EFFICIENT MODEL PERFORMANCE ESTIMATION VIA FEATURE HISTORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

An essential step in the task of model selection, such as hyper-parameter optimization (HPO) or neural architecture search (NAS), is the process of estimating a candidate model's (hyper-parameter or architecture) performance. Due to the high computational cost of training models until full convergence, it is necessary to develop efficient methods that can accurately estimate a model's best performance using only a small time budget. To this end, we propose a novel performance estimation method which uses a history of model features observed during the early stages of training to obtain an estimate of final performance. Our method is versatile. It can be combined with different search algorithms and applied to various configuration spaces in HPO and NAS. Using a sampling-based search algorithm and parallel computing, our method can find an architecture which is better than DARTS and with an 80% reduction in search time.

1 INTRODUCTION

Identifying the optimal hyperparameters or best architecture is important for maximizing the performance of neural networks. Accordingly, algorithms for hyperparameter optimization (HPO) and neural architecture search (NAS) have been proposed to automatically select the optimal hyperparameters and architectures in a data-driven manner. Existing HPO or NAS methods typically require that many possible configurations of hyperparameters or architectures are evaluated. However, such evaluation is extremely expensive as fully training one model until convergence may take several GPU days when the dataset is large. This calls for efficient methods that can accurately predict model performance with a small time budget.

We propose to leverage *feature histories*, that is, a sequence of features that describes the evolution of activations of a particular layer of a neural network during training, to predict what the output of the network might be at convergence. Our proposed approach is motivated by this key observation as illustrated in Figure 1: While the overall validation accuracy of a network keeps improving, the correctness of one particular image can be fluctuating during training. For one image, even it is eventually classified correctly, it can move back and forth across the decision boundary many times during training. Therefore, the validation accuracy at the early stage of training is an inaccurate estimate of the final performance of the model. However, we also observe that, if one image lies on the correct side of the optimal decision boundaries in most epochs, it will highly likely to be correctly classified when the model is fully trained. This observation motivates our proposed performance estimation strategy: (1) save the feature histories during network optimization, (2) find the optimal linear classifiers for each epoch, and, (3) evaluate the ensemble of these classifiers to approximate the final performance of the network. This method allows us to quickly reach a high accuracy close to the fully trained model at the early training stage, without waiting for the model to converge. Our empirical results show that this applies to different architecture families, including VGG (Simonyan & Zisserman, 2014), ResNet (He et al., 2016), MobileNet (Sandler et al., 2018), and also different search spaces, including DARTS (Liu et al., 2018) and NASBench-201 (Dong & Yang, 2020).

Our proposed performance estimation strategy has these advantages:

- **Accuracy**: Our method can accurately predict the final performance of given configurations. Additionally, the relative ranking of configurations is well preserved.

- **Efficiency**: The estimation strategy can accurately predict the final performance at the early stage of training, removing the need to fully train the model.
- **Versatility**: Our method does not require any pre-training or external data, but only makes use of the saved features of the network. It can be incorporated into several search algorithms for general purpose, and is applicable to a wide range of tasks in HPO and NAS.
- **Simplicity**: The implementation of our method only requires a few more lines of code in the original training loop.

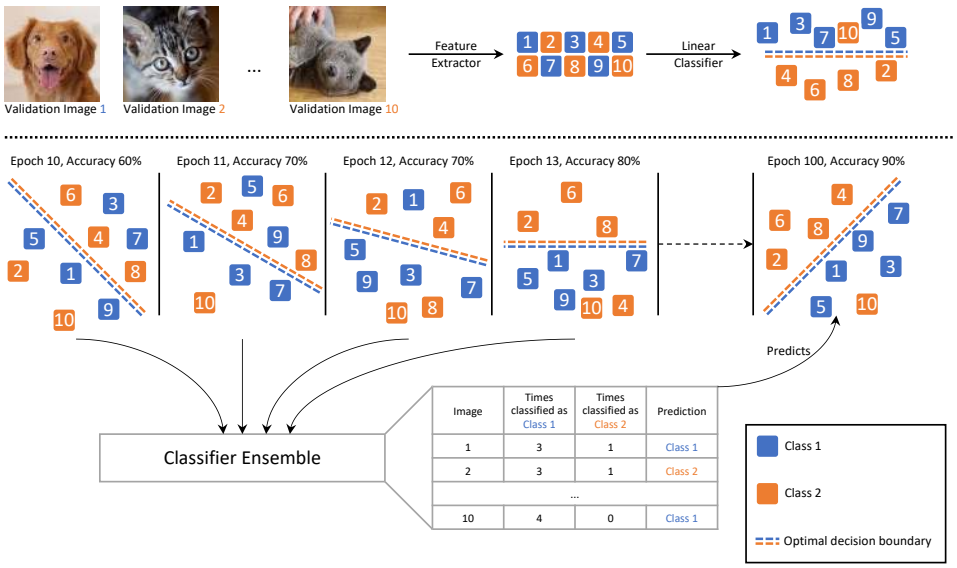


Figure 1: An illustration of our proposed feature-based performance estimation strategy. Top: The CNN architecture can be considered as composition of a feature extractor and a linear classifier. Bottom: Most image features keep moving back and forth the optimal linear boundaries during network optimization. For example, image 4 is misclassified at the latest epoch 13, but if we use the feature histories from the past 4 epochs, we can infer that this image might be correctly classified when the model converges. The classifier ensemble can identify the images which will eventually be correctly classified, and predict the final performance very early.

2 Related Work

Performance estimation: To accelerate hyperparameter optimization, Domhan et al. (2015) proposes to extrapolate the learning curves based on early training stage and terminate bad configurations accordingly. Klein et al. (2016) improves the estimation of the learning curve with a Bayesian neural network. Baker et al. (2017) applies this strategy to NAS, using architectural hyperparameters for the learning curve prediction. Different from these approaches, our method use the feature histories during network optimization, instead of performance metrics, to predict the final performance. Our method does not require training another prediction network, or manually designed learning curve modeling. We also notice a well-designed learning rate schedule can produce high-performance models within limited budget (Li et al., 2019), which can be used for estimating achievable performance. However, this type of learning rate schedule is unsustainable in the case where we need to dynamically increase the budgets for more promising configurations, which is common in hyperparameter optimization algorithms. Other methods specifically designed for accelerating performance estimation in NAS includes inheriting network weights through network morphisms (Chen et al., 2015), and weight sharing across architectures (Bender et al., 2018; Pham et al., 2018).

Hyperparameter optimization: Automated hyperparameter optimization can improve the performance of deep learning models while reducing human efforts in various applications. Model-free methods including random search and grid search, can be considered as the most basic HPO ap-

proaches. Bayesian optimization, an effective optimization method for computational costly functions, has several variants (Bergstra et al., 2011; Snoek et al., 2012; 2015) in machine learning applications. More recently, Li et al. (2017) proposes a bandit-based strategy named HyperBand, which dynamically allocates budgets for configurations based on the evaluation results. BOHB (Falkner et al., 2018) combines Bayesian optimization and HyperBand to achieve the best of both worlds. Our experiment results are mostly based on HyperBand and BOHB.

Neural architecture search: Automated architecture design has recently discovered architectures outperforming manually designed CNNs in computer vision. The search methods include reinforcement learning (Zoph & Le, 2016; Zoph et al., 2018; Tan et al., 2019), evolutionary algortihm (Real et al., 2017; Xie & Yuille, 2017; Real et al., 2019), etc. Gradient-based NAS methods (Liu et al., 2018; Chen et al., 2019; Xu et al., 2019) greatly reduce the search cost by relaxing the search space and applying gradient descent. For better evaluation and comparison of different NAS algorithms, some benchmarks (Ying et al., 2019; Klein & Hutter, 2019; Dong & Yang, 2020) are proposed. For a comprehensive overview of NAS research, one may refer to Elsken et al. (2018).

3 METHOD

We first describe our proposed method for efficient performance estimation in Section 3.1, and then explain how to combine our method with other search algorithms for neural architecture search (NAS) and hyperparameter optimization (HPO) applications in Section 3.2.

3.1 CLASSIFIER ENSEMBLE VIA FEATURE HISTORIES

A modern CNN designed for image classification (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014; He et al., 2016; Xie et al., 2017) can be divided into two parts: the feature extractor and the linear classifier. The feature extractor part typically consists of multiple convolutional layers, normalization layers, and pooling layers. The learned feature mapping can often be transferred to other datasets or tasks including object detection (Ren et al., 2015) and segmentation (He et al., 2017). In contrast, the linear classifier is one single fully connected layer, which outputs logits for each image class and learns the optimal linear boundaries with respect to the loss function. Compared with the linear classifier, the feature extractor has more parameters, is more computationally expensive and harder to optimize. Formally, we can consider a CNN as the composition of two functions: $f(\boldsymbol{x}; \boldsymbol{w}^{fea}, \boldsymbol{w}^{cls}) = h(g(\boldsymbol{x}; \boldsymbol{w}^{fea}); \boldsymbol{w}^{cls})$, where $f : \mathbb{X} \to \mathbb{R}^c$ is a CNN which maps the input image space $\mathbb{X}$ to c class logits, $g : \mathbb{X} \to \mathbb{R}^d$ is the feature extractor with parameters $\boldsymbol{w}^{fea}$ which maps images to a d-dim deep feature space, and $h : \mathbb{R}^d \to \mathbb{R}^c$ is the linear classifier with parameters $\boldsymbol{w}^{cls}$ which maps deep features to c class logits.

The task of the feature extractor is to learn a mapping from images to deep features such that the features corresponding to different image classes are linearly separable. Given a fixed feature extractor mapping, it is easy to find the optimal linear boundaries in the feature space. During the process of CNN optimization, we observe that features corresponding to most images keep moving back and forth between both sides of the optimal linear boundaries according to the current feature mapping, but in general more and more features acquire correct positions. Especially, if the feature of one image lies on the correct side of the optimal linear boundaries most of the time, with high probability it will be correctly classified at the end of optimization.

Therefore, we can use the classifier ensemble based on the history information of deep features to approximate the final performance of a CNN at an early training stage. Given saved features at history epochs, we can obtain the optimal linear classifiers for each checkpoint. Then we can test the validation images with these classifiers, and decide if their features are correctly positioned most of the time during training. Thus, we can approximate the set of images which will be correctly classified at the end of training, and give a more accurate estimation of the final performance. An illustration of our proposed feature-based classifier ensemble for performance estimation is shown in Figure 1. More experimental results about this observation are shown in Section 4.1.

The implementation is straight-forward and simple, as summarized in Algorithm 1. In addition to a typical CNN training loop, we only need to explicitly save intermediate features (line 3, 5, 6, 10, 11), optimize the linear classifiers based on saved features (line 13, 14, 15), and ensemble them (line

16, 17) to acquire a performance estimation closer to the final performance that the network can achieve. More details are discussed in Appendix A.1.

Algorithm 1: Training loop with our performance estimation

Input: Network with initial weights $f(\boldsymbol{x}; \boldsymbol{w}_0^{fea}, \boldsymbol{w}_0^{cls})$, training dataset $(\boldsymbol{X}^{train}, \boldsymbol{Y}^{train})$, validation dataset $(\boldsymbol{X}^{val}, \boldsymbol{Y}^{val})$, loss function $\text{Loss}(\boldsymbol{z}, \boldsymbol{y})$, evaluation function $\text{Eval}(\boldsymbol{z}, \boldsymbol{y})$, total epochs N, window size K

Output: Improved network performance estimation $\bar{E}_N$

for $i = 1$ *to* N **do**
  Set $\boldsymbol{w}_i^{fea}, \boldsymbol{w}_i^{cls} = \boldsymbol{w}_{i-1}^{fea}, \boldsymbol{w}_{i-1}^{cls}$
  Initialize $\boldsymbol{H}_i^{train} = \emptyset$
  for *Sampled batch* $\boldsymbol{X}_{i,j}^{train}, \boldsymbol{Y}_{i,j}^{train}$ *in* $\boldsymbol{X}^{train}, \boldsymbol{Y}^{train}$ **do**
    `// Save features for training images`
    Compute intermediate features $\boldsymbol{H}_{i,j}^{train} = g(\boldsymbol{X}_{i,j}^{train}; \boldsymbol{w}_i^{fea})$
    Save $\boldsymbol{H}_{i,j}^{train}$ into $\boldsymbol{H}_i^{train}$
    Compute outputs $\boldsymbol{Z}_{i,j}^{train} = h(\boldsymbol{H}_{i,j}^{train}; \boldsymbol{w}_i^{cls})$
    Compute loss $L_{i,j} = \text{Loss}(\boldsymbol{Z}_{i,j}^{train}, \boldsymbol{Y}_{i,j}^{train})$
    Update $\boldsymbol{w}_i^{fea}, \boldsymbol{w}_i^{cls}$ by optimizing $L_{i,j}$
  `// Save features for validation images`
  Compute intermediate features $\boldsymbol{H}_i^{val} = g(\boldsymbol{X}^{val}; \boldsymbol{w}_i^{fea})$
  Save features $\boldsymbol{H}_i^{train}, \boldsymbol{H}_i^{val}$
  Save checkpoint $\boldsymbol{w}_i^{fea}, \boldsymbol{w}_i^{cls}$
`// Optimize linear classifiers based on saved features`
for $k = N - K + 1$ *to* N **do**
  Initialize $\boldsymbol{v}_k^{cls} = \boldsymbol{w}_k^{cls}$
  Optimize $\boldsymbol{v}_k^{cls}$ with $\text{Loss}(h(\boldsymbol{H}_k^{train}; \boldsymbol{v}_k^{cls}), \boldsymbol{Y}^{train})$
  Compute outputs $\boldsymbol{Z}_k^{val} = h(\boldsymbol{H}_k^{val}; \boldsymbol{v}_k^{cls})$
`// Ensemble classifiers and evaluate`
Ensemble $\boldsymbol{Z}_{N-K+1}^{val}, \ldots, \boldsymbol{Z}_N^{val}$ to get $\bar{\boldsymbol{Z}}_N^{val}$
Evaluate $\bar{E}_N = \text{Eval}(\bar{\boldsymbol{Z}}_N^{val}, \boldsymbol{Y}^{val})$
return $\bar{E}_N$

One concern may be the computational overhead of this performance estimation strategy. In fact, compared with the typical training loop, our method usually only requires less than one half epoch time of training the original network, which is almost negligible since we often need to train the network for multiple epochs. The computational overhead of our method has two main parts: 1) Saving the intermediate features. Note the computation of deep features is a part of the original training process, we only need to slightly modify the network implementation, let it return the intermediate results, and save them in the storage after each epoch. 2) Optimization and ensemble of the linear classifiers. As discussed above, the linear classifiers are computationally cheaper and easier to optimize than the feature extraction part. We also have good weight initialization from the saved checkpoints, so a small number of iterations are adequate for convergence. Therefore, the overhead only takes up a small fraction of the whole training process.

3.2 Combination with Search Algorithms

Our method is able to give accurate performance estimation at an early training stage, thus better predict the quality of a configuration with limited computational resources. In order to apply our method in a search task like NAS and HPO, we need to integrate our performance estimation into existing search algorithms for general purpose. In the experiments, we mainly focus on three algorithms: random search, HyperBand (Li et al., 2017), and BOHB (Falkner et al., 2018).

In random search, we generate a pool of configurations by random sampling from the search space, evaluate each configuration after training for some fixed budget, and pick the best one based on the latest performance. HyperBand is a bandit strategy which adaptively allocates training resources for configurations in the sample pool based on their current performance. BOHB can be considered as a variant of HyperBand, which includes a Bayesian optimization component to better generate

the initial configuration pool. These search algorithms do not have assumptions about the search space, making them applicable to most NAS and HPO tasks. When combined with our method, these search algorithm make decisions based on the improved performance estimation instead of the latest performance acquired. In HyperBand and BOHB, this performance metric also determines the configurations which will be allocated with more budgets.

4 EXPERIMENTS

We first show that our method can efficiently estimate the performance of various CNN architecture families. Then we combine our method with search algorithms and demonstrate its effectiveness on neural architecture search (NAS) and hyperparameter optimization (HPO).

4.1 PERFORMANCE ESTIMATION FOR CNNS

We consider the following architecture families: VGG (Simonyan & Zisserman, 2014), ResNet (He et al., 2016), and MobileNetV2 (Sandler et al., 2018) and use the CIFAR-10 and CIFAR-100 dataset (Krizhevsky et al., 2009). We train each architecture for 200 epochs (see experimental details in Appendix A.2). We report the test accuracy every 10 epochs, which is the typical performance estimate at that epoch. We also apply our performance estimation for checkpoints during training every 10 epochs, which gives an improved estimation of the final performance. The results for comparison are shown in Figure 2. By utilizing the history information during network optimization, our method produces stable and accurate estimation of the final performance at an early training stage.

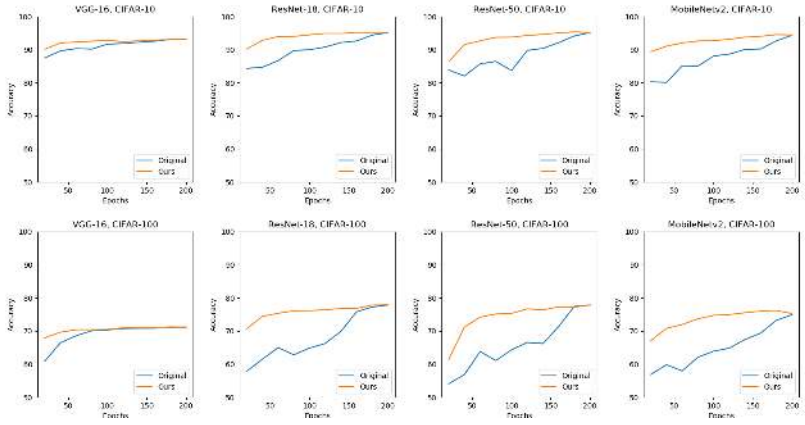


Figure 2: Performance estimation for different architectures including VGG-16, ResNet-18, ResNet-50, and MobileNetV2 on CIFAR-10/CIFAR-100 dataset. **Original** denotes the original test accuracy at different epochs. **Ours** denotes the performance estimated by our method using the checkpoints up to the given epochs. Our method can reach an accuracy closer to the final accuracy at the early stage of training, which indicates a much more accurate performance estimate than the baseline.

In a resource-limited search setting, one may need to shrink the training budget and adjust the learning rate schedule accordingly, at the cost of reducing the final performance. We also investigate the example of ResNet-18 with fewer training epochs, shown in Figure 3. Our method can still accurately predict the final performance early in this setting.

We can take a closer look at the ResNet-18/CIFAR-100 example to validate our observation stated in Section 3.1. We visualize whether an image is classified correctly at each epoch during training in Figure 4. As we can observe, most test image features keep moving between sides of the optimal linear boundaries, but in general the number of features with correct positions is increasing during training. If the feature of one image lies on the correct side of the optimal linear boundaries most of the time, the image will be correctly classified with higher probability after optimization. When we use the ensemble of the optimal classifiers, we can accurately approximate the set of images which will eventually be correctly or incorrectly classified, at an earlier training stage like 80 epochs.

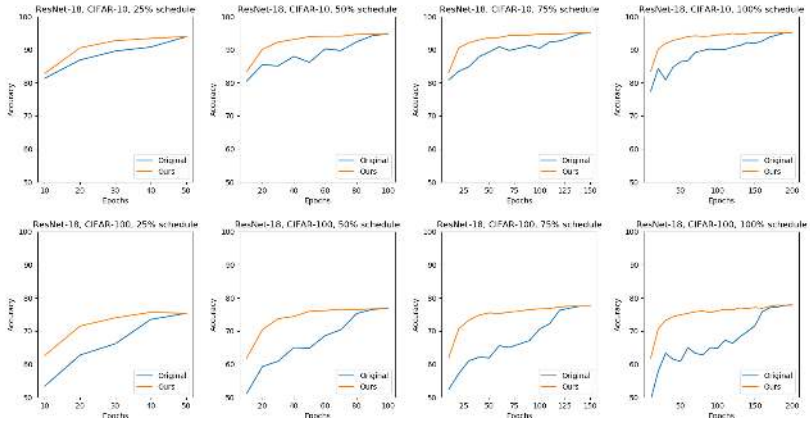
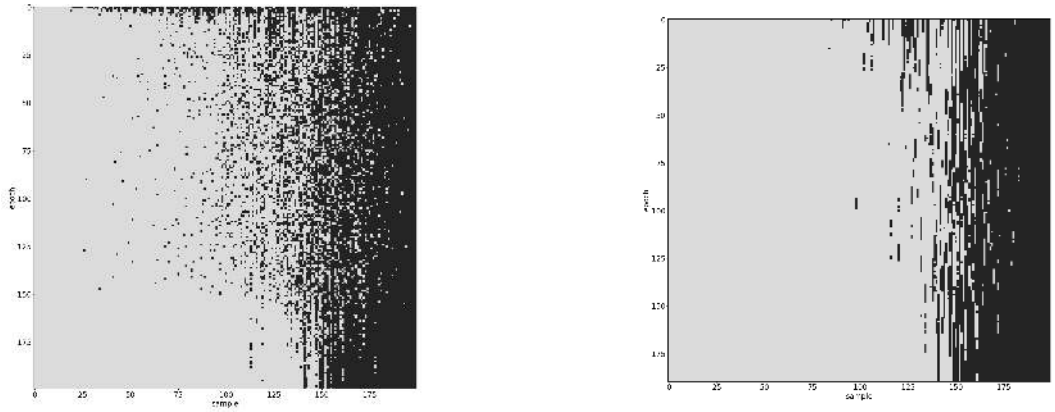


Figure 3: Performance estimation for different training budgets from 25% (50 epochs) to 100% (200 epochs). The linear learning rate schedule is adjusted accordingly.



(a) Normal training

(b) Ensemble of feature histories

Figure 4: Classification results of test images during training ResNet-18 on CIFAR-100. We sample 200 test images to observe if their deep features can be correctly classified by (a) the optimal linear classifiers, and (b) the classifier ensembles at each epoch. Light gray pixels denotes that image is correctly classified, and black pixels denotes a wrong classification. The horizontal axis indicates the image indices, sorted by their accuracy for better visualization. The vertical axis indicates the training epochs up to 200 at the bottom.

4.2 Neural Architecture Search

For NAS applications, we conduct experiments in two search spaces: NAS-Bench-201 (Dong & Yang, 2020) and DARTS (Liu et al., 2018).

4.2.1 NAS-Bench-201

NAS-Bench-201 (Dong & Yang, 2020) is a public benchmark for testing NAS algorithms. It defines a search space consisting of $15,625$ architectures, and includes full training log of all the architectures on CIFAR-10, CIFAR-100 (Krizhevsky et al., 2009), and downsampled ImageNet (Deng et al., 2009). In the following experiments, we mainly use the information about the "true performance" on CIFAR-10, which is defined as the average top-1 test accuracy of three independent runs of training the given architecture for 200 epochs.

It is important for a NAS algorithm to obtain the relative ranking of architectures that is consistent with the true performance, so that the algorithm can return an architecture among the best ones in

the search space. Usually we need to train the architectures for some epochs and evaluate on the validation set to acquire a performance estimation, and the longer we train, the relative ranking is closer to the true performance. Given the true performance and estimated performance, we can measure the consistency of their relative ranking with the Kendall's τ rank correlation coefficient (Kendall, 1938). The τ coefficient ranges in $[-1, 1]$, and when the two observations have perfectly matched relative rankings, τ reaches its maximum value 1.

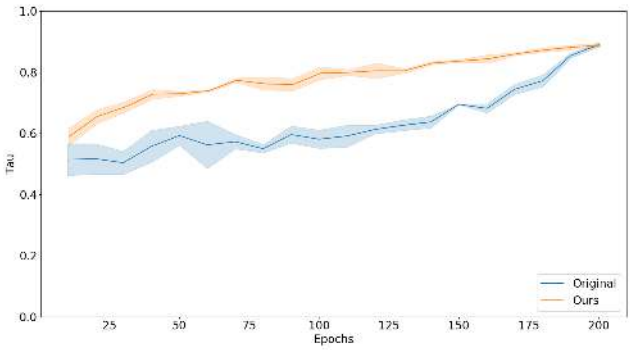
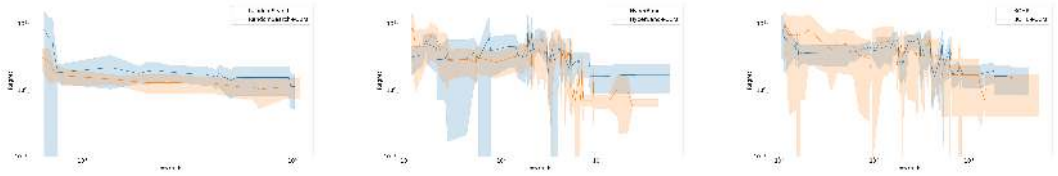


Figure 5: Relative ranking measured by Kendall's τ coefficient and the true performance. **Original** denotes the original accuracy metrics at different epochs. **Ours** denotes the predicted performance by our method using the checkpoints up to the given epochs. For fair comparison, we slightly shift **Ours** rightwards because of the computational overhead.

We first verify that our performance estimation can produce the relative ranking which is more consistent with the true performance, when each architecture is trained for some fixed budget. We randomly sample 100 architectures from the search space, train each for 200 epochs and evaluate the consistency of both the original performance metric and our improved performance estimation on the validation dataset every 10 epochs (see experimental details in Appendix A.3.1). The results are shown in Figure 5. It is evident that our method not only predicts the performance closer to the true performance, but also leads to better relative ranking for distinguishing different architectures. With our performance estimation strategy, we can find better architectures with limited training budgets.

We also compare several search algorithms with and without our performance estimation, as described in Section 3.2 (see experimental details in Appendix A.3.2). We quantify the performance of each search algorithm over time by regret, defined as the difference in the true performance between the best architecture determined by the search algorithm and the best architecture in the whole search space. The results are shown in Figure 6. Our method can improve all three search algorithms in terms of the final searched architectures. Especially, the combination of HyperBand and our method can robustly find an architecture which has $< 1\%$ test accuracy gap from the global optimum within 2 hours.



(a) Random search (b) HyperBand (c) BOHB

Figure 6: Regret over time of random search, HyperBand, and BOHB with and without our performance estimation for searching in NAS-Bench-201. We use log scale on both axes for better visualization.

4.2.2 DARTS

Differentiable ARchiTecture Search (DARTS) (Liu et al., 2018) is a recent NAS algorithm which relaxes the search space to be continuous, achieving search speed orders of magnitude faster than previous methods. Here we directly transfer the method that we use on NAS-Bench-201 to the DARTS search space, to demonstrate our performance estimation strategy is also helpful in another NAS setting.

Table 1: Comparison with state-of-the-art NAS methods which uses the DARTS search space on CIFAR-10.

Method	Test Error (%)	Params (M)	Search Cost (GPU days / run)	Search Method
DARTS (first order)	3.00 ± 0.14	3.3	0.4	Gradient
DARTS (second order)	2.76 ± 0.09	3.3	1	Gradient
P-DARTS	2.50	3.4	0.3	Gradient
PC-DARTS	2.57 ± 0.07	3.6	0.1	Gradient
UNAS	2.53	3.3	4.3	Gradient&RL
Ours	2.63	3.4	$0.2 \times 4^*$	HyperBand

* We use 4 parallel workers. The wall-clock time is 0.2 day.

Table 2: Search results for RandAugment hyperparameter optimization on a subset of CIFAR-10.

Architecture	Test Accuracy (%) BOHB	BOHB + Ours	HyperBand	HyperBand + Ours
VGG-16	85.35 ± 0.44	$\mathbf{85.61} \pm 0.38$	85.38 ± 0.24	$\mathbf{85.60} \pm 0.28$
ResNet-18	86.04 ± 1.16	$\mathbf{86.53} \pm 0.64$	85.83 ± 0.43	$\mathbf{86.06} \pm 0.64$
MobileNetV2	$\mathbf{86.92} \pm 0.46$	86.80 ± 0.71	86.53 ± 0.23	$\mathbf{86.62} \pm 0.48$

We use the combination of HyperBand and our performance estimation, which performs the best on NAS-Bench-201, and follow the practice of architecture evaluation in DARTS (see experimental details in Appendix A.4). The results are summarized in Table 1. We also list some more recent DARTS variants which also use the same search space and evaluation configuration, including P-DARTS (Chen et al., 2019), PC-DARTS (Xu et al., 2019), and UNAS (Vahdat et al., 2020). Our method is able to find better architectures than DARTS with similar search cost. Our result is also close to more recent state-of-the-art NAS methods. It is notable that our method is the only one that does not make use of the gradient-based method, so we can easily parallelize the search process. In fact, we use 4 parallel workers in the experiment, reducing the wall-clock search time to 0.2 day per run. We expect to further reduce search time with even more parallel workers.

4.3 Hyperparameter Optimization

We also test our method in another setting where differentiable search methods do not apply. We consider the problem of optimizing hyperparameters in data augmentation for CNN training. We use the search space defined in RandAugment (Cubuk et al., 2020), which has only two hyperparameters: N is the number of augmentation transformations to apply, and M is the magnitude for the transformations. The search space is significantly smaller than previous work such as AutoAugment (Cubuk et al., 2018), but still computationally expensive for the grid search that Cubuk et al. (2020) does for each CNN architecture and dataset.

To compare HyperBand and BOHB with and without our performance estimation, we search the optimal RandAugment configuration for VGG-16, ResNet-18, and MobileNetV2 on a subset of CIFAR-10 (see experimental details in Appendix A.5). The results are summarized in Table 2. In most cases, our method consistently improves the baseline search algorithms. This task further demonstrates the effectiveness and versatility of our method.

5 Conclusion

We propose a novel performance estimation strategy, which effectively use the saved feature histories during optimization to produce accurate estimation of the final performance. Our method is simple to implement and applicable to many tasks in NAS and HPO, and leads to improvement for general search algorithms. For future directions, we think it would be interesting to use the performance estimation to guide and accelerate CNN training.

REFERENCES

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

Gabriel M. Bender, Pieter jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554, 2011.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1294–1303, 2019.

Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 702–703, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

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

Xuanyi Dong and Yi Yang. Nas-bench-102: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

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

Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969, 2017.

Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

Aaron Klein and Frank Hutter. Tabular benchmarks for joint architecture and hyperparameter optimization. *arXiv preprint arXiv:1905.04970*, 2019.

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. 2016.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

Mengtian Li, Ersin Yumer, and Deva Ramanan. Budgeted training: Rethinking deep neural network training under resource constraints. *arXiv preprint arXiv:1905.04753*, 2019.

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

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *ICML*, 2018.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

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

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pp. 91–99, 2015.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pp. 2171–2180, 2015.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Arash Vahdat, Arun Mallya, Ming-Yu Liu, and Jan Kautz. Unas: Differentiable architecture search meets reinforcement learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11266–11275, 2020.

Lingxi Xie and Alan Yuille. Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pp. 1379–1388, 2017.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. Pc-darts: Partial channel connections for memory-efficient differentiable architecture search. *arXiv preprint arXiv:1907.05737*, 2019.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pp. 7105–7114, 2019.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

A EXPERIMENTAL DETAILS

A.1 CLASSIFIER ENSEMBLE

The training procedure of the classifier ensemble has the same following configuration across all experiments. We use the the saved features from the most recent $K = 10$ epochs. We first obtain K optimal classifiers by training them on the saved features of images from the training set and their labels. The weights are initialized from the saved network weights of the corresponding epochs. We use the cross entropy loss and SGD optimizer with momentum 0.9 and weight decay 5×10^{-4}. The learning rate starts from 0.05 and decays following a linear schedule (Li et al., 2019). The batch size is 1024, and the number of epochs is 5. To build the ensemble, we collect the classification results of the optimized linear classifiers for the validation image features, and output the mean of the softmax-ed probability distribution, which is then used for evaluation and estimation. The whole process introduces little computational cost compared with the original training loop. For example, in NAS-Bench-201, we observe the overhead of our method is typically 0.5 epoch time for each architecture.

A.2 PERFORMANCE ESTIMATION FOR CNNS

In the first part, we train each CNN on CIFAR-10 and CIFAR-100 for 200 epochs with batch size 256. The initial learning rate is set to 0.1 for ResNet and MobileNetV2, 0.01 for VGG, and we use a linearly decaying learning rate schedule as suggested by Li et al. (2019). We use SGD optimizer with momentum 0.9 and weight decay 5×10^{-4}.

In the second part, we train ResNet-18 on CIFAR-100 for $50, 100, 150, 200$ epochs, corresponding to $25\%, 50\%, 75\%, 100\%$ budgets. The learning rate still follows the linear schedule, which drops to zero at the end of each budget. Other settings are the same as the first part.

A.3 NAS-BENCH-201

A.3.1 RELATIVE RANKING

In each run of this experiment, we first randomly sample 100 architectures from NAS-Bench-201, and train each for 200 epochs with batch size 256. The initial learning rate is set to 0.1 and we use the linear learning rate schedule. We use SGD optimizer with momentum 0.9 and weight decay 5×10^{-4}. We split the original CIFAR-10 training set into two subsets: 40000 images for training and 10000 images for validation. Then we compare the relative ranking of the original performance metrics and our performance estimation strategy measured on the validation set, using the true performance from the benchmark and Kendall's τ coefficient. The experiment is repeated 3 times.

A.3.2 ARCHITECTURE SEARCH

For random search, we randomly sample 64 architectures, and train each for 128 epochs with batch size 256. For HyperBand and BOHB, the training budget for each architecture ranges in $[1, 128]$ epochs, and the factor for increasing training budget and shrinking sample pool is set as $\eta = 2$. We use 4 parallel workers, each using an NVIDIA GeForce RTX 2080 Ti GPU. The total search time is about the same for the three algorithms. During search, the initial learning rate is set to 0.1 and we use the linear learning rate schedule with the maximal budget set to 128. We use SGD optimizer with momentum 0.9 and weight decay 5×10^{-4}. we use a subset from the original CIFAR-10 training set with 8000 images for training, and 2000 images for validation. After search, we calculate the regret as the difference in the true performance between the architecture given by each algorithm and the global optimal architecture, provided by the benchmark. Each experiment is repeated 5 times with different random seeds.

A.4 DARTS

During search, the search space relaxation is not applicable in our method. We still randomly sample from the discrete search space defined in DARTS, and select the best architectures using HyperBand and our performance estimation. We decompose the process of sampling architecture into sampling

the operations and connections for each node in the cells from the uniform distribution of valid choices. At search time, we set the batch size to 96, initial number of channels to 32, number of cells to 12. We also use Cutout (DeVries & Taylor, 2017), path dropout, and auxiliary towers. The initial learning rate is 0.025 and we use the linear learning rate schedule with the maximal budget set to 128. We use SGD optimizer with momentum 0.9 and weight decay 3×10^{-4}. Other search settings are the same as Appendix A.3.2.

For architecture evaluation, we closely follow the setup of DARTS: We repeat the search process for 4 times and pick the best architecture for fully train and evaluation. An enlarged architecture with 36 initial channels is trained for 600 epochs with batch size 96, using Cutout, path dropout, and auxiliary towers. The only difference in architecture evaluation is we increase the number of cells from 20 to 22. The reason is that our searched cells have more parameter-free operations like skip connections and pooling layers compared with DARTS, so we increase the number of cells to match the number of parameters in DARTS.

A.5 RANDAUGMENT

To better show the influence of data augmentation, we only use a small subset of the original CIFAR-10 training set with 8000 images for training and 2000 images for validation. Besides RandAugment, default data augmentation for training images also includes random flips, pad-and-crop and Cutout, following RandAugment. Other search settings are the same as Appendix A.3.2. Each experiment is repeated 3 times with different random seeds.