ACCELERATING NEURAL ARCHITECTURE SEARCH USING PERFORMANCE PREDICTION

Bowen Baker[1*], **Otkrist Gupta**[1*], **Ramesh Raskar**[1], **Nikhil Naik**[2]
(* - indicates equal contribution)
[1]**MIT Media Laboratory**, Cambridge, MA 02139
[2]**Harvard University**, Cambridge, MA 02138
{bowen, otkrist, raskar, naik}@mit.edu

ABSTRACT

Methods for neural network hyperparameter optimization and meta-modeling are computationally expensive due to the need to train a large number of model configurations. In this paper, we show that standard frequentist regression models can predict the final performance of partially trained model configurations using features based on network architectures, hyperparameters, and time-series validation performance data. We empirically show that our performance prediction models are much more effective than prominent Bayesian counterparts, are simpler to implement, and are faster to train. Our models can predict final performance in both visual classification and language modeling domains, are effective for predicting performance of drastically varying model architectures, and can even generalize between model classes. Using these prediction models, we also propose an early stopping method for hyperparameter optimization and meta-modeling, which obtains a speedup of a factor up to 6x in both hyperparameter optimization and meta-modeling. Finally, we empirically show that our early stopping method can be seamlessly incorporated into both reinforcement learning-based architecture selection algorithms and bandit based search methods. Through extensive experimentation, we empirically show our performance prediction models and early stopping algorithm are state-of-the-art in terms of prediction accuracy and speedup achieved while still identifying the optimal model configurations.

1 INTRODUCTION

At present, significant human expertise and labor is required for designing high-performing neural network architectures and successfully training them for different applications. Ongoing research in two areas—meta-modeling and hyperparameter optimization—attempts to reduce the amount of human intervention required for these tasks. Hyperparameter optimization methods (e.g., Hutter et al. (2011); Snoek et al. (2015); Li et al. (2017)) focus primarily on obtaining good optimization hyperparameter configurations for training human-designed networks, whereas meta-modeling algorithms (Bergstra et al., 2013; Verbancsics & Harguess, 2013; Baker et al., 2017; Zoph & Le, 2017) aim to design neural network architectures from scratch. Both sets of algorithms require training a large number of neural network configurations for identifying the right set of hyperparameters or the right network architecture—and are hence computationally expensive.

When sampling many different model configurations, it is likely that many subpar configurations will be explored. Human experts are quite adept at recognizing and terminating suboptimal model configurations by inspecting their partial learning curves. In this paper we seek to emulate this behavior and automatically identify and terminate subpar model configurations in order to speedup both meta-modeling and hyperparameter optimization methods. Our method parameterizes learning curve trajectories with simple features derived from model architectures, training hyperparameters, and early time-series measurements from the learning curve. We use these features to train a set of frequentist regression models that predicts the final validation accuracy of partially trained neural network configurations using a small training set of fully trained curves from both image classification and language modeling domains. We use these predictions and uncertainty estimates

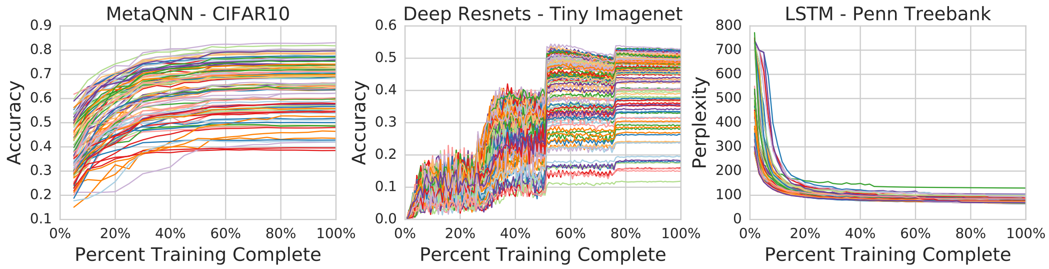

Figure 1: **Example Learning Curves:** Example learning curves from experiments considered in this paper. Note the diversity in convergence times and overall learning curve shapes.

obtained from small model ensembles to construct a simple early stopping algorithm that can speedup both meta-modeling and hyperparameter optimization methods.

While there is some prior work on neural network performance prediction using Bayesian methods (Domhan et al., 2015; Klein et al., 2017), our proposed method is significantly more accurate, accessible, and efficient. We hope that our work leads to inclusion of neural network performance prediction and early stopping in the practical neural network training pipeline.

2 Related Work

Neural Network Performance Prediction: There has been limited work on predicting neural network performance during the training process. Domhan et al. (2015) introduce a weighted probabilistic model for learning curves and utilize this model for speeding up hyperparameter search in small convolutional neural networks (CNNs) and fully-connected networks (FCNs). Building on Domhan et al. (2015), Klein et al. (2017) train Bayesian neural networks for predicting unobserved learning curves using a training set of fully and partially observed learning curves. Both methods rely on expensive Markov chain Monte Carlo (MCMC) sampling procedures and handcrafted learning curve basis functions. We also note that Swersky et al. (2014) develop a Gaussian Process kernel for predicting individual learning curves, which they use to automatically stop and restart configurations.

Meta-modeling: We define meta-modeling as an algorithmic approach for designing neural network architectures from scratch. The earliest meta-modeling approaches were based on genetic algorithms (Schaffer et al., 1992; Stanley & Miikkulainen, 2002; Verbancsics & Harguess, 2013) or Bayesian optimization (Bergstra et al., 2013; Shahriari et al., 2016). More recently, reinforcement learning methods have become popular. Baker et al. (2017) use Q-learning to design competitive CNNs for image classification. Zoph & Le (2017) use policy gradients to design state-of-the-art CNNs and Recurrent cell architectures. Several methods for architecture search (Cortes et al., 2017; Negrinho & Gordon, 2017; Zoph et al., 2017; Brock et al., 2017; Suganuma et al., 2017) have been proposed this year since the publication of Baker et al. (2017) and Zoph & Le (2017).

Hyperparameter Optimization: We define hyperparameter optimization as an algorithmic approach for finding optimal values of design-independent hyperparameters such as learning rate and batch size, along with a limited search through the network design space. Bayesian hyperparameter optimization methods include those based on sequential model-based optimization (SMAC) (Hutter et al., 2011), Gaussian processes (GP) (Snoek et al., 2012), TPE (Bergstra et al., 2013), and neural networks Snoek et al. (2015). However, random search or grid search is most commonly used in practical settings (Bergstra & Bengio, 2012). Recently, Li et al. (2017) introduced Hyperband, a multi-armed bandit-based efficient random search technique that outperforms state-of-the-art Bayesian optimization methods.

3 Neural Network Performance Prediction

We first describe our model for neural network performance prediction, followed by a description of the datasets used to evaluate our model, and finally present experimental results.

3.1 Modeling Learning Curves

Our goal is to model the validation accuracy y_T of a neural network configuration $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$ at epoch $T \in \mathbb{Z}^+$ using previous performance observations $y(t)$. For each configuration $\mathbf{x}$ trained for T epochs, we record a time-series $y(T) = y_1, y_2, \ldots, y_T$ of validation accuracies. We train a population of n configurations, obtaining a set $\mathcal{S} = \{(\mathbf{x}^1, y^1(t)), (\mathbf{x}^2, y^2(t)), \ldots, (\mathbf{x}^n, y^n(t))\}$. Note that this problem formulation is very similar to Klein et al. (2017).

We propose to use a set of features $u_{\mathbf{x}}$, derived from the neural network configuration $\mathbf{x}$, along with a subset of time-series accuracies $y(\tau) = (y_t)_{t=1,2,\ldots,\tau}$ (where $1 \leq \tau < T$) from $\mathcal{S}$ to train a regression model for estimating y_T. Our model predicts y_T of a neural network configuration using a feature set $x_f = \{u_{\mathbf{x}}, y(t)_{1\text{–}\tau}\}$. For clarity, we train $T-1$ regression models, where each successive model uses one more point of the time-series validation data. As we shall see in subsequent sections, this use of *sequential regression models* (SRM) is more computationally and more precise than methods that train a single Bayesian model.

Features: We use features based on time-series (TS) validation accuracies, architecture parameters (AP), and hyperparameters (HP). (1) TS: These include the validation accuracies $y(t)_{1\text{–}\tau} = (y_t)_{t=1,2,\ldots,\tau}$ (where $1 \leq \tau < T$), the first-order differences of validation accuracies (i.e., $y_t{}' = (y_t - y_{t-1})$), and the second-order differences of validation accuracies (i.e., $y_t{}'' = (y_t{}' - y'_{t-1})$). (2) AP: These include total number of weights and number of layers. (3) HP: These include all hyperparameters used for training the neural networks, e.g., initial learning rate and learning rate decay (full list in Appendix Table 2).

3.2 Datasets and Training Procedures

We experiment with small and very deep CNNs (e.g., ResNet, Cuda-Convnet) trained on image classification datasets and with LSTMs trained with Penn Treebank (PTB), a language modeling dataset. Figure 1 shows example learning curves from three of the datasets considered in our experiments. We provide brief summary of the datasets below. Please see Appendix Section A for further details on the search space, preprocessing, hyperparameters and training settings of all datasets.

Datasets with Varying Architectures:

Deep Resnets (TinyImageNet): We sample 500 ResNet architectures and train them on the TinyImageNet[*] dataset (containing 200 classes with 500 training images of 32×32 pixels) for 140 epochs. We vary depths, filter sizes and number of convolutional filter block outputs. The network depths vary between 14 and 110.

Deep Resnets (CIFAR-10): We sample 500 39-layer ResNet architectures from a search space similar to Zoph & Le (2017), varying kernel width, kernel height, and number of kernels. We train these models for 50 epochs on CIFAR-10.

MetaQNN CNNs (CIFAR-10 and SVHN): We sample 1,000 model architectures from the search space detailed by Baker et al. (2017), which allows for varying the numbers and orderings of convolution, pooling, and fully connected layers. The models are between 1 and 12 layers for the SVHN experiment and between 1 and 18 layers for the CIFAR-10 experiment. Each architecture is trained on SVHN and CIFAR-10 datasets for 20 epochs.

LSTM (PTB): We sample 300 LSTM models and train them on the Penn Treebank dataset for 60 epochs, evaluating perplexity on the validation set. We vary number of LSTM cells and hidden layer inputs between 10-1400.

Datasets with Varying Hyperparameters:

Cuda-Convnet (CIFAR-10 and SVHN): We train Cuda-Convnet architecture (Krizhevsky, 2012) with varying values of initial learning rate, learning rate reduction step size, weight decay for convolutional and fully connected layers, and scale and power of local response normalization layers. We train models with CIFAR-10 for 60 epochs and with SVHN for 12 epochs.

[*] https://tiny-imagenet.herokuapp.com/

Dataset	ν-SVR (RBF)	ν-SVR (Linear)	Random Forest	OLS
MetaQNN (CIFAR-10)	94.22 ± 0.25	94.44 ± 0.14	92.27 ± 0.91	93.22 ± 1.1
Resnet (TinyImageNet)	85.78 ± 1.82	91.8 ± 1.1	91.37 ± 2.18	90.15 ± 1.8
LSTM (Penn Treebank)	83.29 ± 7.71	98.59 ± 0.8	91.38 ± 1.97	89.8 ± 0.16

Table 1: **Frequentist Model Comparison:** We report the coefficient of determination R^2 for four standard methods. Each model is trained with 100 samples on 25% of the learning curve. We find that ν-SVR works best on average, though not by a large margin.

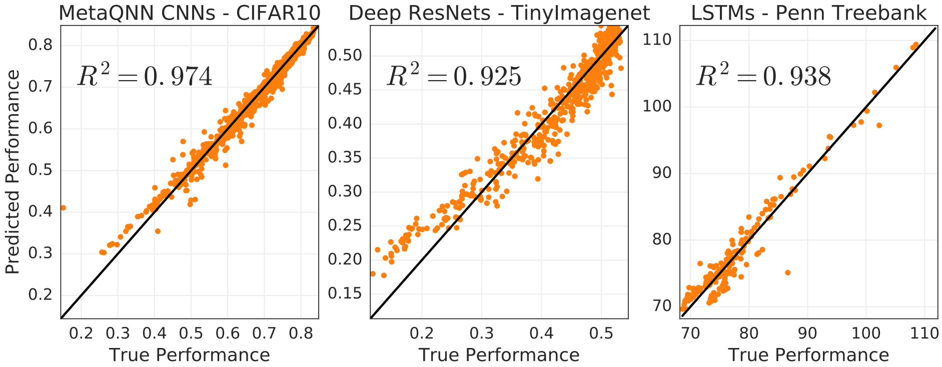

Figure 2: **Predicted vs True Values of Final Performance:** We show the shape of the predictive distribution on three experiments: MetaQNN models, Deep Resnets, and LSTMs. Each ν-SVR (RBF) model is trained with 100 configurations with data from 25% of the learning curve. We predict validation set classification accuracy for MetaQNN and Deep ResNets, and perplexity for LSTMs.

3.3 Prediction Performance

Choice of Regression Method: We now describe our results for performing final neural network performance. For all experiments, we train our SRMs on 100 randomly sampled neural network configurations. We obtain the best performing method using random hyperparameter search over 3-fold cross-validation. We then compute the regression performance over the remainder of the dataset using the coefficient of determination R^2. We repeat each experiment 10 times and report the results with standard errors. We experiment with a few different frequentist regression models, including ordinary least squares (OLS), random forests, and ν-support vector machine regressions (ν-SVR). As seen in Table 1, ν-SVR with linear or RBF kernels perform the best on most datasets, though not by a large margin. For the rest of this paper, we use ν-SVR RBF unless otherwise specified.

Ablation Study on Feature Sets: In Table 2, we compare the predictive ability of different feature sets, training SVR (RBF) with time-series (TS) features obtained from 25% of the learning curve, along with features of architecture parameters (AP), and hyperparameters (HP). TS features explain the largest fraction of the variance in all cases. For datasets with varying architectures, AP are more important that HP; and for hyperparameter search datasets, HP are more important than AP, which is expected. AP features almost match TS on the ResNet (TinyImageNet) dataset, indicating that choice of architecture has a large influence on accuracy for ResNets. Figure 2 shows the true vs. predicted performance for all test points in three datasets, trained with TS, AP, and HP features.

Generalization Between Depths: We also test to see whether SRMs can accurately predict the performance of out-of-distribution neural networks. In particular, we train SVR (RBF) with 25% of TS, along with AP and HP features on ResNets (TinyImagenet) dataset, using 100 models with number of layers less than a threshold d and test on models with number of layers greater than d, averaging over 10 runs. Value of d varies from 14 to 110. For $d = 32$, R^2 is 80.66 ± 3.8. For $d = 62$, R^2 is 84.58 ± 2.7.

Feature Set	MetaQNN (CIFAR-10)	ResNets (TinyImageNet)	LSTM (Penn Treebank)	Cuda-Convnet (CIFAR-10)
TS	93.98 ± 0.15	86.52 ± 1.85	97.81 ± 2.45	95.54 ± 0.24
AP	27.45 ± 4.25	84.33 ± 1.7	16.11 ± 1.13	1.1 ± 0.6
HP	12.60 ± 1.79	8.78 ± 1.14	3.98 ± 0.88	18.19 ± 2.19
TS+AP	84.09 ± 1.4	88.82 ± 2.95	$96.92 \pm 2.8)$	95.36 ± 0.27
AP+HP	27.01 ± 5.2	81.71 ± 3.9	15.97 ± 2.57	21.65 ± 2.72
TS+AP+HP	94.44 ± 0.14	91.8 ± 1.1	98.24 ± 2.11	95.69 ± 0.15

Table 2: **Ablation Study on Feature Sets:** Time-series features (TS) refers to the partially observed learning curves, architecture parameters (AP) refer to the number of layers and number of weights in a deep model, and hyperparameters (HP) refer to the optimization parameters such as learning rate. All results with SVR (RBF). 25% of learning curve used for TS.

3.3.1 Comparison with Existing Methods:

We now compare the neural network performance prediction ability of SRMs with three existing learning curve prediction methods: (1) Bayesian Neural Network (BNN) (Klein et al., 2017), (2) the learning curve extrapolation (LCE) method (Domhan et al., 2015), and (3) the last seen value (LastSeenValue) heuristic (Li et al., 2017). When training the BNN, we not only present it with the subset of fully observed learning curves but also all other partially observed learning curves from the training set. While we do not present the partially observed curves to the ν-SVR SRM for training, we felt this was a fair comparison as ν-SVR uses the entire partially observed learning curve during inference. Methods (2) and (3) do not incorporate prior learning curves during training. Figure 3 shows the R^2 obtained by each method for predicting the final performance versus the percent of the learning curve used for training the model. We see that in all neural network configuration spaces and across all datasets, either one or both SRMs outperform the competing methods. The LastSeenValue heuristic only becomes viable when the configurations are near convergence, and its performance is worse than an SRM for very deep models. We also find that the SRMs outperform the LCE method in all experiments, even after we remove a few extreme prediction outliers produced by LCE. Finally, while BNN outperforms the LastSeenValue and LCE methods when only a few iterations have been observed, it does worse than our proposed method. In summary, we show that our simple, frequentist SRMs outperforms existing Bayesian approaches on predicting neural network performance on modern, very deep models in computer vision and language modeling tasks.

Since most of our experiments perform stepwise learning rate decay; it is conceivable that the performance gap between SRMs and both LCE and BNN results from a lack of sharp jump in their basis functions. We experimented with exponential learning rate decay (ELRD), which the basis functions in LCE are designed for. We trained 630 random nets with ELRD, from the 1000 MetaQNN-CIFAR10 nets. Predicting from 25% of the learning curve, the R^2 is 0.95 for ν-SVR (RBF), 0.48 for LCE (with extreme outlier removal, negative without), and 0.31 for BNN. This comparison illuminates another benefit of our method: we do not require handcrafted basis functions to model new learning curve types.

Training and Inference Speed Comparison: Another advantage of our regression approach is speed. SRMs are much faster to train and do inference in than proposed Bayesian methods (Domhan et al., 2015; Klein et al., 2017). On 1 core of a Intel 6700k CPU, an ν-SVR (RBF) with 100 training points trains in 0.006 seconds, and each inference takes 0.00006 seconds. In comparison, the LCE code takes 60 seconds and BNN code takes 0.024 seconds on the same hardware for each inference.

4 Applying Performance Prediction For Early Stopping

To speed up hyperparameter optimization and meta-modeling methods, we develop an algorithm to determine whether to continue training a partially trained model configuration using our sequential regression models. If we would like to sample N total neural network configurations, we begin by sampling and training $n \ll N$ configurations to create a training set $\mathcal{S}$. We then train a model $f(x_f)$to predict y_T. Now, given the current best performance observed y_{BEST}, we would like to terminate training a new configuration $\mathbf{x}'$ given its partial learning curve $y'(t)_{1-\tau}$ if $f(x_f{}') = \hat{y}_T \leq y_{\text{BEST}}$ so as to not waste computational resources exploring a suboptimal configuration.

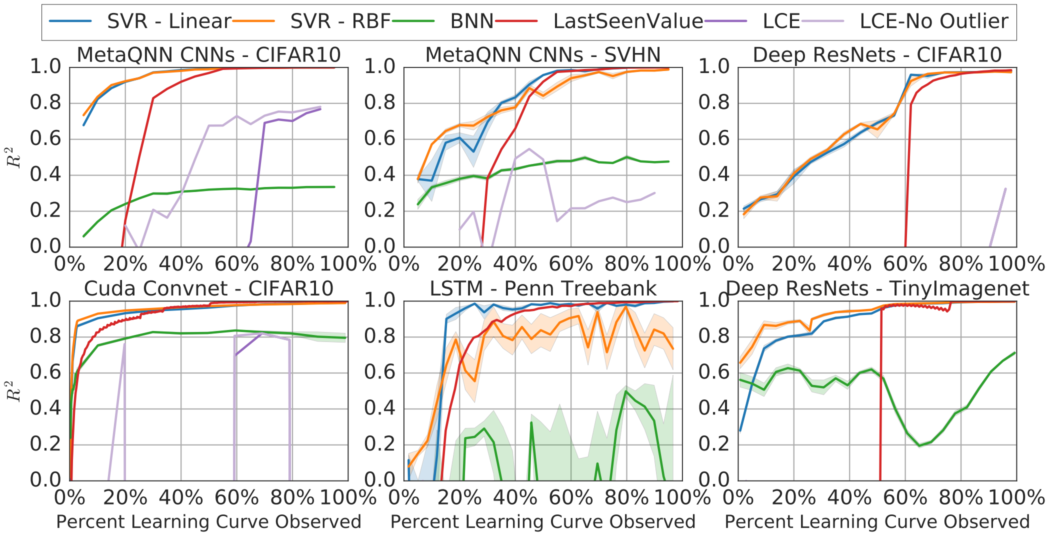

Figure 3: **Performance Prediction Results:** We plot the performance of each method versus the percent of learning curve observed. For BNN and ν-SVR (linear and RBF), we sample 10 different training sets, plot the mean R^2, and shade the corresponding standard error. We compare our method against BNN (Klein et al., 2017), LCE (Domhan et al., 2015), and a "last seen value" heuristic (Li et al., 2017). Absent results for a model indicate that it did not achieve a positive R^2. The results for Cuda-Convnet on the SVHN dataset are shown in Appendix Figure 7.

However, in the case $f(x_f)$ has poor out-of-sample generalization, we may mistakenly terminate the optimal configuration. If we assume that our estimate can be modeled as a Gaussian perturbation of the true value $\hat{y}_T \sim \mathcal{N}(y_T, \sigma(\mathbf{x}, \tau))$, then we can find the probability $p(\hat{y}_T \leq y_{\text{BEST}} | \sigma(\mathbf{x}, \tau)) = \Phi(y_{\text{BEST}}; y_T, \sigma)$, where $\Phi(\cdot; \mu, \sigma)$ is the CDF of $\mathcal{N}(\mu, \sigma)$. Note that in general the uncertainty will depend on both the configuration and τ, the number of points observed from the learning curve. Because frequentist models do not admit a natural estimate of uncertainty, we assume that σ is independent of $\mathbf{x}$ yet still dependent on τ and estimate it via Leave One Out Cross Validation.

Now that we can estimate the model uncertainty, given a new configuration $\mathbf{x}'$ and an observed learning curve $y'(t)_{1-\tau}$, we may set our termination criteria to be $p(\hat{y}_T \leq y_{\text{BEST}}) \geq \Delta$. Δ balances the trade-off between increased speedups and risk of prematurely terminating good configurations. In many cases, one may want several configurations that are close to optimal, for the purpose of ensembling. We offer two modifications in this case. First, one may relax the termination criterion to $p(\hat{y}_T \leq y_{\text{BEST}} - \delta) \geq \Delta$, which will allow configurations within δ of optimal performance to complete training. One can alternatively set the criterion based on the n^{th} best configuration observed, guaranteeing that with high probability the top n configurations will be fully trained.

4.1 Early Stopping for Meta-modeling

Baker et al. (2017) train a Q-learning agent to design convolutional neural networks. In this method, the agent samples architectures from a large, finite space by traversing a path from input layer to termination layer. However, the MetaQNN method uses 100 GPU-days to train 2700 neural architectures and the similar experiment by Zoph & Le (2017) utilized 10,000 GPU-days to train 12,800 models on CIFAR-10. The amount of computing resources required for these approaches makes them prohibitively expensive for large datasets (e.g., Imagenet) and larger search spaces. The main computational expense of reinforcement learning-based meta-modeling methods is training the neural network configuration to T epochs (where T is typically a large number at which the network stabilizes to peak accuracy).

We now detail the performance of a ν-SVR (RBF) SRM in speeding up architecture search using sequential configuration selection. First, we take 1,000 random models from the MetaQNN (Baker et al., 2017) search space. We simulate the MetaQNN algorithm by taking 10 random orderings of each set and running our early stopping algorithm. We compare against the LCE early stopping

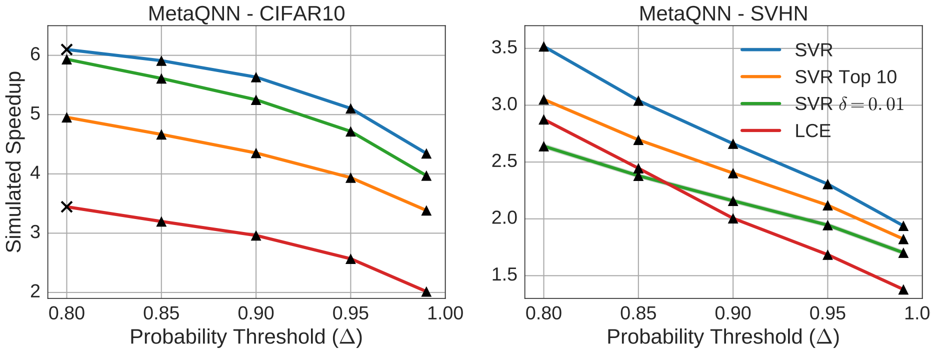

Figure 4: **Simulated Speedup in MetaQNN Search Space:** We compare the three variants of the early stopping algorithm presented in Section 4. Each ν-SVR SRM is trained using the first 100 learning curves, and each algorithm is tested on 10 independent orderings of the model configurations. Triangles indicate an algorithm that successfully recovered the optimal model for more than half of the 10 orderings, and X's indicate those that did not.

algorithm (Domhan et al., 2015) as a baseline, which has a similar probability threshold termination criterion. Our SRM trains off of the first 100 fully observed curves, while the LCE model trains from each individual partial curve and can begin early termination immediately. Despite this "burn in" time needed by an SRM, it is still able to significantly outperform the LCE model (Figure 4). In addition, fitting the LCE model to a learning curve takes between 1-3 minutes on a modern CPU due to expensive MCMC sampling, and it is necessary to fit a new LCE model each time a new point on the learning curve is observed. Therefore, on a full meta-modeling experiment involving thousands of neural network configurations, our method could be faster by several orders of magnitude as compared to LCE based on current implementations.

We furthermore simulate early stopping for ResNets trained on CIFAR-10. We found that only the probability threshold $\Delta = 0.99$ resulted in recovering the top model consistently. However, even with such a conservative threshold, the search was sped up by a factor of 3.4 over the baseline. While we do not have the computational resources to run the full experiment from Zoph & Le (2017), our method could provide similar gains in large scale architecture searches.

It is not enough, however, to simply simulate the speedup because meta-modeling algorithms typically use the observed performance in order to update an acquisition function to inform future sampling. In the reinforcement learning setting, the performance is given to the agent as a reward, so we also empirically verify that substituting $\hat{y}_T$ for y_T does not cause the MetaQNN agent to converge to a subpar policy. Replicating the MetaQNN experiment on CIFAR-10 (see Figure 5), we find that integrating early stopping with the Q-learning procedure does not disrupt learning and resulted in a speedup of 3.8x with $\Delta = 0.99$. The speedup is relatively low due to a conservative value of Δ. After training the top models to 300 epochs, we also find that the resulting performance (just under 93%) is on par with original results of Baker et al. (2017).

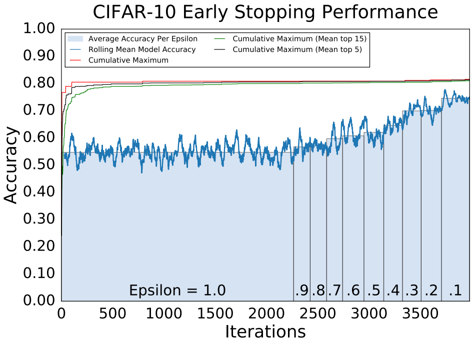

Figure 5: **MetaQNN on CIFAR-10 with Early Stopping:** A full run of the MetaQNN algorithm (Baker et al., 2017) on the CIFAR-10 dataset with early stopping. We use the ν-SVR SRM with a probability threshold $\Delta = 0.99$. Light blue bars indicate the average model accuracy per decrease in ϵ, which represents the shift to a more greedy policy. We also plot the cumulative best, top 5, and top 15 to show that the agent continues to find better architectures.

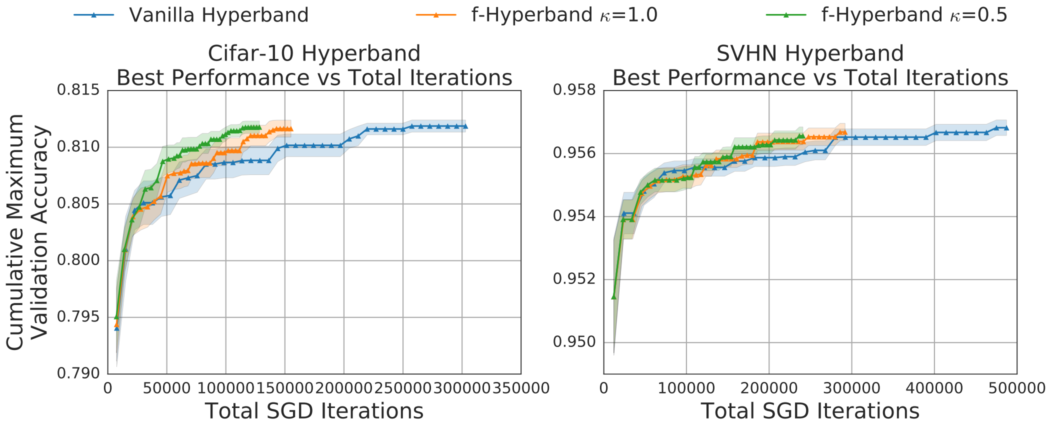

Figure 6: **Simulated Max Accuracy vs SGD Iterations for Hyperband:** We show the trajectories of the maximum performance so far versus total computational resources used for 40 consecutive Hyperband runs with $\eta = 3.0$ and $\Delta = 0.95$. f-Hyperband remains above the Hyperband curve at all iterations, and less aggressive settings for κ converge to the same or better final accuracy. Each triangle marks the completion of full Hyperband algorithm.

4.2 Early Stopping for Hyperparameter Optimization

Recently, Li et al. (2017) introduced Hyperband, a random search technique based on multi-armed bandits that obtains state-of-the-art performance in hyperparameter optimization in a variety of settings. The Hyperband algorithm trains a population of models with different hyperparameter configurations and iteratively discards models below a certain percentile in performance among the population until the computational budget is exhausted or satisfactory results are obtained.

4.2.1 Fast Hyperband

We present a Fast Hyperband (f-Hyperband) algorithm based on our early stopping scheme. During each iteration of successive halving, Hyperband trains n_i configurations to r_i epochs. In f-Hyperband, we train an SRM to predict y_{r_i} and do early stopping within each iteration of successive halving. We initialize f-Hyperband in exactly the same way as vanilla Hyperband, except once we have trained 100 models to r_i iterations, we begin early stopping for all future successive halving iterations that train to r_i iterations. By doing this, we exhibit no initial slowdown to Hyperband due to a "burn-in" phase. We also introduce a parameter κ which denotes the proportion of the n_i models in each iteration that must be trained to the full r_i iterations. This is similar to setting the criterion based on the n^{th} best model in the previous section. See Appendix section C for an algorithmic representation of f-Hyperband.

We empirically evaluate f-Hyperband using Cuda-Convnet trained on CIFAR-10 and SVHN datasets. Figure 6 shows that f-Hyperband evaluates the same number of unique configurations as Hyperband within half the compute time, while achieving the same final accuracy within standard error. When reinitializing hyperparameter searches, one can use previously-trained set of SRMs to achieve even larger speedups. Figure 8 in Appendix shows that one can achieve up to a 7x speedup in such cases.

5 Conclusion

In this paper we introduce a simple, fast, and accurate model for predicting future neural network performance using features derived from network architectures, hyperparameters, and time-series performance data. We show that the performance of drastically different network architectures can be jointly learned and predicted on both image classification and language models. Using our simple algorithm, we can speedup hyperparameter search techniques with complex acquisition functions, such as a Q-learning agent, by a factor of 3x to 6x and Hyperband—a state-of-the-art hyperparameter search method—by a factor of 2x, without disturbing the search procedure. We outperform all competing methods for performance prediction in terms of accuracy, train and test time, and speedups obtained on hyperparameter search methods. We hope that the simplicity and success of our method will allow it to be easily incorporated into current hyperparameter optimization pipelines for deep neural networks. With the advent of large scale automated architecture search (Baker et al., 2017; Zoph & Le, 2017), methods such as ours will be vital in exploring even larger and more complex search spaces.

REFERENCES

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *International Conference on Learning Representations*, 2017.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *JMLR*, 13 (Feb):281–305, 2012.

James Bergstra, Daniel Yamins, and David D Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML (1)*, 28:115–123, 2013.

Andrew Brock, Theodore Lim, JM Ritchie, and Nick Weston. Smash: One-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. AdaNet: Adaptive structural learning of artificial neural networks. *International Conference on Machine Learning*, 70:874–883, 2017.

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. *IJCAI*, 2015.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. *International Conference on Learning Representations*, 17, 2017.

Alex Krizhevsky. Cuda-convnet. *https://code.google.com/p/cuda-convnet/*, 2012.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *International Conference on Learning Representations*, 2017.

Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

J David Schaffer, Darrell Whitley, and Larry J Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 1–37, 1992.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1): 148–175, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *NIPS*, 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.

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

Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. *arXiv preprint arXiv:1704.00764*, 2017.

Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

Phillip Verbancsics and Josh Harguess. Generative neuroevolution for deep learning. *arXiv preprint arXiv:1312.5355*, 2013.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.

APPENDIX

A DATASETS AND ARCHITECTURES

Deep Resnets (TinyImageNet): We sample 500 ResNet architectures and train them on the TinyImageNet[†] dataset (containing 200 classes with 500 training images of 32 × 32 pixels) for 140 epochs. We vary depths, filter sizes and number of convolutional filter block outputs. Filter sizes are sampled from $\{3, 5, 7\}$ and number of filters is sampled from $\{2, 3, 4, ..., 22\}$. Each ResNet block is composed of three convolutional layers followed by batch normalization and summation layers. We vary the number of blocks from 2 to 18, giving us networks with depths varying between 14 and 110. Each network is trained for 140 epochs, using Nesterov optimizer. The learning rate is set to 0.1 and learning rate reduction and momentum are set to 0.1 and 0.9 respectively.

Deep Resnets (CIFAR-10): We sample 500 39-layer ResNet architectures from a search space similar to Zoph & Le (2017), varying kernel width, kernel height, and number of kernels. We train these models for 50 epochs on CIFAR-10. Each architecture consists of 39 layers: 12 *conv*, a 2x2 *max pool*, 9 *conv*, a 2x2 *max pool*, 15 *conv*, and *softmax*. Each *conv* layer is followed by batch normalization and a ReLU nonlinearity. Each block of 3 *conv* layers are densely connected via residual connections and also share the same kernel width, kernel height, and number of learnable kernels. Kernel height and width are independently sampled from $\{1, 3, 5, 7\}$ and number of kernels is sampled from $\{6, 12, 24, 36\}$. Finally, we randomly sample residual connections between each block of *conv* layers. Each network is trained for 50 epochs using the RMSProp optimizer, with weight decay 10^{-4}, initial learning rate 0.001, and a learning rate reduction to 10^{-5} at epoch 30 on the CIFAR-10 dataset.

MetaQNN CNNs (CIFAR-10 and SVHN): We sample 1,000 model architectures from the search space detailed by Baker et al. (2017), which allows for varying the numbers and orderings of convolution, pooling, and fully connected layers. The models are between 1 and 12 layers for the SVHN experiment and between 1 and 18 layers for the CIFAR-10 experiment. Each architecture is trained on SVHN and CIFAR-10 datasets for 20 epochs. Table 3 displays the state space of the MetaQNN algorithm.

Layer Type	Layer Parameters	Parameter Values
Convolution (C)	$i \sim$ Layer depth $f \sim$ Receptive field size $\ell \sim$ Stride $d \sim$ # receptive fields $n \sim$ Representation size	< 12 Square. $\in \{1, 3, 5\}$ Square. Always equal to 1 $\in \{64, 128, 256, 512\}$ $\in \{(\infty, 8], (8, 4], (4, 1]\}$
Pooling (P)	$i \sim$ Layer depth $(f, \ell) \sim$ (Receptive field size, Strides) $n \sim$ Representation size	< 12 Square. $\in \big\{(5, 3), (3, 2), (2, 2)\big\}$ $\in \{(\infty, 8]$, $(8, 4]$ and $(4, 1]\}$
Fully Connected (FC)	$i \sim$ Layer depth $n \sim$ # consecutive FC layers $d \sim$ # neurons	< 12 < 3 $\in \{512, 256, 128\}$
Termination State	$s \sim$ Previous State $t \sim$ Type	Global Avg. Pooling/Softmax

Table 3: **Experimental State Space For MetaQNN.** For each layer type, we list the relevant parameters and the values each parameter is allowed to take. The networks are sampled beginning from the starting layer. Convolutional layers are allowed to transition to any other layer. Pooling layers are allowed to transition to any layer other than pooling layers. Fully connected layers are only allowed to transition to fully connected or softmax layers. A convolutional or pooling layer may only go to a fully connected layer if the current image representation size is below 8. We use this space to both randomly sample and simulate the behavior of a MetaQNN run as well as directly run the MetaQNN with early stopping.

LSTM (PTB): We sample 300 LSTM models and train them on the Penn Treebank dataset for 60 epochs. Number of hidden layer inputs and lstm cells was varied from 10 to 1400 in steps of 20. Each network was trained for 60 epochs with batch size of 50 and trained the models using stochastic

[†] https://tiny-imagenet.herokuapp.com/

gradient descent. Dropout ratio of 0.5 was used to prevent overfitting. Dictionary size of 400 words was used to generate embeddings when vectorizing the data.

Cuda-Convnet (CIFAR-10 and SVHN): We train Cuda-Convnet architecture (Krizhevsky, 2012) with varying values of initial learning rate, learning rate reduction step size, weight decay for convolutional and fully connected layers, and scale and power of local response normalization layers. We train models with CIFAR-10 for 60 epochs and with SVHN for 12 epochs. Table 4 show the hyperparameter ranges for the Cuda Convnet experiments.

Experiment	Hyperparameter	Scale	Min	Max
CIFAR-10, Imagenet, SVHN	Initial Learning Rate	Log	5×10^{-5}	5
	Learning Rate Reductions	Integer	0	3
CIFAR-10, SVHN	Conv1 L_2 Penalty	Log	5×10^{-5}	5
	Conv2 L_2 Penalty	Log	5×10^{-5}	5
	Conv3 L_2 Penalty	Log	5×10^{-5}	5
	FC4 L_2 Penalty	Log	5×10^{-5}	5
	Response Normalization Scale	Log	5×10^{-6}	5
	Response Normalization Power	Linear	1×10^{-2}	3

Table 4: Range of hyperparameter settings used for the Hyperband experiment (Section 4.1)

B Hyperparameter selection in Random Forest and SVM based experiments

When training SVM and Random Forest we divided the data into training and validation and used cross validation techniques to select optimal hyperparameters. The SVM and RF model was then trained on full training data using the best hyperparameters. For random forests we varied number of trees between 10 and 800, and varied ratio of number of features from 0.1 to 0.5. For ν-SVR, we perform a random search over 1000 hyperparameter configurations from the space $C \sim$ LogUniform(10^{-5}, 10), $\nu \sim$ Uniform(0, 1), and $\gamma \sim$ LogUniform(10^{-5}, 10) (when using the RBF kernel).

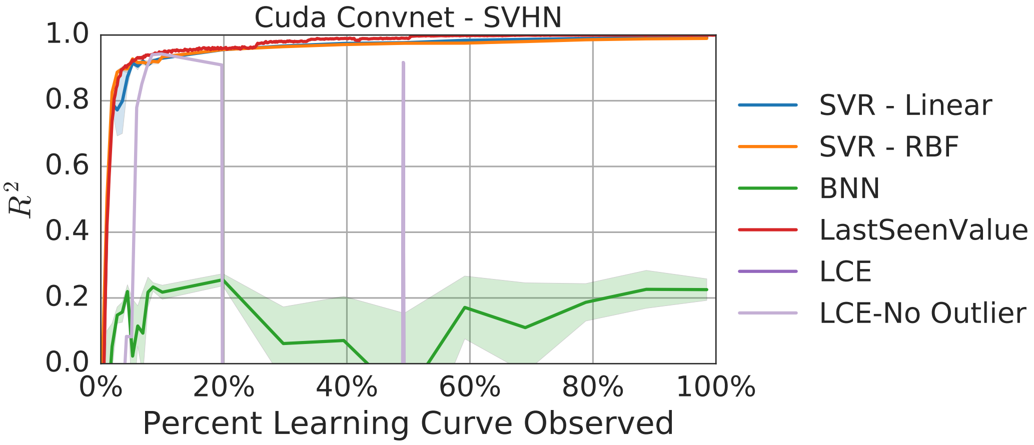

Figure 7: **Cuda Convnet SVHN Performance Prediction Results:** We plot the performance of each method versus the percent of learning curve observed for the Cuda Convnet SVHN experiment. For BNN and ν-SVR (linear and RBF), we sample 10 different training sets, plot the mean R^2, and shade the corresponding standard error. We compare our method against BNN (Klein et al., 2017), LCE (Domhan et al., 2015), and a "last seen value" heuristic (Li et al., 2017). Absent results for a model indicate that it did not achieve a positive R^2.

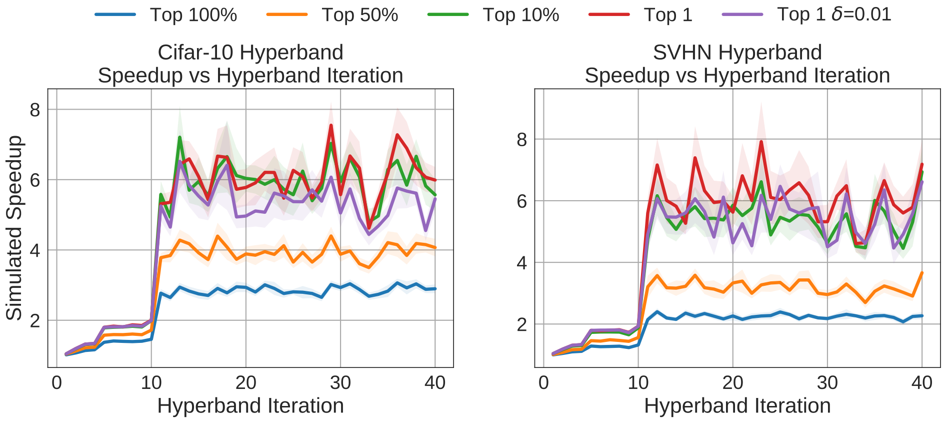

Figure 8: **Simulated Speedup on Hyperband vs Hyperband Iteration:** We show the speedup using the f-Hyperband algorithm over Hyperband on 40 consecutive runs with $\eta = 3.0$ and $\Delta = 0.95$. The major jump in speedup comes at iteration 10, where we have trained more than 100 models to the full R iterations.

C F-HYPERBAND

Algorithm 1 of this text replicates Algorithm 1 from Li et al. (2017), except we initialize two dictionaries: D to store training data and M to store performance prediction models. $D[r]$ will correspond to a dictionary containing all datasets with prediction target epoch r. $D[r][\tau]$ will correspond to the dataset for predicting y_r based on the observed $y(t)_{1-\tau}$, and $M[r][\tau]$ will hold the corresponding performance prediction model. We will assume that the performance prediction model will have a `train` function, and a `predict` function that will return the prediction and standard deviation of the prediction. In addition to the standard Hyperband hyperparameters R and η, we include Δ and δ described in Section 4 and κ. During each iteration of successive halving, we train n_i configurations to r_i epochs; κ denotes the fraction of the top n_i models that should be run to the full r_i iterations. This is similar to setting the criterion based on the n^{th} best model in the previous section.

We also detail the `run_then_return_validation_loss` function in Algorithm 2. This algorithm runs a set of configurations, adds training data from observed learning curves, trains the performance prediction models when there is enough training data present, and then uses the models to terminate poor configurations. It assumes we have a function `max_k`, which returns the k^{th} max value or $-\infty$ if the list has less than k values.

Algorithm 1: f-Hyperband

```
input      : $R$        –   (Max resources allocated to any configuration)
             $\eta$     –   (default $\eta = 3$)
             $\Delta$   –   (Probability threshold for early termination)
             $\delta$   –   (Performance offset for early termination)
             $d$        –   (# points required to train performance predictors)
             $\kappa$   –   (Proportion of models to train)
initialize : $D = \text{dict()}$
             $M = \text{dict()}$
             $s_{\text{max}} = \lfloor \log_\eta(R) \rfloor$
             $B = (s_{\text{max}} + 1)R$
for $s \in \{s_{max}, \dots, 0\}$ do
    $n = \lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil, \quad r = R\eta^{-s}$
    // begin SUCCESSIVEHALVING with ($n$, $r$) inner loop
    $T =$ get_hyperparameter_configuration($n$)
    for $i \in \{0, \dots, s\}$ do
        $n_i = \lfloor n\eta^{-i} \rfloor, \quad r_i = r\eta^i$
        $n_{\text{next}} = \lfloor \frac{n_i}{\eta} \rfloor$ if $i! = s$ else $1$
        $L =$ run_then_return_validation_loss($T, r_i, n_{\texttt{next}}, D, M$)
        $T =$ top_k($T, L, \lfloor \frac{n_i}{\eta} \rfloor$)
    end
end
```

Algorithm 2: run_then_return_validation_loss

```
input      : $T$   –   hyperparameter configurations
             $r$   –   resources to use for training
             $n$   –   # configurations in next iteration of successive halving
             $D$   –   dictionary storing training data
             $M$   –   dictionary storing performance prediction models
initialize : $L$ = []
for $t \in T$ do
    $\ell$ = []
    for $i \in \{0, \dots, r-1\}$ do
        $\ell_i$ = run_one_epoch_return_validation_loss($t$)
        $\ell$.append($\ell_i$)
        if $M[r][i]$.trained() then
            $\hat{y}_r, \sigma$ = $M[r][i]$.predict($\ell$)
            if $\Phi$(max_k($L$, $\kappa n$) $- \delta; \hat{y}_r, \sigma) \geq \Delta$ then
                $L$.append($\hat{y}_r$)
                break
            end
        end
        else if $i == r - 1$ then
            $L$.append($\ell_i$)
        end
    end
    if length($D[r][0]$) $< d$ and length($\ell$) $== r$ then
        $\{D[r][i]$.append($\{\ell[0, \dots, i], \ell[r]\}$)$: i \in \{0, \dots, r-1\}\}$
        if not$M[r][i]$.trained() then
            $M[r][i]$.train($D[r][i]$)
        end
    end
end
return $L$
```