encoding each type of attribute, such as weight size, operation type, kernel shape and bias attribute.

Alongside our node encoding model, we pass the node encodings and adjacency matrix into our GINConvolution network with 6 layers, followed by another transformer with cross-attention and four heads to create 256 embeddings from each model. These embeddings are concatenated together to create one large 512 long vector embedding for the computational graph. We then aggregate this embedding across the dimensions of all of the nodes using another GIN

Convolutional model.

Our training criterion for this model is twofold and partly based on contrastive learning. First, we use the criterion described in GENAPPE. This involves using dropout augmentation to create similar models in our batch. For this, we repeat the input given to the model with dropout augmentation at the node level. This means that each batch size is doubled. By passing the input into our network $F(x)$ twice, we achieve two embeddings: $F(x) = \mathbf{e_1}$ and $F(x) = \mathbf{e_2}$. This is because the dropout is randomized for each input. For each embedding that we generate, we train a projection head to move it into lower dimensional space for representation learning. Using cosine similarity between two vectors $\mathbf{e_1}$ and $\mathbf{e_2}$

$$\frac{\mathbf{e_1} \cdot \mathbf{e_2}}{\tau}$$

where τ is our temperature, we generate the following loss function:

$$L_{\text{constrastive}} = -\sum_{i \in I} \sum_{l \neq i} \alpha_l^{(i)} \log \frac{\exp\left(\frac{\mathbf{x_i} \cdot \mathbf{x_l}}{\tau}\right)}{\sum_{r \neq i} \left(\frac{\mathbf{x_r} \cdot \mathbf{x_l}}{\tau}\right)}$$

In this function $\mathbf{x_1} = \text{proj}(\mathbf{e_1})$ and $\alpha_l^{(i)}$ is a component of our proxy soft label for each graph such that $\sum_{l \neq i} \alpha_l^{(i)} = 0$ and each $\alpha_l^{(i)} \geq 0$. In order to come up

with this soft label, we define a form of graph distance. For this, we calculate the normalized Laplacian of a graph G defined so:

$$\mathbf{L}_{norm} = \mathbf{I_N} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$$

Note here that $\mathbf{A}$ and $\mathbf{D}$ are the adjacency and degree matrix of graph G.

Given this matrix, we then compute the eigenvector decomposition $A = Q\Lambda Q^{-1}$ where ordered from lowest to highest, we take the first 21 eigenvalues and used $\lambda_{1\cdots21}$ as per Dwivedi & Bresson[16] to create a Euclidean distance metric such that $d = \sqrt{\sum_{i=1}^{n}(\lambda_{vi} - \lambda_{ui})^2}$ where $n = 21$. We then take the softmax of these distances values such that our resultant equation is the following.

$$\alpha_l^{(i)} = \text{softmax}(d_i) = \frac{e^{d_i}}{\sum_{j=1}^{n} e^{d_j}}$$

Once we have calculated our constrastive loss value, we then proceed to add our reconstruction loss based on cosine similarity. We use a large MLP model to take the batch encoding generated by the model and then attempt to recreate the node embeddings of the model before it was passed into the main encoder. From this we generate two versions of node-embeddings: one that is the original encoder output, and the other that is the reconstructed version. We use

consine similarity once more between the two outputs $\left(\frac{\mathbf{e_i} \cdot \mathbf{e_j}}{\tau}\right)$ for each value and average the negative similarity in the loss function. We then define two hyper-parameters β_0 and β_1. We use these parameters to help weigh our contruction and constrastive losses. The final form of the loss function is as follows:

$$L = \beta_0 L_{\text{constrastive}} + \beta_1 L_{\text{reconstruction}}$$

We then conduct hyper-parameter tuning over our losses function weighting. For the cosine temperature, we use a linear decay schedule that starts at 0.35 and ends at 0.05. A decreasing cosine temperature is understood to be a reflection of the model's confidence in the prediction. As a result, our linear schedule is enforcing the model predictions become more confident the longer it trains.

2.1.3 Unsupervised Clustering

Given the diversity of architectures and the breadth of the latent space, it is clear that using a universal regressor to predict performance off a single embedding is a difficult task. In order to capture a greater amount of variance, we aim to use clustering to direct the regressor towards specific regions in the latent space to make precise predictions. Given that encoding is an unsupervised task, we adopt Gaussian mixture models as our unsupervised clustering method.

Mixture models are generative probability models that combine multiple probability distributions. Based on the generated combined probability distribution, our goal is to understand the parameters for each probability distribution, as well as the mixture coefficient, or probability of sampling a point from that distribution. We call each distribution a component of the mixture model. To formalize, in a mixture model we have Q components, a mixture coefficient that determines the probability of sampling each component π_q, the parameters for each component θ_q, and the data generated from the distribution x generated with probability $P(x|\theta_q)$.

The goal for the mixture model is to then infer which component q generated a given point x. As such, this can be seen as a unsupervised classification problem. Given a set of datapoints $\mathcal{X}$, we want to learn a mapping $f : \mathcal{X} \rightarrow Q$, where Q is the set of clusters, or in the mixture modeling case, the set of components.

In this vein, we employ a Gaussian Mixture Model. We assume that the data for each cluster is generated from a spherical Gaussian, all the data $\mathcal{X}$ is generated from a mixture of $|Q|$ Gaussian components, each Gaussian distribution has the same variance σ^2.

We assume this generative probability model for the underlying distribution

of our observed latent embedding space with a dimensionality of 512. This assumption is in line with previous works, such as in GENNAPE[42] where they use a fuzzy k-means clustering algorithm. Our assumptions make the GMM analagous to this fuzzy k-means clustering algorithm, with the added benefit that we are able to denote a probability distribution for the likelihood that a datapoint belongs to each given cluster.

To fit this mixture model to our observed data, we use the Expectation Maximization algorithm. As mentioned, each component is a Gaussian, or normal distribution parameterized as $R_q \sim \mathcal{N}(\mu_q, \sigma^2)$. We begin by randomly guessing the means for each component μ_q. We then calculate posterior probabilities for each data point in the expectation step:

$$\text{Expectation Step: } P(q|x_i, \boldsymbol{\mu}, \sigma^2, \boldsymbol{\pi}) = \frac{\pi_q P(x_i|\mu_q, \sigma^2)}{\sum_{q'=1}^{Q} \pi_{q'} P(x_i|\mu_{q'}, \sigma^2)}$$

Where $\boldsymbol{\mu}$ and $\boldsymbol{\pi}$ are vectors of all $|Q|$ means and mixture values. We know that a given data point x_i is most likely with the largest $P(q|x_i, \boldsymbol{\mu}, \sigma^2, \boldsymbol{\pi})$. We can compute $P(x_i|\mu_q, \sigma^2)$ by using the PDF of the normal distribution with mean μ_q and variance σ^2. Then, in the maximization step, we calculate the new

centroids μ_q and mixture coefficients given the posterior probabilities:

$$\text{Maximization Step: } \hat{\mu}_q = \frac{\sum_i P(q|x_i)x_i}{\sum_i P(q|x_i)}, \quad \hat{\pi}_q = \frac{\sum_i P(q|x_i, \boldsymbol{\mu}, \sigma^2, \boldsymbol{\pi})}{N}$$

Where N is the total number of data points. As such, we converge to a given assignment of components for each data point along with their mean centroids.

Algorithm 1 Expectation Maximization for Gaussian Mixture Model

Require: Data $X = \{x_1, \ldots, x_N\}$, number of components $|Q|$, tolerance threshold ϵ

Ensure: Means $\boldsymbol{\mu}$, variance σ^2, mixture coefficients $\boldsymbol{\pi}$

1: Initialize means $\boldsymbol{\mu} = \{\mu_1, \ldots, \mu_{|Q|}\}$ randomly
2: Initialize variances σ^2
3: Initialize mixture coefficients $\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_{|Q|}\}$ uniformly
4: **repeat**
5: Set $P(q|x_i, \boldsymbol{\mu}, \sigma^2, \boldsymbol{\pi}) = \frac{\pi_q P(x_i|\mu_q, \sigma^2)}{\sum_{q'=1}^{Q} \pi_{q'} P(x_i|\mu_{q'}, \sigma^2)}$
6: Update means using $\hat{\mu}_q = \frac{\sum_i P(q|x_i)x_i}{\sum_i P(q|x_i)}$
7: Update mixture coefficients using $\hat{\pi}_q = \frac{\sum_i P(q|x_i, \boldsymbol{\mu}, \sigma^2, \boldsymbol{\pi})}{N}$
8: **until** $\|\boldsymbol{\mu} - \boldsymbol{\mu}_{\text{old}}\|_2 < \epsilon$

2.1.4 Mixture of Experts Regressor

As previously mentioned, the GMM clusters the points for us in order to provide a soft label for our regressor. The regressor we train is a Mixture of Experts (MOE) model that uses an embedding space to combine the gating weights that determine which experts to go to with the input in order to pass the input

through the weighted mixture of the expert models. Normally, a gated network outputs a softmax representing probabilities for using each expert model, but here we have replaced this with our soft label from the GMM. By having the experts share weights, we reduce overfitting and increase generalizability. This is important for zero-shot evaluation of models where aspects of the latent space have not been seen before, yet we still want to have some combination of experts that contain weights to predict the model's performance given the latent space emebdding.

2.2 Prompting Large Language Models

Using our generalizable neural network performance predictor, we utilize it in order to infer the value of various aspects of the model. We aim for this value to correspond to the normalized contribution of each model layer to the overall performance prediction. In order to accomplish this, we implement a breath-first node masking algorithm. In this, we traverse each layer of the graph and take the operation nodes associated with that layer and mask them along with their edges from the graph. We then replace these edges with dummy edges in order to preserve the topology of the graph. For each masking configuration, we pass the computational graph through the neural network encoder and generate a

new predicted accuracy value. We then subtract the list of accuracy values from the baseline accuracy of the original model and normalize the differences to get a normalized contribution weight for every layer of the neural network.

Algorithm 2 Generalizable NAS-Assist Performance Predictor

procedure NAS-Assist(G, E) ▷ G: graph, E: encoder
  $A_{\text{baseline}} \leftarrow \text{PredictAccuracy}(G, E)$
  Initialize list A_{masked}
  for each layer l in G **do**
    $G_{\text{masked}} \leftarrow \text{MaskLayer}(G, l)$
    $A_{\text{masked}}[l] \leftarrow \text{PredictAccuracy}(G_{\text{masked}}, E)$
  end for
  $\Delta A \leftarrow A_{\text{baseline}} - A_{\text{masked}}$
  $W \leftarrow \text{Normalize}(\Delta A)$ ▷ Normalized contribution weights
  return W
end procedure

Once we have the relative weights for each part of the model, we pass the model code and the corresponding rankings to the large language model with the following prompt. Note that the prompt changes for the differing types of code language models. For this project, we utilized `InCoder`[19] and `GPT Da-Vinci-002 (3.5-Turbo)`[6].

We then provide the following docu-string prompt in order to have the large code language model generate our suggested code. Prompt: `The tensorflow model is annotated with comments that contain a value corresponding to how helpful a layer is. Positive values means the layer is improving`

LAYER NAME	RELEVANCE
Input_9	0.3253
conv2d_364	-0.0404
Batch_normalization_370	-0.0404
re_lu_369	0.3253
conv2d_365	0.3253
batch_normalization_371	0.3253
re_lu_370	0.3253
conv2d_366	-0.0404
batch_normalization_372	-0.0426
re_lu_371	-0.0426
conv2d_367	0.3074
batch_normalization_373	0.0
re_lu_372	-0.0404
conv2d_368	-0.0426
batch_normalization_374	-0.0404
re_lu_373	-0.0426
conv2d_369	-0.0404
batch_normalization_375	-0.0404
re_lu_374	-0.0426
conv2d_370	-0.396
batch_normalization_376	-0.0426
re_lu_375	-0.0426
concatenate_1	0.3253
tf.identity_9	-0.2995

Figure 2.1: Labeled NB101 inception-like model with layer relevance score output for prompting a code language model.

`performance and negative means that the layer is not helping performance. Come up with a new model based off the ranking and the code that follows while maintaining the same types of operations.`

For each piece of code that the large langauge model outputs, we then run algorithm 2 in order to test our the estimated prediction. Based on a user given threshold, we repeat this process until the given threshold is achieved by finding a model computational graph emebedding that has a qualifying accuracy. If NAS Assist cannot converge to the thresholded accuracy in a `MAX_ITERATION`

Algorithm 3 NAS-Assist Prompting

procedure NASAssist(E, `MAX_ITERATION`, `THRESHOLD`) ▷ E: encoder
    Initialize G_{best} and A_{best}
    for $i \leftarrow 1$ to `MAX_ITERATION` **do**
        $G_{\text{current}} \leftarrow$ LargeLanguageModelOutput()
        $W_{\text{current}} \leftarrow$ PredictionModel(G_{current}, E)
        $A_{\text{current}} \leftarrow$ EstimateAccuracy(W_{current})
        if $A_{\text{current}} \geq$ `THRESHOLD` **then**
            return $G_{\text{current}}, A_{\text{current}}$
        else if $A_{\text{current}} > A_{\text{best}}$ **then**
            $G_{\text{best}} \leftarrow G_{\text{current}}$
            $A_{\text{best}} \leftarrow A_{\text{current}}$
        end if
    end for
    return $G_{\text{best}}, A_{\text{best}}$
end procedure

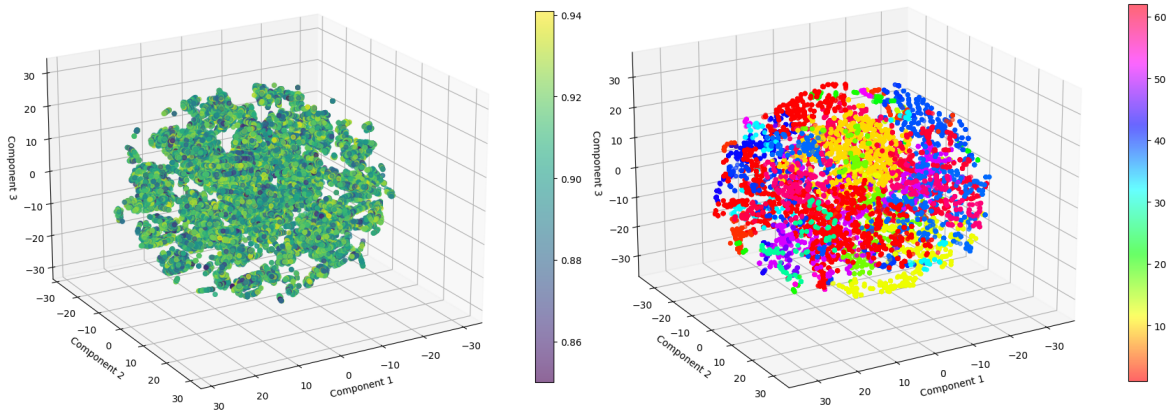
number of steps, then it returns the best mode it found until that point. This methodology is described in algorithm 3.

3

Evaluation and Results

3.1 Encoding Results

After training the node and graph neural network encoder, we then proceed to take the 512 size embeddings and run principle component analysis (PCA) and t-distributed stochastic neighbor embedding (TSNE) in order to visualize the latent encoding space of the computational graphs. Ideally, a successful



(a) TSNE Breakdown of NB101 Space with Accuracy Shading

(b) TSNE Breakdown of NB101 Space with Unsupervised Cluster Assignment Shading

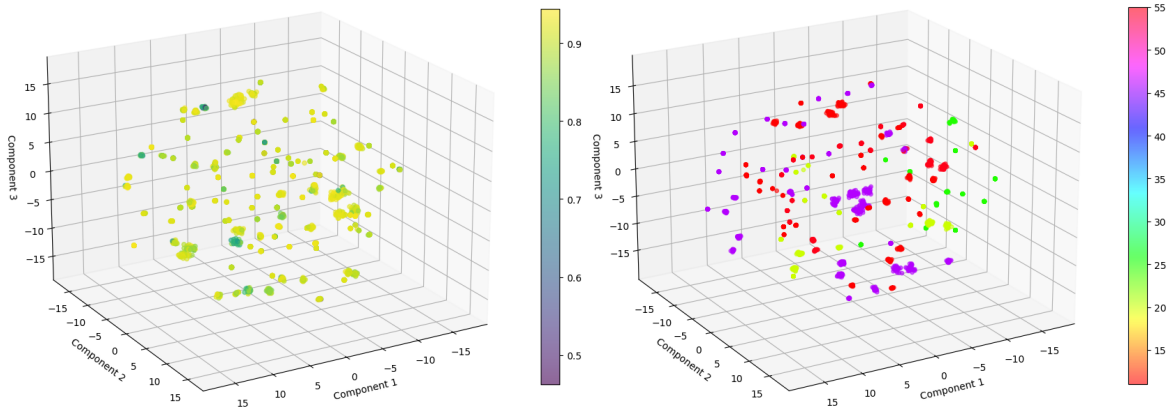
Figure 3.1: NB101 TSNE components reveal that our graph encoder has successfully encoded the computational graphs such that patterns in performance and clusters around different model types are visible.

embedding space will reveal patterns that correspond to non-embedded indicators such as accuracy or flops. We analyze the embedding spaces for each of the families below, as well as the success of the Gaussian mixture model clustering algorithm.

NB101 Encoding and Clustering Results. As shown in figure 3.1 (a), there are distinct clusters of points in the search space that correspond to NB101 architectures. Some of these small clusters contain lightly shaded points that represent higher performance architectures. Likewise, we see that there are small clusters of darkly shaded regions which are encoded computational graphs

from relatively poorly performing architectures. Note that in order to visualize effectively, only the 40,000 best-performing architectures are shown encoded here. Then there are groups of clusters that do not seem to have a pattern in which types of architectures they represent. Some of the central-most clusters are a diverse mix of differently performing architectures, and this represents the difficulty in the downstream task of predicting their performance. In figure 3.1 (b), we see that some of the clusters are well-characterized by the unsupervised learning task while others are not. For example, we can clearly see that cluster 1 (bright red) is present in different portions of the latent space and does not seem to vary in assignment with architecture performance. On the other hand, cluster 21 (bright green) consists of mostly well-performing architectures (> 0.92) and cluster 12 (gold) consists of relatively poor performing architectures (< 0.88). Since the encoder and the Gaussian mixture model were trained on NB101 computational graphs, we can now see how well our methodology extends to other CNN-vision neural network families from alternative benchmarks.

NB201 Encoding and Clustering Results. In figure 3.2 (a), we see that the encodings of the computational graph are much more sparse for the NB201 benchmark models as compared to those seen for NB101 in 3.1. Each of the clusters in this space has a distinct performance profile. The small clusters

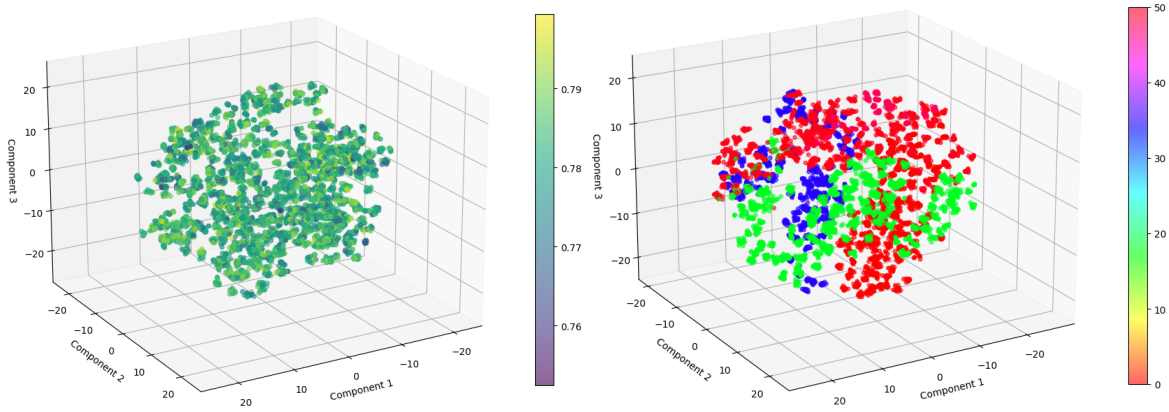


(a) TSNE Breakdown of NB201 Space with Accuracy Shading

(b) TSNE Breakdown of NB201 Space with Unsupervised Cluster Assignment Shading

Figure 3.2: NB201 TSNE components reveal that our graph encoder has successfully encoded the computational graphs outside of its training domain.

mostly consist of architectures that perform similarly. This shows that NAS201 will be easier for the regressor to generalize to because, as long as the clustering model can assign it correctly to the right cluster, the performances within a specific group are close enough to each other that the model will generally be correct. Additionally, figure 3.2 (b) demonstrates that the unsupervised clustering model groups together many distinct clusters. This will likely hurt generalizability in downstream tasks because the grouped mini-clusters do not share the same performance profiles. For example, cluster 45 (purple) consists of several mini-clusters that had better performance as seen in (a) but also two mini-clusters with worse performance. The effectiveness of clustering is highly



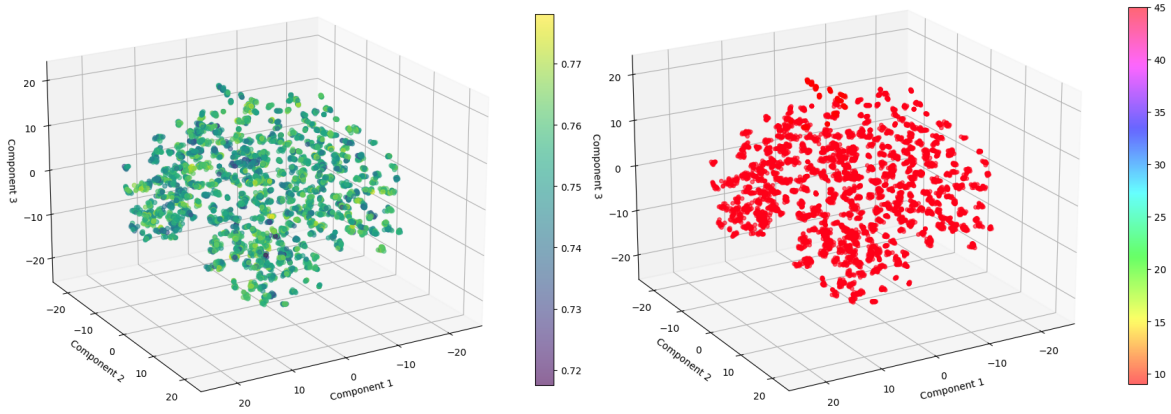
(a) TSNE Breakdown of OFA ResNet Space with Accuracy Shading

(b) TSNE Breakdown of OFA ResNet Space with Unsupervised Cluster Assignment Shading

Figure 3.3: OFA ResNet TSNE components reveal that our graph encoder has successfully encoded the computational graphs outside of its training domain and assigned clusters appropriately.

dependent on the proximity of encodings to the original training data in the nb101 encoded latent space. We see that although NB201 is close enough to the original for cluster differentiation to occur (having a spread of different cluster values), the unsupervised cluster model struggles at labelling each of these distinct mini-clusters due to its distance from the original unsupervised training space.

OFA ResNet Encoding and Clustering Results. As seen in figure 3.3 (a), the encoding schema has managed to create a structured space for the architecture encodings where position has some correlation with performance. For example, we see in the bottom-left-most corner that there is a larger concentra-



(a) TSNE Breakdown of OFA PN Space with Accuracy Shading

(b) TSNE Breakdown of OFA PN Space with Unsupervised Cluster Assignment Shading

Figure 3.4: OFA PN TSNE components reveal that our graph encoder has successfully encoded the computational graphs outside of its training domain but the cluster assignment here is lacking.

tion of better performing architectures (relative to the rest of the family); in the right-hand side we see some mini-clusters of worse- (darker) performing architectures. Once again, distance from the training set plays a role in the cluster assignment. Since the ResNet architecture is a derivative of the NB101 architecture, the cluster assignment in figure 3.3 (b) appears to be more structured and clearly identifies the patterns in the space. Cluster 13 (light green) encapsulates most of the better (> 0.78) performing architectures in this space.

OFA PN Encoding and Clustering Results. Due to the relative distance of the OFA PN class of models from the NB101 embeddings in the shared latent space, we see that the clustering model has done a poor job of differentiating

the cluster assignments for values here. Despite the prescence of various possible clusterings seen in figure 3.4 (a), we see that in (b) everything is still given the same cluster assignment. For future work, it may be more appropriate to use a more generalizable model that can better cluster distant families. The remaining clustering and encoding graphs are included in the appendix.

3.2 Zero-Shot and Fine Tuning Results for Prediction Regressor

Spearman Rank Correlation Coefficient (SRCC). SRCC is commonly used as a measurement of performance for models in Neural Architecture search because it measures the degree of monotonicity between two variables. The coefficient is calculated by getting the ranks of a list of variable and then calculating the difference between the ranks. We then take the Pearson correlation coefficient with respective to the difference in ranks to get the final value. Note that ranking allows us to be insulated from large value outliers. Additionally, the Spearman rank coefficient is commonly used when either the relationship between the two variables is non-linear or the data is not normally distributed. The coefficient ranges between -1 to 1, where negative values indicate a negative monotonic relationship, 0 indicates no monotonic relationship, and positive values indicate a positive monotonic relationship.

NAS Assistant performs better than the baseline and comparably to some categories of GENNAPE. As seen in Table 3.5 (a), the green represents model families for NAS assistant that perform better than the K-GNN model described in Mills et al.[42]. The bold coefficient reflects the best value across all columns. We see that for many model families, the full GENNAPE outperforms our NAS Assistant Predictor. However, we should note that we did not train a separate multi-layered perception for each cluster regressor classification. As a result, our model is much smaller, allowing faster inference time. It is unlikely that the choice of clustering determines the entire gap in performance, because C-Fuzzy means and Gaussian mixture models work similarly as K-means are a special case of GMMs and C-Fuzzy means an extension of K-means. Lastly, our encoder has a more rigorous training routine which achieves similarly-shaped family embeddings to those from Mills et al.[42]. Therefore, we attribute the gap in performance here to the differential in the number of weights in each model.

Normalized Discounted Cumulative Gain (NCDG). This metric is often used in NAS as well as recommendation systems and information retrieval. This is because it measure the quality of the information that has been ranked based on the ground truth (in NAS this would be model performance metrics).

Family	K-GNN	GENNAPE	NAS Assist
NB201	0.4930	0.8146	**0.8685**
NB301	0.0642	0.3214	**0.6570**
HiAML	-0.1211	**0.4331**	0.1300
Inception	-0.2045	**0.4249**	-0.0448
Two Path	0.1970	**0.3413**	0.19987
OFA RN	**0.5721**	0.5115	0.3071
OFA PN	0.0703	**0.8213**	0.7096
OFA MBv3	0.4345	**0.8660**	0.2346

(a) SRCC Table for Zero-shot Prediction on Architecture Families

Family	K-GNN	GENNAPE	NAS Assist
NB201	0.9270	0.9793	**0.9851**
NB301	0.5341	0.7885	**0.9217**
HiAML	0.5088	**0.6892**	0.6656
Inception	0.6064	**0.8150**	0.6046
Two Path	0.6339	**0.8275**	0.6650
OFA RN	**0.9470**	0.6606	0.7528
OFA PN	0.4426	0.8736	**0.9011**
OFA MBv3	0.8464	**0.9234**	0.6749

(b) NDCG10 Table for Zero-shot Prediction on Architecture Families

Figure 3.5: Our custom performance predictor for NAS Assistance performs comparably in some categories to GENNAPE and far better than the standard k-GNN. Note: The green highlight means that the NAS Assistant Model performs better than the baseline and bold means that the value is the best within the specified category. For both measures, higher is better.

We normalize the metric by taking the discounted cumulative gain (DCG) and the ideal discounted cumulative gain (IDCG) which measures the best possible retrieval for the top 10 ranks and divide the DCG by the IDCG in order to get the NDCG10 metric. Note that DCG here is the sum of the relavance scores (ie. accuracy) for the top models. The higher the value, the more likely it is that our algorithm was able to correctly identify the best models.

NAS Assistant performs comparably to GENNAPE and better than the k-GNN baseline given nDCG scores. We see that from table 3.5 (b), NAS-Assist performs better in a majority of the categories in zero-shot performance as compared to the baseline (shown in green) and better in 4 categories versus worse in 4 compared to GENNAPE. Once again, this result may be due to the fact that NAS-Assist has a smaller set of weights given that we

do not train separate MLP-E for each cluster assignment since that is compute heavy and not as generalizable.

3.3 Generative AI Prompt Tuning

In order to evaluate the neural architectures reliably, we sample 50 architectures from each high performing family based on the zero-shot results from the predictor. Namely this includes NB101, NB201, NB301, and OFA-PN. For each architecture set, we set a threshold to be model_accuracy + $\frac{\sigma}{4}$ based on the standard deviation of the given family. We also set `MAX_ITERATIONS` to be 10. For each run per architecture, we measure the number of iterations it took to converge and the average performance gain per architecture. Additionally, we count the number of times max iterations was reached for each set of models. We present the averages in table 3.6.

	Average Number of Iterations	Average Accuracy Gain	Number of Max outs
NB101	4.32	0.032	2 / 50
NB201	7.91	0.012	14 / 50
NB301	9.36	0.002	32 / 50
OFA PN	9.60	0.001	48 / 50

Figure 3.6: Our table shows that the generative AI model based search methods suffer in performance when the predictor does not have solid granularity.

Better generative code performance comes from better predictive

modelling. We can clearly see from the results that the better the model's zeros-shot predictive performance on the architecture family, the greater the likelihood of the code language model finding a locally optimal model in the neighborhood of the model. NB101 likely had the best performance because the predictor module was directly trained on the dataset. Additionally, NB201 is very similar to NB101 and for that reason it also likely had better convergence time. NB301 performed worse and often converged to "inception" classified models. This may be because inception models are more common in code usage and the generative model could be biased towards them through their training corpus. OFA-PN regularly pursued OFA-RN models and became stuck in the prediction model classifying the performance differences in ResNet poorly due to high amounts of noise. This made the predictions and the resulting generated code unreliable. Once again, it may be because of the generative model's training corpus that the code generated for improvement tended to resemble ResNet architectures which are very common for vision related tasks.

Choice of generative model. Ideally, we would give the model more chances to improve by feeding it back to the generative model, however this was expensive because the OpenAI API is restricted in terms of the number of queries that can be made and the Incoder model from Meta is restricted in the number

```
RESPONSE (Iter 1): Replace conv2d_364 and batch_normalization_370 with
a new conv2d layer with 32 filters, kernel size of 3x3 and padding
of "same" for improved performance. Replace conv2d_366 with a new
conv2d layer with 64 filters, kernel size of 3x3 and padding of "same"
for improved performance. Replace batch_normalization_372 with a
new batch normalization layer for improved performance. Replace
conv2d_368 with a new conv2d layer with 64 filters, kernel size of
3x3 and padding of "same" for improved performance. Remove conv2d_370
and batch_normalization_376 layers as they are not helping performance.
Replace tf.identity_9 with a new global average pooling layer to reduce
the number of parameters and improve performance.
RESPONSE (Iter 2): Removed the tf.identity layer since it has a
negative ranking and does not contribute to the model's performance.
Kept the helpful layers with positive ranking values such as the
batch normalization and ReLU layers in the first convolution block.
Removed or replaced layers with negative ranking values in the second
convolution block to improve the model's performance.
RESPONSE (Iter 3): Removed batch normalization from the first
convolution block (conv1_1_bn) since it has a negative ranking. Removed
batch normalization and ReLU from the second convolution block
(conv2_1_bn and conv2_1_relu) since they have negative rankings. Kept
the layers with positive ranking values, such as the input and the
convolution layers in the first and second blocks. Removed or replaced
layers with negative ranking values in the identity blocks to improve
the model's performance.
```

Figure 3.7: GPT Langauge model displays intuitive reasoning given layer importance.

of tokens it can parse and queries it can remember. For this reason we opted to stay with `GPT Da-Vinci-002 (3.5-Turbo)`. Additionally, the benefit of using a GPT model is that we could request a worded explanation of layers that were changed. For example, figure 3.7 is example of the model explaining the series of changes made to a NB101 model.

The generative code model correctly interprets relevance weights

and attributes them to correct lines of code. Within the text of figure 3.7 we see that the model logically processes the weights and correctly interprets areas of the model that are clearly over-regularizing and hurting performance while also finding a way to replace lower performing layers with blocks that correspond to better performance. The model also seems to reason about the number of weights provided by each layer and actively removes layers that adds an unnecessary number of parameters to the model. This leads us to believe that with greater granularity in the information provided by the predictor regarding the ranking of components in the model, the large language model will have the capability to modify the code in an effective way given the additional information.

Preventing the generative model from changing the model type completely. Given prompt tuning, we can prevent the model from radically changing the structure of the user input to a standard architecture by requiring that the same types of operations are used in the new model and that the number of parameters is not increased significantly. This logic is found our prompt when we specify that the model must "Come up with a new model based off the ranking and the code that follows while maintaining the same types of operations." For OFA - PN we had to make the prompt more strict by specifying

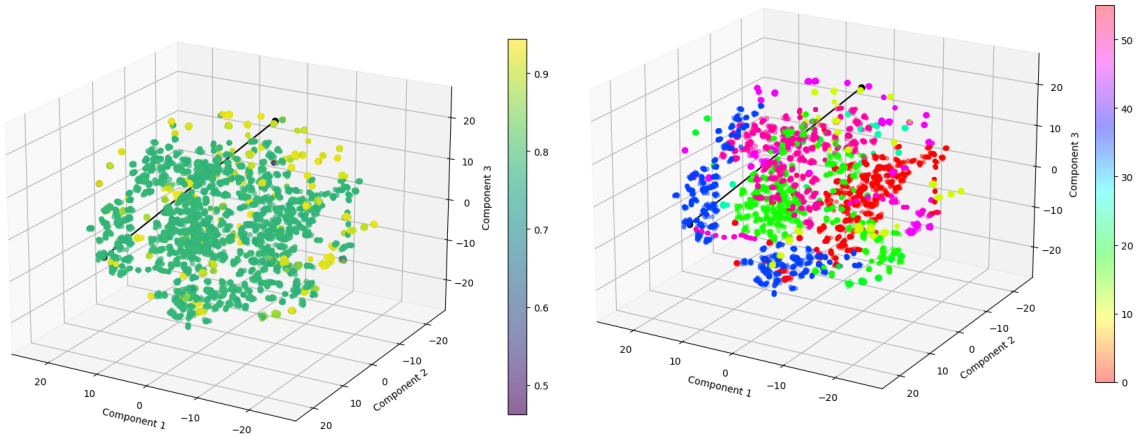
which operations were acceptable in order to prevent the generative code model from just replacing the code with a ResNet structure. In figure 3.8 we provide an example of what the user interface is meant to look like locally, where the generative model edits some lines of code after being prompt constrained. Although this is a prototype model, in the future we hope to provide a more intuitive interface through visual studio or other capable text editor applications.

```
from tensorflow.keras.layers import *

# NAS - Assist (Performance Gain 0.8539 -> 0.8646 (CIFAR-10 Based))

 inputs = Input(shape=input_shape, name="input")
-x = Conv2D(filters=16, kernel_size=(3, 3), activation='relu')(inputs)
-x = BatchNormalization()(x)
-x = MaxPooling2D(pool_size=(2, 2))(x)
-x = Conv2D(filters=8, kernel_size=(3, 3), activation='relu')(x)
-x = MaxPooling2D(pool_size=(2, 2))(x)
+x = Conv2D(filters=32, kernel_size=(3, 3), activation='relu')(inputs)
+x = Conv2D(filters=16, kernel_size=(3, 3), activation='relu')(x)
 x = Flatten()(x)
 x = Dense(units=128, activation='relu')(x)
 x = Dense(units=128, activation='relu')(x)
+x = BatchNormalization()(x)
 x = Dense(units=64, activation='relu')(x)
+x = Dense(units=32, activation='relu')(x)
 x = Dropout(0.3)(x)
-x = Dense(units=32, activation='relu')(x)
+x = Dense(units=16, activation='relu')(x)
 x = Dropout(0.5)(x)
 outputs = Dense(units=10, activation='softmax', name='final')(x)
```

Figure 3.8: NAS-Assist Model edit example where the model architecture is constrained from being changed completely through prompt tuning.

We remove the limitations on search space with generalizability. We construct a generalized performance predictor that can access the performance of an architecture and is capable of rendering a computation graph. We also

incorporate a search engine powered by generative AI that is not limited to a specific search space other than the artificial limitations imposed by our prompt tuning that constricts the code changes to the local architecture space. This allows us to improve models by suggesting reasonable edits that remain within a close distance in our latent embedding space. As seen in figure 3.9, we have an architecture that started out in the NB201c10 space and then the search algorithm was able to use the generative model to take the architecture and find a close, yet better performing architecture in an adjacent OFA ResNet family. This demonstrates the power of extreme generalizability, where our schema is not hindered by the structure of an arbitrary search space.



(a) TSNE Breakdown of Combined NB201 and OFA ResNet Space with Accuracy Shading

(b) TSNE Breakdown of Combined NB201 and OFA ResNet Space with Unsupervised Cluster Assignment Shading

Figure 3.9: We see here that our search algorithm that moved from one search space of models to an adjacent one in order to optimize performance. By doing so, we demonstrate the uniqueness in our algorithm of being able to explore and evaluate outside of the starting search space. Note that the black line represents our search route over 3 iterations and starts from a region with lower accuracies and ends in a region with higher accuracies.

4

Conclusion

We present NAS-Assistant, the first NAS search and evaluation tool using generative code language models to enable a search space agnostic architecture optimizer tool. Through a combination of NAS methods, we present an algorithm that uses a novel layer relevance metric that enables users to add "performance intuition" to code generation models. The flexibility of the tool offers the ability

for NAS-Assistant to be applied to any computer vision -related architecture optimization problem.

Current Issues. There are two main weaknesses of this model. The first is the capability of the performance predictor. As we saw in previous experiments, if the performance predictor is even slightly worse in zero-shot inference, then the model can get stuck in performance prediction loops where there is too much noise in the prediction to achieve convergence. The second issue is that the regressor has currently only trained on each model's CIFAR-10 performance. Ideally, we would expand this to other vision datasets and then use zero-cost metrics to create performance bound estimates that can infer the performance decrease or increase given the similarity of the test dataset.

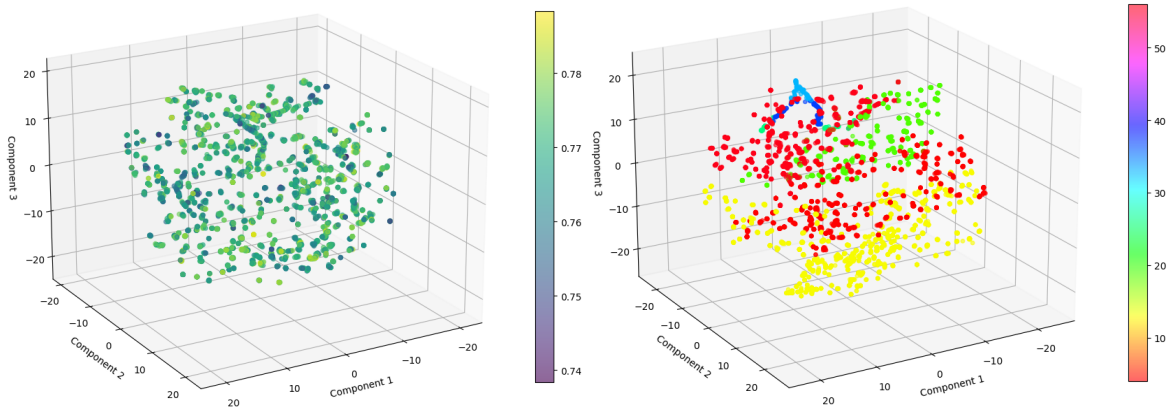
Future work. Given that the version of NAS-Assistant presented in this paper is a prototype, we seek to complete work to make this a full distributable software. This entails adding zero-cost performance metrics for ground-truth measurement of data set similarity and accuracy error bounding. We also plan to incorporate this work with an adjacent research project that uses multi-fidelity optimization to improve model evaluation. By integrating both projects, NAS-Assistant can be used a prior distribution generator.

Through this paper, we establish that it is possible to integrate the advances

of generative AI into NAS and create search space agnostic architecture optimization tools. Moving forward, we aim to further integrate generative AI methods in NAS in order to enhance existing and future search and evaluation methodologies.

A

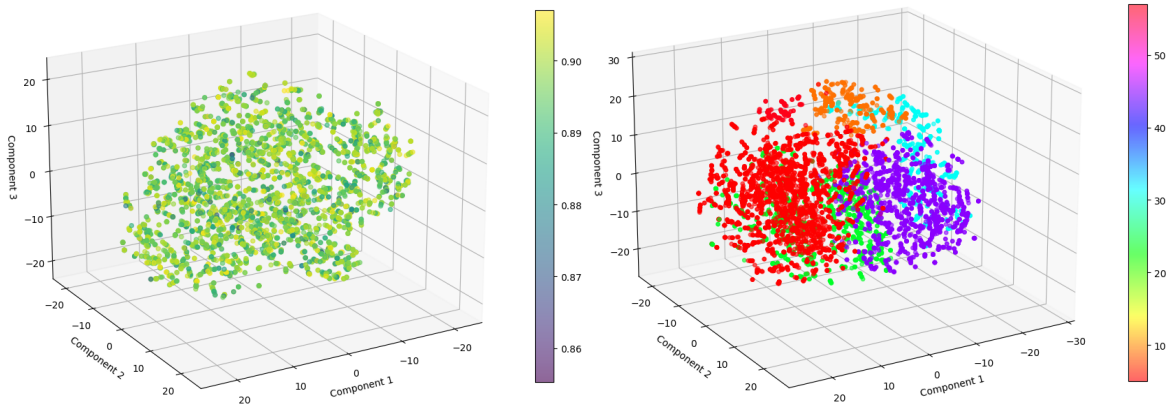
Appendix



(a) TSNE Breakdown of OFA MBv3 Space with Accuracy Shading

(b) TSNE Breakdown of OFA MBv3 Space with Unsupervised Cluster Assignment Shading

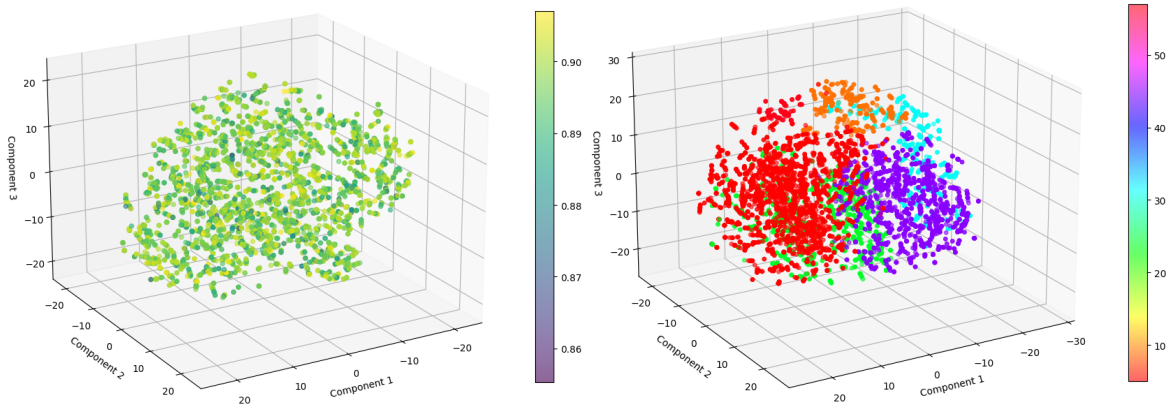
Figure A.1: OFA MBv3 Clustering and Encoding.



(a) TSNE Breakdown of Two Path Space with Accuracy Shading

(b) TSNE Breakdown of Two Path Space with Unsupervised Cluster Assignment Shading

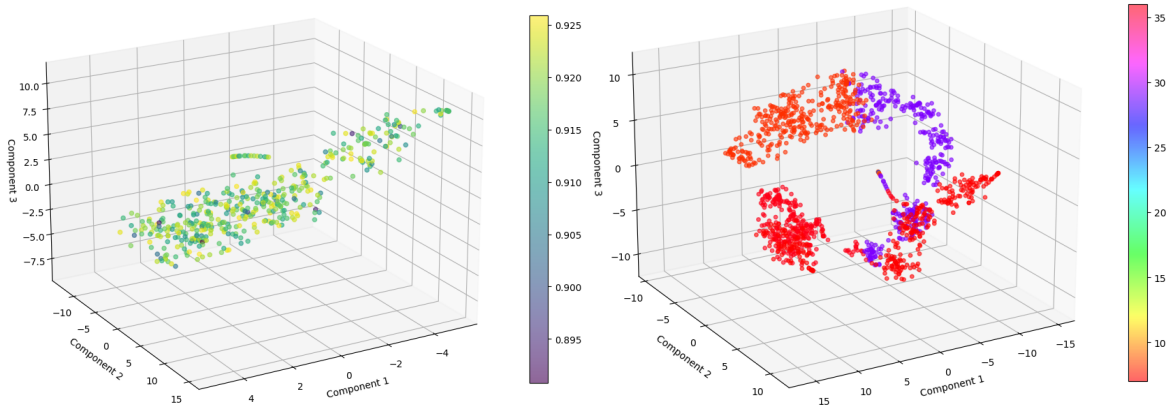
Figure A.2: OFA Two Path Clustering and Encoding.



(a) TSNE Breakdown of Two Path Space with Accuracy Shading

(b) TSNE Breakdown of Two Path Space with Unsupervised Cluster Assignment Shading

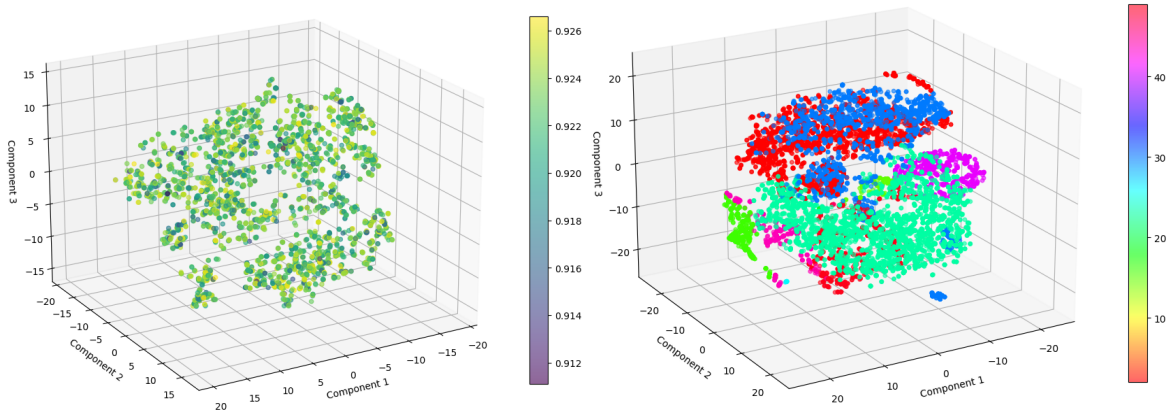
Figure A.3: Two Path Clustering and Encoding.



(a) TSNE Breakdown of Inception Space with Accuracy Shading

(b) TSNE Breakdown of Inception Space with Unsupervised Cluster Assignment Shading

Figure A.4: Inception Clustering and Encoding.



(a) HiAML Breakdown of Two Path Space with Accuracy Shading

(b) TSNE Breakdown of HiAML Space with Unsupervised Cluster Assignment Shading

Figure A.5: HiAML Clustering and Encoding.

References

[1] Abdelfattah, M. S., Mehrotra, A., Dudziak, Ł., & Lane, N. D. (2021). Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*.

[2] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

[3] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

[4] Bai, J., Lu, F., Zhang, K., et al. (2019). Onnx: Open neural network exchange. https://github.com/onnx/onnx.

[5] Bohdal, O., Balles, L., Ermis, B., Archambeau, C., & Zappella, G. (2022). Pasha: Efficient hpo with progressive resource allocation. *arXiv preprint arXiv:2207.06940*.

[6] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877–1901.

[7] Cai, H., Gan, C., Wang, T., Zhang, Z., & Han, S. (2019). Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*.

[8] Cai, H., Zhu, L., & Han, S. (2018). Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*.

[9] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[10] Chen, X., Xie, L., Wu, J., & Tian, Q. (2019). Progressive darts: Bridging the optimization gap for nas in the wild.

[11] Cheng, W., Greaves, C., & Warren, M. (2006). From n-gram to skipgram to concgram. *International journal of corpus linguistics*, 11(4), 411–433.

[12] Cui, P., Wang, X., Pei, J., & Zhu, W. (2018). A survey on network embedding. *IEEE transactions on knowledge and data engineering*, 31(5), 833–852.

[13] Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29.

[14] Domhan, T., Springenberg, J. T., & Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*.

[15] Dong, X. & Yang, Y. (2020). Nas-bench-201: Extending the scope of reproducible neural architecture search.

[16] Dwivedi, V. P. & Bresson, X. (2020). A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*.

[17] Eggensperger, K., Müller, P., Mallik, N., Feurer, M., Sass, R., Klein, A., Awad, N., Lindauer, M., & Hutter, F. (2021). Hpobench: A collection of reproducible multi-fidelity benchmark problems for hpo.

[18] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

[19] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., & Lewis, M. (2022). Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

[20] Gao, X., Gürbüzbalaban, M., & Zhu, L. (2021). Global convergence of stochastic gradient hamiltonian monte carlo for nonconvex stochastic optimization: Nonasymptotic performance bounds and momentum-based acceleration. *Operations Research*.

[21] Goldberg, Y. & Levy, O. (2014). word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

[22] Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., & Sculley, D. (2017). 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).

[23] Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*.

[24] Hammond, D. K., Vandergheynst, P., & Gribonval, R. (2011). Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2), 129–150.

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

[26] Hu, Y., Liang, Y., Guo, Z., Wan, R., Zhang, X., Wei, Y., Gu, Q., & Sun, J. (2020). Angle-based search space shrinking for neural architecture search.

[27] Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. (2017). Population based training of neural networks. *arXiv preprint arXiv:1711.09846*.

[28] Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *International conference on machine learning* (pp. 5110–5121).: PMLR.

[29] Kipf, T. N. & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

[30] Klein, A., Falkner, S., Springenberg, J. T., & Hutter, F. (2016). Learning curve prediction with bayesian neural networks.

[31] Koonce, B. & Koonce, B. (2021). Mobilenetv3. *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, (pp. 125–144).

[32] Lee, H., Lee, G., Kim, J., Cho, S., Kim, D., & Yoo, D. (2022). Improving multi-fidelity optimization with a recurring learning rate for hyperparameter tuning. *arXiv preprint arXiv:2209.12499*.

[33] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1), 6765–6816.

[34] Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-tzur, J., Hardt, M., Recht, B., & Talwalkar, A. (2020). A system for massively parallel hyperparameter tuning. In *Third Conference on Systems and Machine Learning*.

[35] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. (2022). Competition-level code generation with alphacode. *Science*, 378(6624), 1092–1097.

[36] Lin, M., Wang, P., Sun, Z., Chen, H., Sun, X., Qian, Q., Li, H., & Jin, R. (2021). Zen-nas: A zero-shot nas for high-performance image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 347–356).

[37] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., & Kavukcuoglu, K. (2017). Hierarchical representations for efficient architecture search.

[38] Liu, H., Simonyan, K., & Yang, Y. (2018). Darts: Differentiable architecture search.

[39] Luo, R., Tan, X., Wang, R., Qin, T., Chen, E., & Liu, T.-Y. (2020). Semi-supervised neural architecture search. *Advances in Neural Information Processing Systems*, 33, 10547–10557.

[40] Ma, L., Cui, J., & Yang, B. (2019). Deep neural architecture search with deep graph bayesian optimization. In *IEEE/WIC/ACM International Conference on Web Intelligence* (pp. 500–507).

[41] Mallat, S. (1999). *A wavelet tour of signal processing*. Elsevier.

[42] Mills, K. G., Han, F. X., Zhang, J., Chudak, F., Mamaghani, A. S., Salameh, M., Lu, W., Jui, S., & Niu, D. (2022). Gennape: Towards generalized neural architecture performance estimators.

[43] Pan, S., Hu, R., Long, G., Jiang, J., Yao, L., & Zhang, C. (2018). Adversarially regularized graph autoencoder for graph embedding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (pp. 2609–2615).: International Joint Conferences on Artificial Intelligence Organization.

[44] Panch, T., Szolovits, P., & Atun, R. (2018). Artificial intelligence, machine learning and health systems. *Journal of global health*, 8(2).

[45] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 701–710).

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

[47] Ren, P., Xiao, Y., Chang, X., Huang, P.-y., Li, Z., Chen, X., & Wang, X. (2021). A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Comput. Surv.*, 54(4).

[48] Schmucker, R., Donini, M., Zafar, M. B., Salinas, D., & Archambeau, C. (2021). Multi-objective asynchronous successive halving. *arXiv preprint arXiv:2106.12639*.

[49] Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J., & Zhang, T. (2020). Bridging the gap between sample-based and one-shot neural architecture search with bonas. *Advances in Neural Information Processing Systems*, 33, 1808–1819.

[50] Shuman, D. I., Narang, S. K., Frossard, P., Ortega, A., & Vandergheynst, P. (2013). The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE signal processing magazine*, 30(3), 83–98.

[51] Siems, J., Zimmer, L., Zela, A., Lukasik, J., Keuper, M., & Hutter, F. (2020a). Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777*.

[52] Siems, J., Zimmer, L., Zela, A., Lukasik, J., Keuper, M., & Hutter, F. (2020b). Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777*.

[53] Sun, F.-Y., Hoffmann, J., Verma, V., & Tang, J. (2019). Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. *arXiv preprint arXiv:1908.01000*.

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

[55] Theis, T. N. & Wong, H.-S. P. (2017). The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2), 41–50.

[56] Tu, R., Khodak, M., Roberts, N., & Talwalkar, A. (2021). Nas-bench-360: Benchmarking diverse tasks for neural architecture search. *arXiv preprint arXiv:2110.05668*.

[57] Tu, R., Khodak, M., Roberts, N. C., & Talwalkar, A. (2022). NAS-bench-360: Benchmarking diverse tasks for neural architecture search.

[58] Velickovic, P., Fedus, W., Hamilton, W. L., Liò, P., Bengio, Y., & Hjelm, R. D. (2019). Deep graph infomax. *ICLR (Poster)*, 2(3), 4.

[59] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

[60] Wei, C., Niu, C., Tang, Y., Wang, Y., Hu, H., & Liang, J. (2022). Npenas: Neural predictor guided evolution for neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*.

[61] White, C., Neiswanger, W., & Savani, Y. (2021a). Bananas: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35 (pp. 10293–10301).

[62] White, C., Zela, A., Ru, R., Liu, Y., & Hutter, F. (2021b). How powerful are performance predictors in neural architecture search? *Advances in Neural Information Processing Systems*, 34, 28454–28469.

[63] Wuest, T., Weimer, D., Irgens, C., & Thoben, K.-D. (2016). Machine learning in manufacturing: advantages, challenges, and applications. *Production & Manufacturing Research*, 4(1), 23–45.

[64] Xie, S., Kirillov, A., Girshick, R., & He, K. (2019). Exploring randomly wired neural networks for image recognition.

[65] Xu, B., Shen, H., Cao, Q., Qiu, Y., & Cheng, X. (2019). Graph wavelet neural network. *arXiv preprint arXiv:1904.07785*.

[66] Xu, C., Tao, D., & Xu, C. (2013). A survey on multi-view learning. *arXiv preprint arXiv:1304.5634*.

[67] Yan, S., Zheng, Y., Ao, W., Zeng, X., & Zhang, M. (2020). Does unsupervised architecture representation learning help neural architecture search? *Advances in Neural Information Processing Systems*, 33, 12486–12498.

[68] Yang, C., Liu, Z., Zhao, D., Sun, M., & Chang, E. Y. (2015). Network representation learning with rich text information. In *IJCAI*, volume 2015 (pp. 2111–2117).

[69] Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., & Hutter, F. (2019). Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning* (pp. 7105–7114).: PMLR.

[70] Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., & Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31.

[71] Yu, K., Sciuto, C., Jaggi, M., Musat, C., & Salzmann, M. (2019). Evaluating the search phase of neural architecture search.

[72] Zhang, D., Yin, J., Zhu, X., & Zhang, C. (2018). Network representation learning: A survey. *IEEE transactions on Big Data*, 6(1), 3–28.

[73] Zhong, Z., Yan, J., Wu, W., Shao, J., & Liu, C.-L. (2017). Practical block-wise neural network architecture generation.

[74] Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the*

IEEE conference on computer vision and pattern recognition (pp. 8697–8710).

Good, Better, Best
Let us never rest
Until our good is better
And our better is best.