Neural Predictor for Neural Architecture Search

Wei Wen[1,2⋆], Hanxiao Liu[1], Yiran Chen[2], Hai Li[2]
Gabriel Bender[1], Pieter-Jan Kindermans[1]

[1] Google Brain, [2] Duke University

Abstract. Neural Architecture Search methods are effective but often use complex algorithms to come up with the best architecture. We propose an approach with three basic steps that is conceptually much simpler. First we train N random architectures to generate N (architecture, validation accuracy) pairs and use them to train a regression model that predicts accuracies for architectures. Next, we use this regression model to predict the validation accuracies of a large number of random architectures. Finally, we train the top-K predicted architectures and deploy the model with the best validation result. While this approach seems simple, it is more than 20× as sample efficient as Regularized Evolution on the NASBench-101 benchmark. On ImageNet, it approaches the efficiency of more complex and restrictive approaches based on weight sharing such as ProxylessNAS while being fully (embarrassingly) parallelizable and friendly to hyper-parameter tuning.

Keywords: Neural Architecture Search, Automated Machine Learning, Graph Neural Networks, NASBench-101, Mobile Models, ImageNet

1 Introduction

Early Neural Architecture Search (NAS) methods showed impressive results, allowing researchers to automatically find high-quality neural networks within human-defined search spaces [22, 23, 18, 17]. However, these early methods required thousands of models to be trained from scratch to run a single search, making the methods prohibitively expensive for most practitioners. Thus, algorithms which can improve the *sample efficiency* are of high value. A second consideration when designing a NAS algorithm is *friendliness to hyper-parameter tuning*. In many existing approaches – such as those based on Reinforcement Learning (RL) [22] or Evolutionary Algorithms (EA) [17], trying out a new set of hyper-parameters for the search algorithm requires us to train and evaluate a new set of neural network from scratch. Ideally, we would be able to train a single set of neural networks, evaluate them once, and then use the results to try out many different hyper-parameter configurations for the search algorithm. A third design consideration of NAS is *full parallelizability* (or *embarassing parallelizability)*: existing methods based on RL [22], EA [17] and Bayesian Optimization (BO) [6] are complex to implement, requiring complex coordination

⋆ Work done as a Research Intern and Student Researcher in Google Brain.

between tens or hundreds of workers when collecting reward/fitness/acquisition during a search. An fully parallelizable algorithm can avoid this coordination and accelerate the search when idle computing resources are available. Ideally, we pursue an algorithms with the above merits – *sample efficiency*, *friendliness to hyper-parameter tuning*, and *full parallelizability*.

To design a NAS algorithm possessing the three merits described above, we investigate how well it is possible to do using a combination of two techniques which are ubiquitous in the ML community: supervised learning and random sampling. We show that an algorithm that intelligently combines these two approaches can be surprisingly effective in practice. With an infinite compute budget, a very simple but naïve approach to architecture search would be to sample tons of random architectures, train and evaluate each one, and then select the architectures with the best validation set accuracies for deployment; this is a straightforward application of the ubiquitous random search heuristic. It is *friendly to hyper-parameter tuning* and is *fully parallelizable*. However, the *efficiency* (computational requirements) of this approach makes it infeasible in practice. For example, to exhaustively train and evaluate all of the $423,624$ architectures in the NASBench-101 [21], it would take roughly 25 years of TPU training time. Only a small number of companies and corporate research labs can afford this much compute, and it is far out of reach for most ML practitioners.

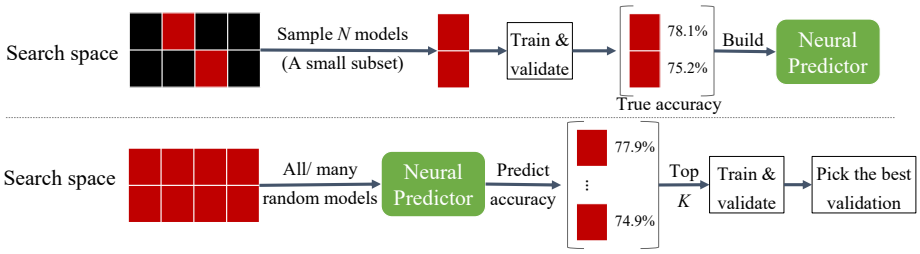

Fig. 1: Building (top) and applying (bottom) the Neural Predictor.

One way to alleviate this is to identify a small subset of promising models. If this is done with reasonably high precision (i.e., most models selected are indeed of high quality) then we can train and validate just this limited set of models to reliably select a good one for deployment. To achieve this, the proposed Neural Predictor uses the following steps to perform an architecture search:

(1) Build a predictor by training N random architectures to obtain N (architecture, validation accuracy) pairs. Use this data to train a regressor.

(2) Quality prediction using the regression model over a large set of random architectures. Select the K most promising architectures for final validation.

(3) Final validation of the top K architectures by training them. Then we select the architecture with the highest validation accuracy to deploy.

The workflow is illustrated in Figure 1. In this setup, the first step is a traditional regression problem where we first generate a dataset of N samples to train on. The second step can be carried out efficiently because evaluating a model using the predictor is cheap and parallelizable. The third step is nothing

more than traditional validation where we only evaluate a well curated set of K models. While the method outlined above might seem straightforward, this solution is surprisingly effective and satisfies the three goals discussed above:

- *Efficiency*: The Neural Predictor strongly outperforms random search on NASBench-101. It is also about 22.83 times as sample-efficient as Regularized Evolution – the best performing method in the NASBench-101 paper. The Neural Predictor can easily handle different search spaces. In addition to NASBench-101, we evaluated it on the ProxylessNAS [5] search space and found that the predicted architecture is as accurate as ProxylessNAS and clearly better than random search.
- *Friendliness to hyper-parameter tuning*: All hyper-parameters of the regression model are cross validated by the dataset collected just once in step **(1)**. The cost of tuning those hyper-parameters is small because the predictor model is small.
- *Full parallelizability*: The most computationally intensive components of the method (training N models in step **(1)** and K models in step **(3)**) are trivially parallelizable when sufficient computation resources are available.

Furthermore, the architecture selection process uses two of the most ubiquitous tools from the ML toolbox: random sampling and supervised learning. In contrast, many existing NAS approaches rely on more advanced techniques such as RL, EA, BO, and weight sharing.

2 Related Work

Neural Architecture Search was proposed to automate the design of neural networks, by searching models in a design space using techniques such as RL [22], EA [18] or BO [9, 6]. A clear limitation of the early approaches is their computation efficiency. Thus, recent methods often focus on efficient NAS by using weight sharing [2, 4, 16, 5, 14]. As aforementioned, when comparing with previous works, Neural Predictor is friendly to hyper-parameter tuning, conceptually simple and fully parallelizable. Moreover, our Neural Predictor can potentially work as a surrogate model to accelerate accuracy acquisition of architectures during the search in RL, EA and BO, or during the candidate architecture evaluation in weight sharing approaches [2].

The idea of predictive models of the accuracy, which we use, has been explored in prior works. In [7] an LSTM was used to generate a feature representation of an architecture, which was subsequently used to predict the quality. The one-shot approach by Bender et al. [2] used a weight sharing model to predict the accuracy of an individually trained architecture. Baker et al. [1] used predictive models to perform early stopping to speed up architecture and hyper-parameter optimization. NAO [15] used both a learned representation and a predictor to search for high quality architecture. PNAS [13] progressively trained a predictor to accelerate the search. A key difference between PNAS and Neural Predictor is that in PNAS the predictor is only a small component in a large traditional

NAS system. In PNAS, the predictor and the models are trained over time as the architectures become more complex. Because of this, the PNAS approach cannot be completely parallelized. Another popular approach combined a predictor with Bayesian Optimization [6]. Unlike above methods, which used ν-SVR/random forests [1], multi-layer perceptrons [13], LSTMs [15, 7] or Gaussian processes [6], we use a Graph Convolutional Network (GCN) [10] for our regression model. GCNs are naturally permutation-invariant, capturing the intuition that an architecture under different node permutations should have the same predicted accuracy. Furthermore, we show strong results can be achieved without the use of advanced techniques such as RL, EA, BO or NAO [15], enabling our merits of *friendliness to hyper-parameter tuning* and *full parallelizability*. Finally, because the effectiveness of NAS has been questioned [12], we include random baseline to show that the search spaces used in this work are meaningful and that while the proposed approach is simple, it is clearly better than a random approach.

While preparing the final version of this paper, we found a concurrent work [19] which also used a GCN to predict accuracy. However, a *node* in their graph is a model in the search space; as the search space is usually huge, their method is hard to scale up. In our design, a *graph* represents a model and the graph size is approximately proportional to the number of layers in a model. Therefore, our Neural Predictor is able to scale to huge search spaces, such as the ProxylessNAS ImageNet search space with the size of 6.64×10^{17}.

3 Neural Predictor

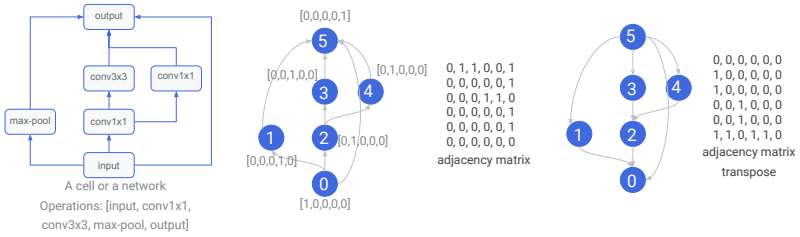

Fig. 2: An illustration of graph and node representations. Left: A neural network architecture with 5 candidate operations per node. Each node is represented by a one-hot code of its operation. The one-hot codes are inputs of a bidirectional GCN, which takes into account both the original adjacency matrix (middle) and its transpose (right).

The core idea behind the Neural Predictor is that carrying out the actual training and validation process is the most reliable way to find the best model. The goal of the Neural Predictor is to provide us with a curated list of promising models for final validation prior to deployment. The entire Neural Predictor process is outlined below.

Step 1: Build the predictor using N samples. We train N models to obtain a small dataset of (architecture, validation accuracy) pairs. The dataset is then used to train a regression model that maps an architecture to a predicted validation accuracy.

Step 2: Quality prediction. Because architecture evaluation using the learned predictor is efficient and trivially parallelizable, we use it to rapidly predict the accuracies of a large number of random architectures. We then select the top K predicted architectures for final validation.

Step 3: Final validation on K samples. We train and validate the top K models in the traditional way. This allows us to select the best model based on the actual validation accuracy. Even if our predictor is somewhat noisy, this step allows us to use a more reliable measurement to select our final architecture for deployment.

Training $N+K$ models is by far the most computationally expensive part of the Neural Predictor. If we assume a constant compute budget, N and K are key hyper-parameters which need to be set; we will discuss this next. Note that the two most expensive steps (Step 1 and Step 3) are both fully parallelizable.

3.1 Hyper-parameters in the Workflow

Hyper-parameters for model training are always needed if we train a single model in the search space. In this respect the Neural Predictor is no different from other methods. We found that using the same hyper-parameters for all models we train is an effective strategy, and was also used in NASBench-101.

Trade-off between N and K for a fixed budget: For a given compute budget, the total number of architectures we train and evaluate, $N + K$, must remain fixed. However, we can trade off between N (the number of samples used to train the predictor) and K (the number of samples used for final evaluation). If N is too small, the predictor's outputs will be very noisy and will not provide a reliable signal for the search. As we increase N, the predictor will become more accurate; however, increasing N requires us to decrease K. If K is small, the predictor must very reliably identify high-quality models from the search space. As K increases, we will be able to tolerate larger noise in the predictor, and the predictor's ability to precisely rank architectures in the search space will become less important. Because it is difficult to theoretically predict the optimal trade-off between N and K, we will investigate this in the experimental setting.

To find a lower bound on N we can start with a small number of samples, and iteratively increase N until we observe a good cross-validation accuracy. This means that contrast to some other methods such as Regularized Evolution [17], ENAS [16], NASNet [23], ProxylessNAS [5], there is no need to repeat the entire search experiment in order to tune this hyper-parameter. The same applies to the hyper-parameters and the architecture of the Neural Predictor itself.

The hyper-parameters of the Neural Predictor can be optimized by cross-validation using the N training samples. The cost of training a neural predictor including the hyper-parameter tuning is negligible compared to the cost of training image models. It takes 25 seconds to train a neural predictor on

$N = 172$ samples from NASBench-101. The mean (resp. median) training time of a CIFAR-10 model in NASBench-101 is 32 (resp. 26) minutes. At the cost of training two CIFAR-10 models (about one hour), we could try 144 hyper-parameter configurations for the neural predictor. In contrast, RL or EA require us to train more models in order to try out a new hyper-parameter configuration.

3.2 Modeling by Graph Convolutional Networks

We tried many options for the architecture of the predictor. We find Graph Convolutional Networks (GCNs) work best. Due to space constraints we will limit our discussion to GCNs. A comparison against other regression models is in the supplementary material. Graph Convolutional Networks (GCNs) are good at learning representations for graph-structured data [10, 20] such as a neural network architecture. The graph convolutional model we use is based on [10], which assumes undirected graphs. We will modify their approach to handle neural architectures represented as directed graphs.

We start with a D_0-dimensional representation for each of the I nodes in the graph, giving us an initial feature vector $\boldsymbol{V}_0 \in \mathbb{R}^{I \times D_0}$. For each node we use a one-hot vector representing the selected operation. An example for NASBench-101 is shown in Figure 2. The node representation is iteratively updated using Graph Convolutional Layers. Each layer uses an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{I \times I}$ based on the node connectivity and a trainable weight matrix $\boldsymbol{W}_l \in \mathbb{R}^{D_l \times D_{l+1}}$:

$$\boldsymbol{V}_{l+1} = \mathrm{ReLU}\left(\boldsymbol{A}\boldsymbol{V}_l\boldsymbol{W}_l\right). \tag{1}$$

Following previous work [10], we add an identity matrix to $\boldsymbol{A}$ (corresponding to self cycles) and normalize it using the node degree.

The original GCNs [10] assume undirected graphs. When applied to a directed acyclic graph, the directed adjacency matrix allows information to flow only in a single direction. To make information flow both ways, we always use the average of two GCN layers: one where we use $\mathbf{A}$ to propagate information in the forward directions and another where we use $\mathbf{A}^T$ to reverse the direction:

$$\boldsymbol{V}_{l+1} = \frac{1}{2}\mathrm{ReLU}\left(\boldsymbol{A}\boldsymbol{V}_l\boldsymbol{W}_l^+\right) + \frac{1}{2}\mathrm{ReLU}\left(\boldsymbol{A}^T\boldsymbol{V}_l\boldsymbol{W}_l^-\right).$$

Figure 2 shows an example of how the adjacency matrices are constructed (without normalization or self-cycles).

GCNs are able to learn high quality node representations by stacking multiple of these layers together. Since we are more interested in the accuracy of the overall network (a global property), we take the average over node representations from the final graph convolutional layer and attach one or more fully connected layers to obtain the desired output. Details are provided in the supplementary.

4 Experiments

In this section we will discuss two studies. First we will analyze the Neural Predictor's behavior in the controlled environment from NASBench-101 [21].

Afterwards we will use our approach to search for high quality mobile models in the ProxylessNAS search space [5].

4.1 NASBench-101

NASBench-101 [21] is a dataset used to benchmark NAS algorithms. The goal is to come up with a high quality architecture as efficiently as possible. The dataset has the following properties: **(1)** train time, validation and test accuracy are provided for all 423,624 models in the search space; **(2)** each model was trained and evaluated three times. This allows us to look at the variance across runs; **(3)** all models were trained in a consistent manner, preventing biases from the implementation from skewing results; **(4)** NASBench-101 recommends using only validation accuracies during a search, and reserving test accuracies for the final report; this is important to avoid overfitting.

NASBench-101 uses a cell-based NAS [23] on CIFAR-10 [11]. Each cell is a Directed Acyclic Graph (DAG) with up to 7 nodes. There is an input node, an output node and up to 5 interior nodes. Each interior node can be a 1×1 convolution (`conv1x1`), 3×3 convolution (`conv3x3`) or max-pooling op (`max-pool`). One example is shown in Figure 2 (left). In each experiment, we use the validation accuracy from a *single run*[1] as a search signal. The single run is uniformly sampled from these three records. This simulates training the architecture once. Test accuracy is only used for reporting the accuracy on the model that was selected at the end of a search. For that model we use the *mean* test accuracy over three runs as the "ground truth" measure of accuracy.

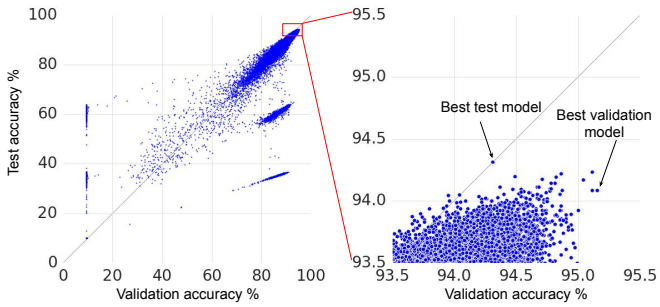

Fig. 3: (Left) Validation vs. test accuracy in NASBench-101. (Right) Zoomed in on the highly accurate region. Each model (point) is the validation accuracy from a single training run. Test accuracies are averaged over three runs. This plot demonstrates that even knowing the validation accuracy of every possible model is not sufficient to predict which model will perform best on the *test* set.

[1] In the training dataset of our Neural Predictor, this means that each model's accuracy label is sampled once and fixed across all epochs.

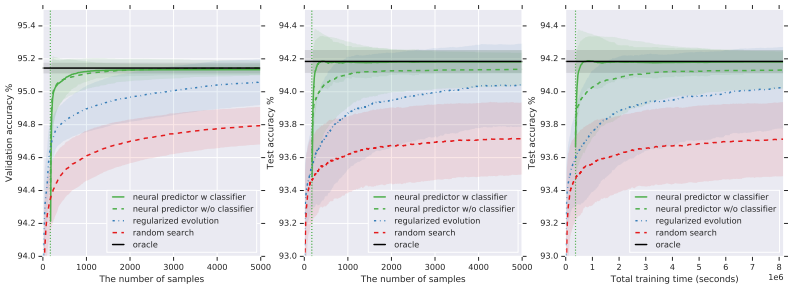

Fig. 4: Comparison of search efficiency among oracle, random search, Regularized Evolution and our Neural Predictor (with and without a two stage regressor). All experiments are averaged over 600 runs. The x-axis represents the total compute budget $N + K$. The vertical dotted line is at $N = 172$ and represents the number of samples (or total training time) used to build our Neural Predictor. From this line on we start from $K = 1$ and increase it as we use more architectures for final validation. The shaded region indicates standard deviation of each search method.

Oracle: an upper bound baseline. Under the assumption of infinite compute, a traditional machine learning approach would be to train and validate all possible architectures to select the best one. We refer to this baseline as the "oracle" method. Figure 3 plots the validation versus the test accuracy for all models. The model that the oracle method would select based on the validation accuracy of 95.15% has a test set accuracy of 94.08%. This means that **the oracle does not select the model with the highest test set accuracy.** The global optimum on the test set is 94.32%. However, since this model cannot be found using extensive validation, one should not expect this model to be found using any NAS algorithm. A more reasonable goal is to reliably select a model that has similar quality to the one selected by the oracle. Furthermore, it is important to realize that **even an oracle approach has variance.** We have three training runs for each model, which allows us to run multiple variations of the "oracle". This simulates the impact of random variations on the final result. Averaged over 100 oracle experiments, where in each experiment we randomly select one of 3 validation results, the best validation accuracy has a mean 95.13% and a standard deviation 0.03%. The test accuracy has a mean of 94.18% and a standard deviation 0.07%.

Random search: a lower bound baseline. Recently, Li *et al.* [12] questioned whether architecture search methods actually outperform random search. Because this depends heavily on the search space and Li *et al.* [12] did not investigate the NASBench-101 search space, we need to check this ourselves. Therefore we replicate the random baseline from NASBench-101 by sampling architectures without replacement. After training, we pick the architecture with the highest validation accuracy and report its result on the test set. Here we observe that

even when we train and validate 2000 models, which requires a massive compute budget, the gap to the oracle is large (Figure 4). For random search the average test accuracy is 93.66% compared to 94.18% for the oracle. This implies that **there is a large margin for improvement over random search.** Moreover, the variance is quite high, with a standard deviation of 0.25%. Finally, evaluating 5000 models in total produces only a small gain over evaluating 2000 models at a high computational cost.

Regularized evolution: a state of the art baseline. In the NASBench-101 paper [21], Regularized Evolution [17] was the best performing method. We replicated those experiments using the open source code and their hyper-parameter settings (available in the supplementary material). **Regularized evolution is significantly better than random** as shown in Figure 4. However even after 2000 models are trained, it is still clearly worse than the oracle (on average) with an accuracy of 93.97% and a standard deviation of 0.26%.

Neural Predictor. Having set our baselines, we now describe the precise Neural Predictor setup and evaluation. The graph representation of a model is a DAG with up to 7 nodes. Each node is represented by an one-hot code of "[`input`, `conv1x1`, `conv3x3`, `max-pool`, `output`]". The GCN has three Graph Convolutional layers with the constant node representation size D and one hidden fully-connected layer with output size 128. Finally, the accuracy we need to predict is limited to a finite range. While it is not that common for regression, we can force the network to make predictions in this finite range by using a sigmoid at the output layer. Specifically, we use a sigmoid function that is scaled and shifted such that its output accuracy is always between 10% and 100%.

All hyper-parameters for the predictor are first optimized using cross-validation where $\frac{1}{3}N$ samples were used for validation. After setting the hyper-parameters, we use all N samples to train the final predictor. At this point we heuristically increase the node representation size D of the predictor such that the number of parameters in the Neural Predictor is also $1.5\times$ as large. Specific N and D values and other training details are in the supplementary material. The models selected for final evaluation are always trained in the same way, regardless of the method (baseline or predictor) that selected the model, to ensure fairness.

A two stage predictor. Looking at a small dataset of $N = 172$ models in Figure 6 (left) during cross-validation,[2] we realized that for NASBench-101 a two stage predictor is needed. The NasBench-101 dataset contains many models that are not stable during training or perform very poorly (e.g. a model with only pooling operations). The two stage predictor, shown in Figure 5, filters obviously bad models first by predicting whether each model will achieve an accuracy above 91%. This allows the the second stage to focus on a narrower accuracy range. It makes training the regression model easier, which in turn

[2] In our implementation, we split the NASBench-101 dataset to $10,000$ shards and each shard has 43 samples. The $N = 172$ comes from a random 4 shards.

makes it more reliable. Both stages share the same GCN architecture but have different output layers. A classifier trained on these $N = 172$ models has a low False Negative Rate as shown in Figure 6 (right). This implies that the classifier will filter out very few actually good models.

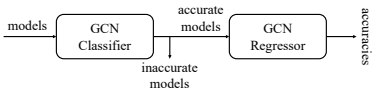

Fig. 5: Neural Predictor on NASBench-101. It is a cascade of a classifier and a regressor. The classifier filters out inaccurate models and the regressor predicts accuracies of accurate models.

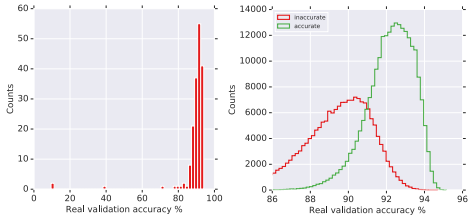

Fig. 6: The classifier filtering out inaccurate models in NASBench. 172 models (left) are sampled to build the classifier, which is tested by unseen data (right).

The two stage approach improves the results but is not required. If we only use a single stage, the MSE for the validation accuracy is 1.95 (averaged over 10 random splits). By introducing the filtering stage this reduces to 0.66. In Figure 4 we observe that even without the filtering stage the predictor clearly outperforms the random search baseline and regularized evolution. Therefore the two stage approach should be seen as a non-essential fine-tuning of the proposed method. Using only a single stage is more elegant, but adding the second stage gives additional performance benefits.

Results using N=172 (or 0.04% of the search space) for training are shown in Figure 4. We used $N = 172$ models to train the predictor. Then we vary K, the number of architectures with the highest predicted accuracies to be trained and validated to select the best one. Therefore, "the number of samples" in the figure equals $N + K$ for Neural Predictor. In Figure 4 (left), our Neural Predictor significantly outperforms Regularized Evolution in terms of sample efficiency. The mean validation accuracy is comparable to that of the oracle after about 1000 samples. The sample efficiency in validation accuracy transfers well to test accuracy in terms of both the total number of trained models in Figure 4 (middle) and wall-clock time in Figure 4 (right). After 5000 samples, Regularized Evolution reaches validation and test accuracies of 95.06% and 94.04% respectively; our predictor can reach the same validation accuracy 12.40× faster and the same test accuracy 22.83× faster. Another advantage we observe is that Neural Predictor has small search variance.

N vs K and ablation study. We next consider the problem of choosing an optimal value of N when the total number of models we're permitted to train, $N + K$, is fixed. Figure 7 summarizes our study on N. A Neural Predictor underperforms with a very small N (43 or 86), as it cannot predict accurately enough which models are interesting to evaluate. Finally, we consider the case where N is large (e.g., $N = 860$) but K is small. In this case we clearly see that the increase in quality of the GCN cannot compensate for the decrease in evaluation budget. Note that in Figure 3 we have shown that some models are

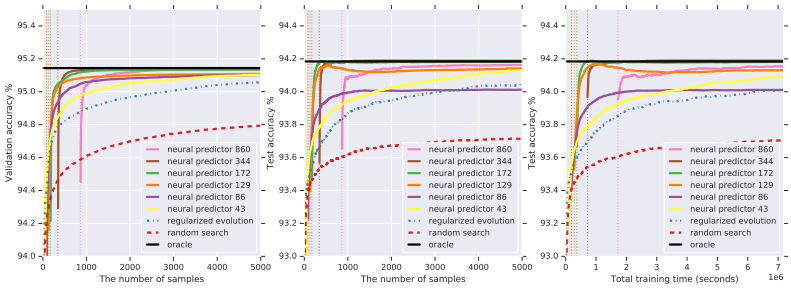

Fig. 7: Analysis of the trade-off between N training samples vs K final validation samples in the neural predictor. The x-axis is the total compute budget $N + K$. The vertical lines indicate different choices for N – the number of training samples and the point where we start validating K models. All experiments are averaged over 600 runs.

higher ranked according to validation accuracies than test accuracies. This can cause the test accuracy to degrade as we increase K in Figure 7.

4.2 ImageNet Experiments

While the NASBench-101 dataset allows us to look at the behavior in a well controlled environment, it does not allow us to evaluate whether the approach generalizes to larger scale problems. It also does not address the issue of finding high-quality inference time constrained models. Therefore, to demonstrate that our approach is more widely applicable we look at this use case in our second set of experiments on ImageNet [8] with the ProxylessNAS search space [5]. We will compare our results to a random baseline and our own reproduction of the ProxylessNAS search. In this search space, the goal is to find a good model that has an inference time between 75ms and 85ms on a Pixel-1 phone.

Search space. The ProxylessNAS search space does not have the cell-based structure from NASBench-101; it instead requires independent choices for the individual layers. There is a visualization of the search space in the supplementary material. The layers are divided up into blocks, each of which has its own fixed resolution and a fixed number of output filters. We search over which layers to skip and what operations to use in each layer. (The first layer of a block is always present.) There are approximately 6.64×10^{17} models in the search space.

Baselines. Because this search space is so large, we cannot generate the oracle baseline; we must instead rely on the random search and ProxylessNAS re-implementations we discuss next.

The random search baseline samples 256 models with inference times between 75ms and 85ms. All these models are trained for 90 epochs. We then look at which models are Pareto optimal (i.e. have good trade-offs between inference time and validation quality). All Pareto optimal models were then trained for 360 epochs to be evaluated on the test set. The results are shown in Figure 8. Implementation details are in the supplementary.

ProxylessNAS [5] is an efficient NAS algorithm based on weight sharing and RL. It trains a large neural network where different paths can be switched on or off to mimic specific architectures in the search space. Our baselines were obtained using reproductions of their search algorithm and RL reward function, as well as one of their best searched models. These reproductions come from the TuNAS [3] codebase. Our reproduction of their model (with slightly improved training hyper-parameters) achieves 74.9% accuracy, compared with 74.6% in the original paper. Across 5 independent runs, the *search algorithm* finds models with an average accuracy of 75.0% with a variance of 0.1%; latencies are comparable to that of the ProxylessNAS model. These results are near state-of-the-art for mobile CPUs. The RL controllers for ProxylessNAS and TuNAS assume that for a single network all decisions can be made independently (i.e. the probability distribution over architectures is factorized). To train the shared weights of the large model, TuNAS repeatedly (i) samples an architecture from the RL controller and (ii) trains it for a single step. To update the RL controller, another batch is sampled. This time the batch is evaluated on the validation set, and this result is used in combination with additional information (i.e. the latency) to compute a reward used to update the RL controller. Since these results are close together we consider this sufficiently good as a basis for comparison.

Neural predictor. Overall, we use the same basic pipeline as in the NASBench-101 experiments. However, because the models in the ProxylessNAS search space are much more stable than those in NASBench-101, we only need a single stage predictor. To transfer from NASBench-101 to the ProxylessNAS search space, all we have to do is to modify the graph and its node representations. The ProxylessNAS search space is just a linear graph, and the node representation at the input is nothing more than a one hot vector with length 7. This allows us to describe all architectures.

Training and validating the neural predictor. To build the neural predictor we randomly sample 119 models; 79 samples are used for training and 40

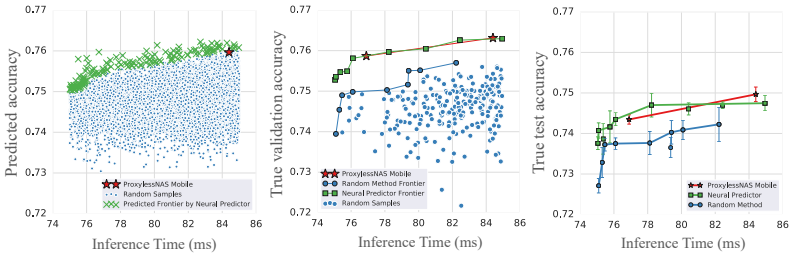

Fig. 8: Comparison between random search, ProxylessNAS and Neural Predictor. **Left:** Frontier models predicted by the Neural Predictor. **Middle:** Validation accuracy of each found frontier models. **Right:** Test accuracy of each found frontier model. (Each test accuracy is averaged over 5 training runs under different initial weight values. Error bars with 95% confidence interval are also plotted in the figure.)

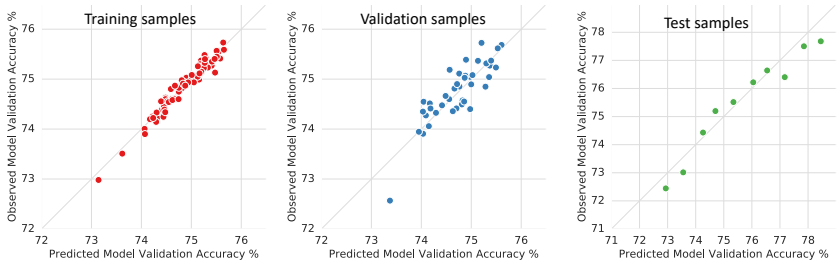

Fig. 9: The performance of Neural Predictor on training, validation and test samples.

samples are for validation. To find the GCN's hyperparameters we average validation MSE scores over 10 random training and validation splits. Based on this we select a GCN with 18 Graph Convolutional layers with node representation size 96, and with two fully-connected layers with hidden sizes 512 and 128 on top of the mean node representations in the last Graph Convolutional layer. After all hyper-parameters are finalized, we train our GCN with all 119 samples.

Our validation also showed that for ImageNet experiments, no classifier is needed to filter inaccurate models. This is because model accuracies lie within a relatively small range as as shown in Figure 9 (left and middle). Our final settings for the Neural Predictor achieved on MSE 0.109 ± 0.028 averaged over 10 validation runs. Figure 9 shows an example of the correlation between true accuracy and predicted accuracy for training samples (left) and validation samples (middle). For validation samples, the Kendall rank correlation coefficient is 0.649 and the R^2 score is 0.648895.

Looking at the predictive performance of the predictor. In Figure 9 (right), we test the generalization of our Neural Predictor to unseen test architectures. We first randomly sample 100,000 models from the ProxylessNAS search space without inference time constraint and predict their accuracies. We then pick the model with minimum predicted accuracy (72.94%), the model with maximum predicted accuracy (78.45%), and 8 additional models which are evenly spaced between those two endpoints. We train those 10 models to obtain their true accuracies. In Figure 9 (right), the Kendall rank correlation coefficient is 0.956 and the R^2 score is 0.929. More interesting, although our training dataset never observed models with accuracies higher than 76%, our Neural Predictor can still successfully predict the 5 models with accuracies higher than 76%. This demonstrates the generalization of our Neural Predictor to unseen data.

Finding high quality mobile sized models. We now use the predictor to select frontier models with good trade-offs between accuracies and inference times. We randomly sample $N = 112{,}000$ models with inference times between 75ms and 85ms, and predict their accuracies as shown in Figure 8 (left). As a sanity check, we also predict the quality of the ProxylessNAS model. The predicted validation accuracy is 76.0% and close to its true accuracy 76.3%.

The next step is selecting K Pareto optimal models. However, because the predictor can make mistakes, we need a soft version of Pareto optimality. To

do so, we sort the models based on increasing inference time. In the regular definition, a model is Pareto optimal if no faster model has higher quality. In our setup, we define a model as "soft-Pareto optimal" when the predicted accuracy is higher than the minimum of the previous J models. In our experiments we set $J = 6$. This leaves us with 137 promising models in green in Figure 8 (left). All $K = 137$ models are then trained and validated. This allows us to obtain a traditional Pareto frontier as shown in Figure 8 (middle). The architectures of those true frontier models are included in the supplementary material.

The Neural Predictor outperforms the random baseline and is comparable to ProxylessNAS. Recall that the random baseline trained 256 models. This is the same number of models we trained in total for our method ($N = 119$ models for training the predictor and $K = 137$ models selected for final validation). For test accuracy comparisons, we train the frontier models in Figure 8 (middle) for 360 epochs. The results are shown in Figure 8 (right). Now we observe that the gap between the Pareto frontier of the neural predictor and the random baseline is stable on unseen data. Note that, unlike Neural Predictor which obtained frontier models in a *single* search, to obtain a frontier for ProxylessNAS, one should run *multiple* searches to reduce/increase the inference time. We opt to reduce filters in each layer to $0.92\times$ for a faster model. The results show that the Neural Predictor and ProxylessNAS perform comparably.

The resource cost of the Neural Predictor vs. ProxylessNAS. Directly comparing the resource cost of the Neural Predictor versus ProxylessNAS is difficult. Training all $N + K = 256$ models for the entire Neural Predictor experiment took 47.5 times as much compute as a single ProxylessNAS search. However, in practice the gap is actually much smaller because optimizing the hyper-parameters of the Neural Predictor has negligible cost. Trying out a new hyper-parameter configuration for ProxylessNAS requires a full search. In our experiments we needed to run 7 searches to fine-tune the hyper-parameters for ProxylessNAS when we made a modification to the search space. This makes the Neural Predictor at most 7 times as expensive as ProxylessNAS. On top of that, the Neural Predictor is more effective at targeting different latency targets than ProxylessNAS, which needs a search per target. This could reduce the gap even more. Finally, in the ideal case we can parallelize model training for the Neural Predictor ($N = 119$ models for training in parallel followed by $K = 136$ for validation in parallel). This would finish in half the time of a ProxylessNAS run. Based on the analysis above, we believe that the Neural Predictor and ProxylessNAS are complementary. The method of choice will depend on the effort on tuning hyper-parameters, the complexity of implementing the search space (with weight sharing) and the available resources. The biggest advantage of our approach, and the most remarkable result to us, is how effective the Neural Predictor is given the simplicity of the method.

Acknowledgements. We would like to thank Chris Ying, Ken Caluwaerts, Esteban Real, Jon Shlens and Quoc Le for valuable input and discussions.

References

1. Baker, B., Gupta, O., Raskar, R., Naik, N.: Accelerating neural architecture search using performance prediction. arXiv preprint arXiv:1705.10823 (2017)
2. Bender, G., Kindermans, P.J., Zoph, B., Vasudevan, V., Le, Q.: Understanding and simplifying one-shot architecture search. In: International Conference on Machine Learning. pp. 549–558 (2018)
3. Bender, G., Liu, H., Chen, B., Chu, G., Cheng, S., Kindermans, P.J., Le, Q.V.: Can weight sharing outperform random architecture search? an investigation with tunas. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 14323–14332 (2020)
4. Brock, A., Lim, T., Ritchie, J.M., Weston, N.: Smash: one-shot model architecture search through hypernetworks. arXiv preprint arXiv:1708.05344 (2017)
5. Cai, H., Zhu, L., Han, S.: Proxylessnas: Direct neural architecture search on target task and hardware. arXiv preprint arXiv:1812.00332 (2018)
6. Dai, X., Zhang, P., Wu, B., Yin, H., Sun, F., Wang, Y., Dukhan, M., Hu, Y., Wu, Y., Jia, Y., et al.: Chamnet: Towards efficient network design through platform-aware model adaptation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 11398–11407 (2019)
7. Deng, B., Yan, J., Lin, D.: Peephole: Predicting network performance before training. arXiv preprint arXiv:1712.03351 (2017)
8. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
9. Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., Xing, E.P.: Neural architecture search with bayesian optimisation and optimal transport. In: Advances in neural information processing systems. pp. 2016–2025 (2018)
10. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
11. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images. Tech. rep., Citeseer (2009)
12. Li, L., Talwalkar, A.: Random search and reproducibility for neural architecture search. arXiv preprint arXiv:1902.07638 (2019)
13. Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.J., Fei-Fei, L., Yuille, A., Huang, J., Murphy, K.: Progressive neural architecture search. In: Proceedings of the European Conference on Computer Vision (ECCV). pp. 19–34 (2018)
14. Liu, H., Simonyan, K., Yang, Y.: Darts: Differentiable architecture search. arXiv preprint arXiv:1806.09055 (2018)
15. Luo, R., Tian, F., Qin, T., Chen, E., Liu, T.Y.: Neural architecture optimization. In: Advances in neural information processing systems. pp. 7816–7827 (2018)
16. Pham, H., Guan, M.Y., Zoph, B., Le, Q.V., Dean, J.: Efficient neural architecture search via parameter sharing. arXiv preprint arXiv:1802.03268 (2018)
17. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 4780–4789 (2019)
18. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70. pp. 2902–2911. JMLR. org (2017)

19. Tang, Y., Wang, Y., Xu, Y., Chen, H., Shi, B., Xu, C., Xu, C., Tian, Q., Xu, C.: A semi-supervised assessor of neural architectures. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (June 2020)
20. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
21. Ying, C., Klein, A., Real, E., Christiansen, E., Murphy, K., Hutter, F.: Nas-bench-101: Towards reproducible neural architecture search. arXiv preprint arXiv:1902.09635 (2019)
22. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
23. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 8697–8710 (2018)