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 meta-modeling are computationally expensive due to the need to train a large number of model configurations. In this paper, we propose a method for accelerating meta-modeling using a method for predicting final performance of neural networks using a frequentist regression models trained using features based on network architectures, hyperparameters, and time-series validation performance data of partially-trained networks. Our method obtains state-of-the-art performance in predicting the final accuracy of models in both visual classification and language modeling domains, is effective for predicting performance of drastically varying model architectures, and even generalizes between model classes. Our early stopping scheme, based on this prediction method, obtains a speedup of up to 6x on reinforcement learning-based neural architecture search, while still identifying the optimal model configurations.

1 Introduction

Meta-modeling methods [2, 13, 1, 14] aim to design neural network architectures from scratch to reduce the amount of human expertise and labor usually required. However, these methods require training a large number of neural network configurations for identifying the right network architecture—and are hence computationally expensive. For example, Zoph and Le [14] train 12,800 networks using 10,000 GPUdays to design a state-of-the-art CNN for the CIFAR-10 dataset. We propose to speedup meta-modeling methods by automatically predicting final neural network performance from partially-trained learning curves and early-terminating subpar model configurations. We briefly summarize related work next.

Neural Network Performance Prediction: Domhan et al.[5] 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 [5], Klein et al. [6] train Bayesian neural networks for predicting unobserved learning curves using a training set of fully and partially observed learning curves. Both methods rely on Markov chain Monte Carlo (MCMC) sampling procedures and handcrafted learning curve basis functions.

Meta-modeling: The earliest meta-modeling approaches were based on genetic algorithms [9, 11, 13] or Bayesian optimization [2, 10]. More recently, reinforcement learning methods have become popular. Baker et al. [1] introduce MetaQNN, a Q-learning-based method to designs CNNs. Zoph and Le [14] use policy gradients to design CNNs and recurrent cell architectures. Several meta-modeling methods [4, 8, 15, 3, 12] have been proposed since the publication of [1, 14].

2 Neural Network Performance Prediction

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 $y(t)$ (see [6]). For each configuration $\mathbf{x}$ trained for T epochs, we record a time-series

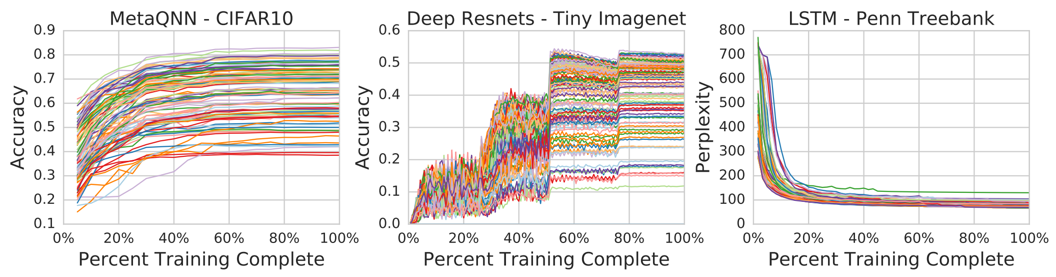

Figure 1: **Example learning curves** showing diversity in convergence times and curve shapes.

$y(T) = y_1, y_2, \ldots, y_T$ of validation accuracies. We train a population of n configurations, obtaining a set $\mathcal{S} = \{(\mathbf{x}^1, y^1(t)), (\mathbf{x}^2, y^2(t)), \ldots, (\mathbf{x}^n, y^n(t))\}$. 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 $x_f = \{u_{\mathbf{x}}, y(t)_{1-\tau}\}$. For clarity, we train $T-1$ *sequential regression models* (SRM), where each successive model uses one more point of the time-series validation data.

Features: We use features based on time-series (TS) validation accuracies, architecture parameters (AP), and hyperparameters (HP). (1) TS: These include the validation accuracies $y(t)_{1-\tau} = (y_t)_{t=1,2,\ldots,\tau}$ (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 (full list in Appendix).

2.1 Datasets and Training Procedures

We experiment with very deep CNNs (e.g., Resnet) trained on image classification datasets and with LSTMs trained with Penn Treebank (PTB) (details in Appendix Section A).

Deep Resnets (TinyImageNet): We sample 500 Resnet architectures with varying depths (between 14–110), filter sizes and number of convolutional filter block outputs and train them on the TinyImageNet[*] dataset (200 classes with 500 32×32-size train images) for 140 epochs.

MetaQNN CNNs (CIFAR-10 and SVHN): We sample 1,000 architectures from the MetaQNN [1] search space, varying the numbers and orderings of convolution, pooling, and fully connected layers. The model depths are 1–12 layers (SVHN) or 1–18 (CIFAR-10). Models are trained for 20 epochs.

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

2.2 Prediction Performance

Choice of Regression Method: We train our SRMs on 100 randomly sampled neural network configurations and compute the prediction accuracy over the rest of the dataset using the coefficient of determination R^2. We experiment with a few different frequentist regression models, including ordinary least squares, random forests, and ν-support vector machine regression (ν-SVR). 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 (see Appendix for comparison)

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) [6], (2) the learning curve extrapolation (LCE) method [5], and (3) the last seen value (LastSeenValue) heuristic [7]. Figure 3 shows that in all neural network configuration spaces and across all datasets, either one or both SRMs outperform the competing methods. Since LCE [5] and BNN [6] contain basis functions for exponential learning rate decay (ELRD), instead of stepwise learning rate decay used in our datasets (which has become a common practice), we trained 630 random nets with ELRD from the 1000 MetaQNN-CIFAR10 nets. Predicting from 25% of the

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

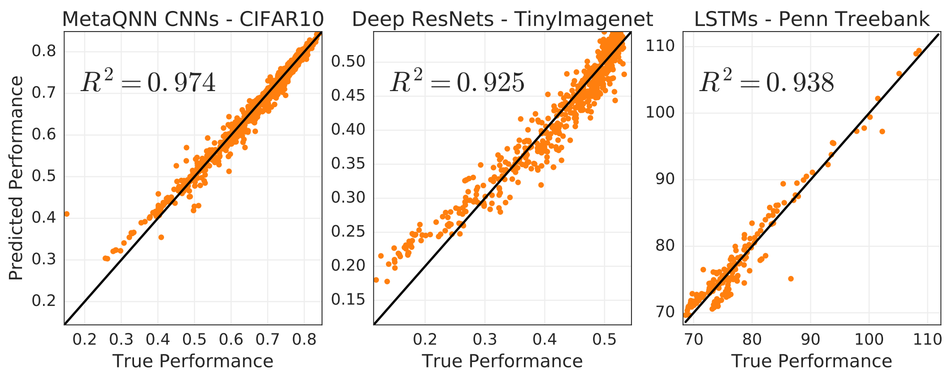

Figure 2: **Predicted versus true final performance** with a ν-SVR (RBF) model trained with 25% of learning curve (TS), along with architecture (AP) and hyperparameter (HP) features.

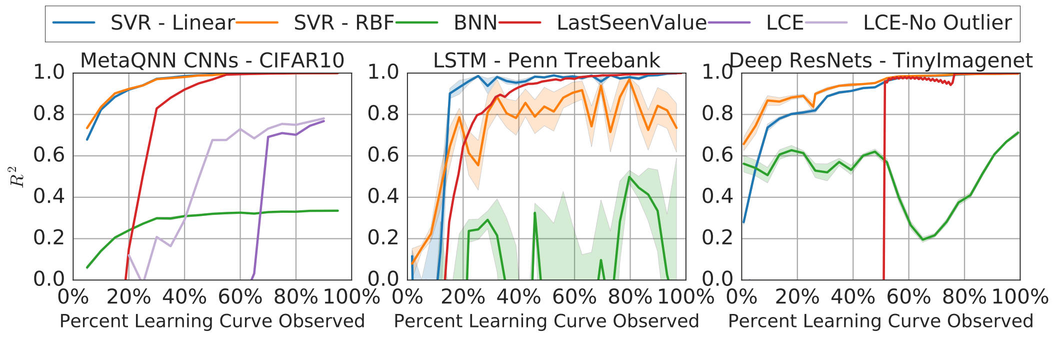

Figure 3: **Performance prediction results** comparing against BNN [6], LCE [5], and a "last seen value" heuristic [7]. Absent results for a model indicate that it did not achieve a positive R^2.

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. SRMs are also much faster to train and do inference as compared to [5, 6].

3 Accelerating Meta-modeling with Performance Prediction

We will use the sequential regression models to speedup meta-modeling using an **early stopping scheme** to determine whether to continue training a partially trained model configuration. 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. 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

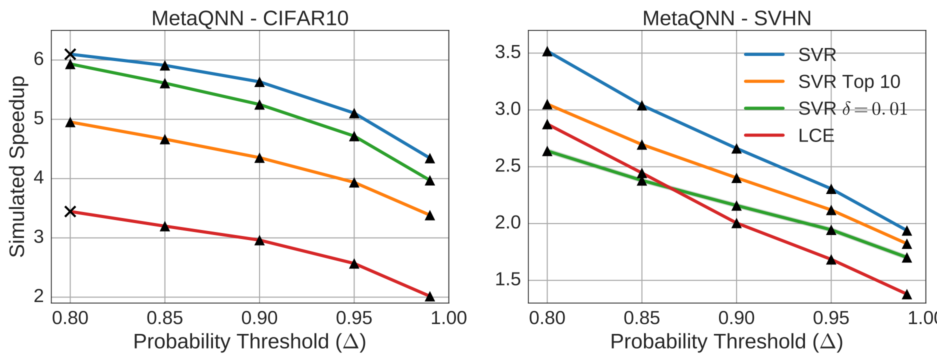

Figure 4: We simulate **speedup in MetaQNN search space** with SRMs and compare against existing methods. Triangles indicate an algorithm that successfully recovered the optimal model for more than half of the 10 orderings, and X's indicate those that did not.

to $p(\hat{y}_T \leq y_{\text{BEST}} - \delta) \geq \mathit{\Delta}$, 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.

Experiment with MetaQNN: Baker et al [1] develop MetaQNN, a Q-learning-based method to design convolutional neural networks. The main computational expense of MetaQNN and other 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 use ν-SVR (RBF) SRM to speed up MetaQNN. First, we take 1,000 random models from the MetaQNN [1] 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 algorithm [5] 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 and obtains up to 6x speedup on the original method (Figure 4).

In addition to the simulation, we perform a full run of the MetaQNN algorithm with early stopping. In the reinforcement learning setting, the performance is given to the agent as a reward, so we need to 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 $\mathit{\Delta} = 0.99$. The speedup is relatively low due to a conservative value of $\mathit{\Delta}$. 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 [1].

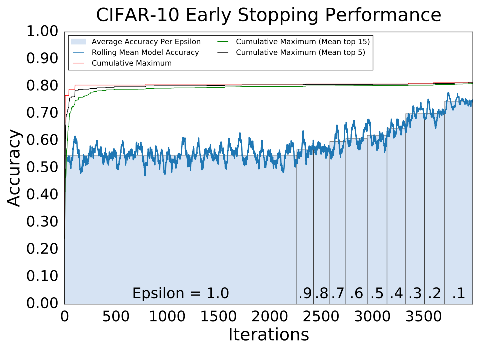

Figure 5: A full run of **MetaQNN on CIFAR-10 with early stopping**, with ν-SVR SRM with a probability threshold $\mathit{\Delta} = 0.99$. Light blue bars indicate the average model accuracy per decrease in ϵ, which represents the shift to a more greedy policy. The cumulative best, top 5, and top 15 show that the agent continues to find better architectures.

4 Conclusion

In this paper, we focus on speeding up meta-modeling methods with an early-stopping scheme that utilizes a simple neural network performance prediction model. However, our method also accelerates hyperparameter search methods (for example, our experiments show upto 2.5x speedup on Hyperband [7]). With the advent of large scale automated architecture search [1, 14, 4, 8, 15, 3], methods such as ours will be vital in exploring large and complex search spaces.

References

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

[2] 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.

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

[4] 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.

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

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

[7] 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.

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

[9] 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*, pages 1–37, 1992.

[10] 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.

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

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

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

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

[15] 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.

MetaQNN CNNs (CIFAR-10 and SVHN): We sample 1,000 model architectures from the search space detailed by [1], 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 1 displays the state space of the MetaQNN algorithm.

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

Table 1: **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 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.

B Ablation Study on Feature Sets

Table 2 shows that times-series features explain the largest fraction of the variance in all cases, though architecture features are almost as important for Resnets. Figure 2 shows the true vs. predicted performance. 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.

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

Feature Set	MetaQNN (CIFAR-10)	Resnets (TinyImageNet)	LSTM (Penn Treebank)
TS	93.98 ± 0.15	86.52 ± 1.85	97.81 ± 2.45
AP	27.45 ± 4.25	84.33 ± 1.7	16.11 ± 1.13
HP	12.60 ± 1.79	8.78 ± 1.14	3.98 ± 0.88
TS+AP	84.09 ± 1.4	88.82 ± 2.95	96.92 ± 2.8
AP+HP	27.01 ± 5.2	81.71 ± 3.9	15.97 ± 2.57
TS+AP+HP	94.44 ± 0.14	91.8 ± 1.1	98.24 ± 2.11

Table 2: **Ablation Study on Feature Sets:** TS refers to time-series features obtained from partially observed learning curves, AP refers to architecture parameters (e.g., #layers), and HP refers to training hyperparameters. All results with SVR (RBF). 25% of learning curve used for TS.

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 3: **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.

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.

C Comparing Frequentist Regression Methods

We experiment with a few different frequentist regression models, including ordinary least squares, random forests, and ν-support vector machine regression (ν-SVR), as shown in Table 3. 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).