Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves

Tobias Domhan, Jost Tobias Springenberg, Frank Hutter
University of Freiburg
Freiburg, Germany
{domhant,springj,fh}@cs.uni-freiburg.de

Abstract

Deep neural networks (DNNs) show very strong performance on many machine learning problems, but they are very sensitive to the setting of their hyperparameters. Automated hyperparameter optimization methods have recently been shown to yield settings competitive with those found by human experts, but their widespread adoption is hampered by the fact that they require more computational resources than human experts. Humans have one advantage: when they evaluate a poor hyperparameter setting they can quickly detect (after a few steps of stochastic gradient descent) that the resulting network performs poorly and terminate the corresponding evaluation to save time. In this paper, we mimic the early termination of bad runs using a probabilistic model that extrapolates the performance from the first part of a learning curve. Experiments with a broad range of neural network architectures on various prominent object recognition benchmarks show that our resulting approach speeds up state-of-the-art hyperparameter optimization methods for DNNs roughly twofold, enabling them to find DNN settings that yield better performance than those chosen by human experts.

1 Introduction

Deep neural networks (DNNs) trained via backpropagation currently constitute the state-of-the-art for many classification problems, such as object recognition from images [Krizhevsky *et al.*, 2012; Donahue *et al.*, 2014] or speech recognition from audio data (see [Deng *et al.*, 2013] for a recent review). Unfortunately, they are also very sensitive to the setting of their hyperparameters [Montavon *et al.*, 2012]. While good settings are hard to find by non-experts, automatic hyperparameter optimization methods have recently been shown to yield performance competitive with human experts, and in some cases even surpassed them [Bergstra *et al.*, 2011; Snoek *et al.*, 2012; Dahl *et al.*, 2013; Bergstra *et al.*, 2013].

However, fitting large DNNs is computationally expensive and the time overhead of automated hyperparameter optimization hampers its widespread adoption. Instead, many human deep learning experts still perform manual hyperparameter search, relying on a "bag of tricks" to determine model hyperparameters and learning rates for stochastic gradient descent (SGD) [Montavon *et al.*, 2012]. Using this acquired knowledge they can often tell after a few SGD steps whether the training procedure will converge to a model with competitive performance or not. To save time, they then prematurely terminate runs expected to perform poorly, allowing them to make more rapid progress than automated methods (which train even poor models until the end).

In this work, we mimic this early termination of bad runs with the help of a probabilistic model that extrapolates performance from the first part of a learning curve to its remainder, enabling us to automatically identify and terminate bad runs to save time. After discussing related work on hyperparameter optimization and studies of learning curves (Section 2), we introduce our probabilistic approach for extrapolating learning curves and show how to use it to devise a *predictive termination criterion* that can be readily combined with any hyperparameter optimization method (Section 3). Experiments with different neural network architectures on the prominent object recognition benchmarks CIFAR-10, CIFAR-100 and MNIST show that predictive termination speeds up current hyperparameter optimization methods for DNNs by roughly a factor of two, enabling them to find DNN settings that yield better performance than those chosen by human experts (Section 4).

2 Foundations and Related Work

We first review modern hyperparameter optimization methods and previous attempts to model learning curves.

2.1 Hyperparameter Optimization Methods

Given a machine learning algorithm A having hyperparameters $\lambda_1, \dots, \lambda_n$ with respective domains $\Lambda_1, \dots, \Lambda_n$, we define its hyperparameter space as $\mathbf{\Lambda} = \Lambda_1 \times \cdots \times \Lambda_n$. For each hyperparameter setting $\boldsymbol{\lambda} \in \mathbf{\Lambda}$, we use $A_{\boldsymbol{\lambda}}$ to denote the learning algorithm A using this setting. We further use $l(\boldsymbol{\lambda}) = \mathcal{L}(A_{\boldsymbol{\lambda}}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}})$ to denote the validation loss (e.g., misclassification rate) that $A_{\boldsymbol{\lambda}}$ achieves on data $\mathcal{D}_{\text{valid}}$ when trained on $\mathcal{D}_{\text{train}}$. The hyperparameter optimization problem is then to find $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ minimizing $l(\boldsymbol{\lambda})$.

For decades, the de-facto standard for hyperparameter optimization in machine learning has been a simple grid search.

Other approaches proposed over the years include racing algorithms [Maron and Moore, 1994] and gradient search [Bengio, 2000]. Recently, it has been shown that a simple random search can perform much better than grid search, particularly for high-dimensional problems with low intrinsic dimensionality [Bergstra and Bengio, 2012]. More sophisticated Bayesian optimization methods perform even better and have yielded new state-of-the-art results for several datasets [Bergstra *et al.*, 2011; Hutter *et al.*, 2011; Snoek *et al.*, 2012; Bergstra *et al.*, 2013].

Bayesian Optimization (see, e.g., [Jones *et al.*, 1998; Brochu *et al.*, 2010]) constructs a probabilistic model $\mathcal{M}$ of f based on point evaluations of f and any available prior information, uses model $\mathcal{M}$ to select subsequent configurations $\boldsymbol{\lambda}$ to evaluate, updates $\mathcal{M}$ based on the new measured performance at $\boldsymbol{\lambda}$, and iterates.

The three most popular implementations of Bayesian optimization are Spearmint [Snoek *et al.*, 2012], which uses a Gaussian process (GP) [Rasmussen and Williams, 2006] model for $\mathcal{M}$; SMAC [Hutter *et al.*, 2011], which uses random forests [Breiman, 2001] modified to yield an uncertainty estimate [Hutter *et al.*, 2014]; and the Tree Parzen Estimator (TPE) [Bergstra *et al.*, 2011], which constructs a density estimate over good and bad instantiations of each hyperparameter to build $\mathcal{M}$. Eggensperger *et al.* [2013] empirically compared these three systems, concluding that Spearmint's GP-based approach performs best for problems with few numerical (and no other) hyperparameters, and that SMAC's and TPE's tree-based approach performs best for high-dimensional and partly discrete hyperparameter optimization problems, as they occur in optimizing DNNs. We therefore use SMAC and TPE in this study.

2.2 Modeling Learning Curves

The term *learning curve* appears in the literature for describing two different phenomena: (1) the performance of an iterative machine learning algorithm as a function of its training time or number of iterations; and (2) the performance of a machine learning algorithm as a function of the size of the dataset it has available for training. While our work concerns learning curves of type 1, we describe related work on modelling both types of learning curves.

Learning curves of type 1 are very popular for visualizing the concept of overfitting: while performance on the training set tends to improve over time, test performance often degrades eventually. The study of these learning curves has led to early stopping heuristics aiming to terminate training before overfitting occurs (see, e.g., [Yao *et al.*, 2007]). We note that the goal behind our new predictive termination criterion is different: we predict validation performance and terminate a run when it is unlikely to beat the performance of the best model we have encountered so far.

In parallel to our work, Swersky *et al.* [2014] devised a GP-based Bayesian optimization method that includes a learning curve model. They used this model for temporarily pausing the training of machine learning models, in order to explore several promising hyperparameter configurations for a short time and resume training on the most promising models later on. Swersky *et al.* [2014] successfully applied this technique to matrix factorization, online Latent Dirichlet Allocation (LDA) and logistic regression. However, so far it does not work well for deep neural networks, possibly since it is limited to one particular parametric learning curve model that may not describe learning curves of deep networks well.[1]

Learning curves of type 2 have been studied to extrapolate performance from smaller to larger datasets. In early work, Frey and Fisher [1999] estimated the amount of data needed by a decision tree to achieve a desired accuracy using linear, logarithmic, exponential and power law parametric models. Subsequent work predicted the performance of multiple machine learning algorithms using a total of 6 parametric models: a power law model with two and three parameters, a logarithmic model, the vapor pressure model, the Morgan-Mercer-Flodin (MMF) model, and the Weibull model [Gu *et al.*, 2001]. More recently, e.g., Kolachina *et al.* [2012] predicted how a statistical machine translation system would perform if more data was available; they used 6 parametric models and concluded that the three parameter power law is most suitable for their task.

All these approaches for extrapolating learning curves have in common that they use maximum likelihood fits of each parametric model by itself. In contrast to the probabilistic approach we propose in this work, the curve models are thus neither combined to increase their representative power nor do they account for uncertainty in the data and model parameters.

3 Extrapolation of Learning Curves

In this paper, we focus on learning curves that describe the performance of DNNs as a function of the number of stochastic gradient descent (SGD) steps. We measure performance as classification accuracy on a validation set.

3.1 Learning Curve Model

In this section, we describe how we probabilistically extrapolate from a short initial portion of a learning curve to a later point. When running SGD on DNNs we measure validation performance in regular intervals. Let $y_{1:n}$ denote the observed performance values for the first n intervals. In our problem setup, we observe $y_{1:n}$ and aim to predict performance y_m after a large number of intervals $m \gg n$. We solve this problem using a probabilistic model.

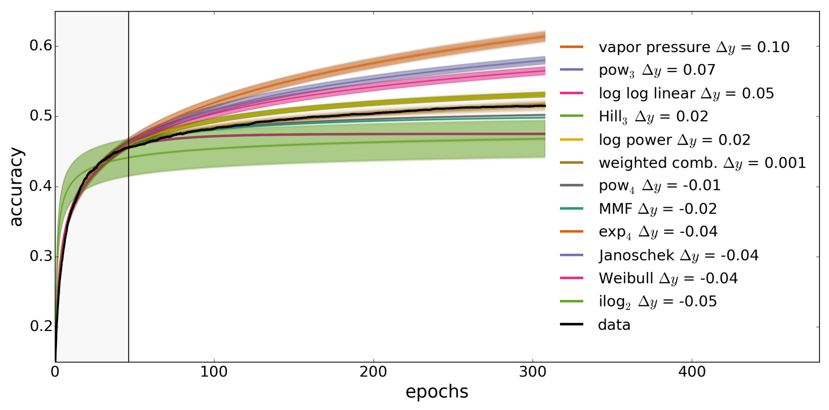
Parametric Learning Curve Models

Our basic approach is to model the partially observed learning curve $y_{1:n}$ by a set of parametric model families $\{f_1, \dots, f_K\}$. Each of these parametric functions f_k is described through a set of parameters $\boldsymbol{\theta}_k$. Assuming additive Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, we can use each f_k to model performance at time step t as $y_t = f_k(t|\boldsymbol{\theta}) + \epsilon$; the probability of a single observation y_t under model f_k is hence given as

$$p(y_t|\boldsymbol{\theta}_k, \sigma^2) = \mathcal{N}(y_t; f_k(t|\boldsymbol{\theta}_k), \sigma^2). \qquad (1)$$

We chose a large set of parametric curve models from the literature whose shape coincides with our prior knowledge

[1]Based on personal communication with the authors.



Reference name	Formula
vapor pressure	$\exp(a + \frac{b}{x} + c\log(x))$
pow$_3$	$c - ax^{-\alpha}$
log log linear	$\log(a\log(x) + b)$
Hill$_3$	$\frac{y_{\max} x^{\eta}}{\kappa^{\eta} + x^{\eta}}$
log power	$\frac{a}{1+\left(\frac{x}{e^{b}}\right)^{c}}$
pow$_4$	$c - (ax + b)^{-\alpha}$
MMF	$\alpha - \frac{\alpha - \beta}{1+(\kappa x)^{\delta}}$
exp$_4$	$c - e^{-ax^{\alpha}+b}$
Janoschek	$\alpha - (\alpha - \beta)e^{-\kappa x^{\delta}}$
Weibull	$\alpha - (\alpha - \beta)e^{-(\kappa x)^{\delta}}$
ilog$_2$	$c - \frac{a}{\log x}$

Figure 1: Left: A typical learning curve and extrapolations from its first part (the end of which is marked with a vertical line), with each of the 11 individual parametric models. The legend is sorted by the residual of the predictions at epoch 300. Right: the formulas for our 11 parameteric learning curve models $f_k(x)$.

about the form of learning curves: They are typically increasing, saturating functions; for example functions from the power law or the sigmoidal family. In total we considered $K = 11$ different model families; Figure 1 shows an example for how each of these functions would model a typical learning curve and also provides their parametric formulas. We note that all of these models capture certain aspects of learning curves, but that no single model can describe all learning curves by itself, motivating us to combine the models in a probabilistic framework.

A Weighted Probabilistic Learning Curve Model

Instead of selecting an individual model we combine all K models into a single, more powerful, model. This combined model is given by a weighted linear combination:

$$f_{comb}(t|\boldsymbol{\xi}) = \sum_{k=1}^{K} w_k f_k(t|\boldsymbol{\theta}_k), \tag{2}$$

where the new combined parameter vector

$$\boldsymbol{\xi} = (w_1, \ldots, w_K, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K, \sigma^2) \tag{3}$$

comprises a weight w_k for each model, the individual model parameters $\boldsymbol{\theta}_k$, and the noise variance σ^2, and $y_t = f_{comb}(t|\boldsymbol{\xi}) + \epsilon$.

Given this model, a simple approach would be to find a maximum likelihood estimate for all parameters. However, this would not properly model the uncertainty in the model parameters. Since our predictive termination criterion aims at only terminating runs that are highly unlikely to improve on the best run observed so far we need to model uncertainty as truthfully as possible and will hence adopt a Bayesian perspective, predicting values y_m using Markov Chain Monte Carlo (MCMC) inference.

To enable such probabilistic inference we also need to place a prior probability on all parameters. It would be simplest to choose an uninformative prior, such that

$$p(\boldsymbol{\xi}) \propto 1. \tag{4}$$

However, with such an uninformative prior we put positive probability mass on parameterizations that yield learning curves which *decrease* after some time. To avoid this situation, we explicitly encode our knowledge into the prior that well-behaved learning curves are *increasing* saturating functions. We also restrict the weights to be positive only, allowing us to interpret each individual model as a non-negative additive component of a learning curve. This has the nice side effect that the predicted curve will only be flat once all models have flattened out.

Concretely, we define a new prior distribution over $\boldsymbol{\xi}$, which encodes the above intuition, as

$$p(\boldsymbol{\xi}) \propto \left(\prod_{k=1}^{K} p(w_k)p(\boldsymbol{\theta}_k)\right) p(\sigma^2)\mathbb{1}(f_{comb}(1|\boldsymbol{\xi}) < f_{comb}(m|\boldsymbol{\xi})) \tag{5}$$

and for all k

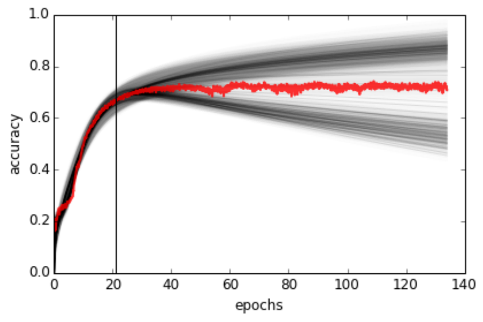
$$p(w_k) \propto \begin{cases} 1 & \text{if } w_k > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{6}$$

The $p(\boldsymbol{\theta}_k)$ and $p(\sigma^2)$ are still set to be uninformative, but are mentioned for the sake of completeness. The indicator function $\mathbb{1}(f_{comb}(1|\boldsymbol{\xi}) < f_{comb}(m|\boldsymbol{\xi}))$ ensures that no pathological model that decreases from the initial value to the point of prediction m gets any probability mass. Finally, the prior on the weights $p(w_k)$ ensures weights are never negative. Figure 2 visualizes the problem this modified prior solves: while the uninformative prior yields learning curves that decrease over time (Figure 2a), our new prior yields increasing saturating curves (Figure 2b).

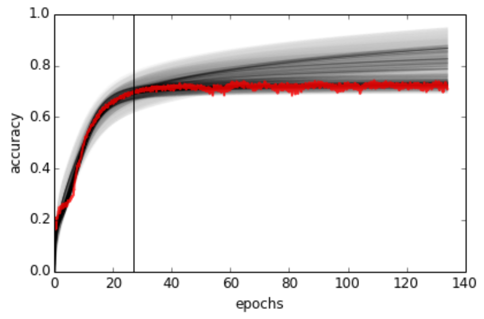
With these definitions in place we can finally perform MCMC sampling over the joint parameter and weight space $\boldsymbol{\xi}$ by drawing S samples $\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_S$ from the posterior

$$P(\boldsymbol{\xi}|y_{1:n}) \propto P(y_{1:n}|\boldsymbol{\xi})P(\boldsymbol{\xi}), \tag{7}$$

where $P(y_{1:n}|\boldsymbol{\xi})$ for the model combination is given as $P(y_{1:n}|\boldsymbol{\xi}) = \Pi_{t=1}^{n}\mathcal{N}(y_t; f_{comb}(t|\boldsymbol{\xi}), \sigma^2)$. To initialize the sampling procedure, we set all model parameters $\boldsymbol{\theta}_k$ to



(a) Uninformative prior



(b) Prior enforcing positive weights and increasing functions

Figure 2: The effect of the prior. We show posterior predictions using the uninformative prior and the prior encoding our knowledge about learning curves. The vertical line marks the point of prediction.

their (per-model) maximum likelihood estimate. The model weights are initialized uniformly, that is $w_k = \frac{1}{K}$. The noise parameter is also initialized to its maximum likelihood estimate $\hat{\sigma}^2 = \frac{1}{n}\sum_{t=1}^{n}(y_t - f_{comb}(t|\boldsymbol{\xi}))^2$.

A sample approximation for y_m with $m > n$ can then be formed as

$$\mathbb{E}[y_m|y_{1:n}] \approx \frac{1}{S}\sum_{s=1}^{S} f_{comb}(m|\boldsymbol{\xi}_s). \tag{8}$$

More importantly, since we also have an estimate of the parameter uncertainty and the predictive distribution $P(y_m|y_{1:n}, \boldsymbol{\xi})$ is a Gaussian for each fixed $\boldsymbol{\xi}$, we can estimate the probability that y_m exceeds a certain value $\hat{y}$ as

$$P(y_{\text{m}} \geq \hat{y}|y_{1:n}) = \int P(\boldsymbol{\xi}|y_{1:n})P(y_m > \hat{y}|\boldsymbol{\xi})d\boldsymbol{\xi} \tag{9}$$

$$\approx \frac{1}{S}\sum_{s=1}^{S} P(y_m > \hat{y}|\boldsymbol{\xi}_s) \tag{10}$$

$$= \frac{1}{S}\sum_{s=1}^{S}\left(1 - \Phi(\hat{y}; f_{comb}(m|\boldsymbol{\xi}_s), \sigma_s^2)\right), \tag{11}$$

where $\Phi(\cdot; \mu, \sigma^2)$ is the cumulative distribution function of the Gaussian with mean μ and variance σ^2.

We note that the entire learning curve extrapolation process is robust and fully automated, with MCMC sampling taking care of all free parameters.

3.2 Speeding up Hyperparameter Optimization

We use our predictive models to speed up hyperparameter optimizers as follows. Firstly, while the hyperparameter optimizer is running we keep track of the best performance $\hat{y}$ found so far (we initialize $\hat{y}$ to $-\infty$). Each time the optimizer queries the performance $l(\boldsymbol{\lambda})$ of a hyperparameter setting $\boldsymbol{\lambda}$ we train a DNN using $\boldsymbol{\lambda}$ as usual, except that we terminate this run early if our extrapolation model predicts the network to eventually yield worse performance than $\hat{y}$. More precisely, at regular intervals i during SGD training we measure and save validation set performance y_i. There are $e_{\max}$ epochs, k such intervals per epoch, and every p epochs, we gather the performance values $y_{1:n}$ of the n intervals so far and run MCMC to probabilistically extrapolate performance to the final step $m = k \times e_{\max}$. We then consider the predicted probability $P(y_{\text{m}} \geq \hat{y}|y_{1:n})$ that the network, after training for m intervals, will exceed the performance $\hat{y}$. If this probability is above a threshold δ then training continues as usual for the next p epochs. Otherwise, training is terminated and we return the expected validation error $1 - \mathbb{E}[y_m|y_{1:n}]$ (where $\mathbb{E}[y_m|y_{1:n}]$ is the expected accuracy from Equation 8) to the hyperparameter optimizer in lieu of the real (yet unknown) loss. We dub this procedure the *predictive termination criterion*. It is agnostic to the precise hyperparameter optimizer used and we will evaluate its performance using two different state-of-the-art optimizers.

We also note that, importantly, the network training does not need to be paused while the termination criterion is run: we simply run MCMC sampling on the available CPU cores while the network training continues on the GPU.

4 Experiments

To test our predictive termination criterion under a wide variety of different conditions, we performed experiments on a broad range of DNN architectures and datasets, carrying out a combined search over architectures and hyperparameters using two state-of-the-art hyperparameter optimization methods.

4.1 Experimental Setup

We used three popular datasets concerning object recognition from small-sized images: the image recognition datasets CIFAR-10 and CIFAR-100 [Krizhevsky, 2009] and the well known MNIST dataset [LeCun *et al.*, 1989]. CIFAR-10 and CIFAR-100 each contain 50,000 training and 10,000 test RGB images with 32×32 pixels that were taken from a subset of the 80-million tiny images database. While CIFAR-10 contains images from 10 categories, CIFAR-100 contains images from 100 categories and thus contains 10 times fewer examples per class. The MNIST dataset is a classic object recognition dataset consisting of 60,000 training and 10,000 test images with 28×28 pixels depicting hand-written digits to be classified into 10 digit classes.

For performing the hyperparameter search on CIFAR-10 and CIFAR-100, we randomly split the training data into training and validation sets containing 40,000 and 10,000 examples, respectively. Likewise, for MNIST, we split the training data into a training set containing 50,000 examples and a validation set containing 10,000 examples. We used the deep learning framework CAFFE [Jia *et al.*, 2014] to train DNNs on a single GPU per run. We further used the hyperparameter optimization toolbox HPOLIB [Eggensperger *et al.*, 2013] in combination with our implementation of the predictive termination criterion based on learning curve prediction, using the MCMC sampler EMCEE [Foreman-Mackey *et al.*, 2013].

For the predictive termination criterion we set the threshold to $\delta = 0.05$ in all experiments, that is, we stopped training a network if our extrapolation model was 95% certain that it would not improve over the best known performance $\hat{y}$ when fully trained. We ran the predictive termination criterion every $p = 30$ epochs. The number of intervals k per SGD epoch at which we evaluate validation performance was chosen separately for each architecture to reflect the cost of computing predictions on the validation data; we used $k = 10$ for fully connected networks and $k = 2$ for convolutional networks.

4.2 Fully Connected Networks

In the first experiment, we trained fully connected networks for classification on a preprocessed variant of CIFAR-10 and on MNIST. To make training a fully connected network on CIFAR-10 feasible we used the same pipeline as Swersky *et al.* [2013], who followed the approach from Coates *et al.* [2011] to create preprocessed CIFAR-10 features that act as a fixed convolutional layer while keeping the required computation time manageable. The pipeline first runs unsupervised k-means (with 400 centroids) on small patches of 6×6 pixels that are randomly extracted from the CIFAR-10 training set. It then builds a feature vector by convolving each image in CIFAR-10 with the centroids and averaging the responses over parts of the image. After this preprocessing step, the network contains only fully connected layers to classify the preprocessed data. We evaluated the benefits that our predictive termination criterion yields in combination with three different hyperparameter optimizers: SMAC, TPE, and random search (all described in Section 2.1). Each hyperparameter optimizer had a budget of evaluating a total of 100 networks. The maximum number of epochs e_{max} was 285.

In the MNIST experiment we fed the raw 784 pixel values to the fully connected networks. Our setup is thus comparable to most results on fully connected networks from the literature, e.g., the recent results on training dropout networks to classify MNIST [Srivastava *et al.*, 2014]. Training a single network on MNIST required between 5 and 20 minutes, and the hyperparameter optimizers had a fixed budget of evaluating a total of 500 networks.

DNN Hyperparameters

The hyperparameters for the fully connected network control several architectural choices and hyperparameters related to the optimization procedure. They include global hyperparameters (which apply to the whole network) and per-layer hyperparameters; since the number of layers is a hyperparameter itself, all hyperparameters of layer i are *conditional* on the number of layers being at least i. Both our hyperparameter optimizers SMAC and TPE can natively handle such conditional hyperparameters to solve the combined architecture search and hyperparameter optimization problem. We used stochastic gradient descent with momentum in all experiments. The learning rate was either fixed or changed according to the *inv* schedule[2]. All units in the network use rectified linear activation functions, and a softmax layer with dimensionality 10 is placed at the end to predict the 10 classes. Weights are either initialized with Gaussian noise or with the method proposed by Glorot and Bengio [2010]. Biases on the other hand are either initialized to zero or to a constant value. Dropout is optionally also applied to the input of the network. Table 1 details all hyperparameters, along with their ranges and the default values used to start the search; in total, the hyperparameter space to be searched has 10(network hyperparams) + 6(layers) $\times$ 7(hyperparams per layer) = 52 hyperparameters.

Network hyperparameters

Hyperparameter	min	max	default
init. learning rate (log)	1×10^{-7}	0.5	0.001
learning rate schedule (choice)	{inv, fixed}		fixed
inv schedule: lr. half-life (cond)	1	50	25
inv schedule: p (cond)	0.5	1.	0.71
momentum	0	0.99	0.6
weight decay (log)	5×10^{-7}	0.05	0.0005
batch size B	10	1000	100
number of layers	1	6	1
input dropout (Boolean)	{true, false}		false
input dropout rate (cond)	0.05	0.8	0.4

Fully connected layer hyperparameters

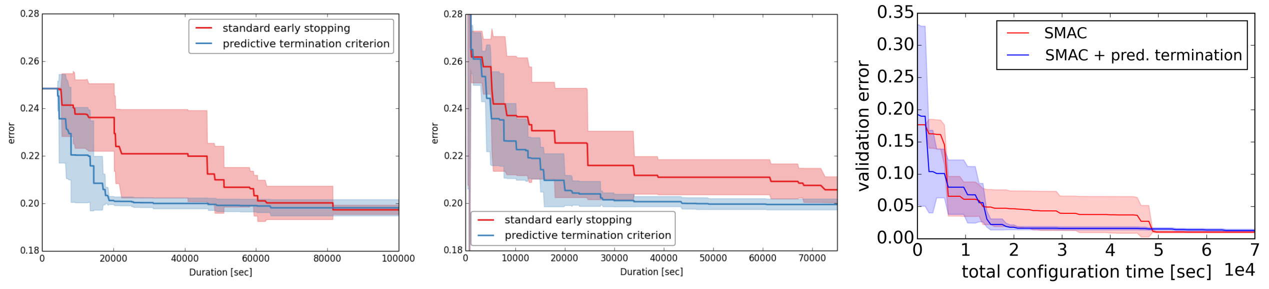
Hyperparameter	min	max	default
number of units	128	6144	1024
weight filler type (choice)	{Gaussian, Xavier}		Gaussian
Gaussian weight init σ (log; cond)	1×10^{-6}	0.1	0.005
bias init (choice)	{const-zero, const-value}		const-zero
constant value bias filler (cond)	0	1	0.5
dropout enabled (Boolean)	{true, false}		true
dropout ratio (cond)	0.05	0.95	0.5

Table 1: Hyperparameters for the fully connected networks and their ranges; lr. stands for learning rate, log indicates that the hyperparameter is optimized on a log scale, and cond indicates that the hyperparameter is conditional on being activated by the Boolean hyperparameter above it.

Results for preprocessed CIFAR-10

Figures 3a and 3b illustrate the speedups that our predictive termination yielded for training fully connected networks on preprocessed CIFAR-10 with SMAC and TPE. We ran each hyperparameter optimizer 5 times with different random

[2]The inv schedule is defined as $\alpha_t = \alpha_0(1 + \gamma t)^{-p}$, where α_0 is the initial learning rate. In order to be able to set bounds intuitively, instead of parameterizing γ directly we make the half-life of α a new hyperparameter, which for a given p can be transformed back into γ.



(a) SMAC on k-means CIFAR-10

(b) TPE on k-means CIFAR-10

(c) SMAC on MNIST

Figure 3: Benefit of predictive termination for SMAC and TPE when optimizing hyperparameters of fully connected networks. (a-b) Results for the preprocessed CIFAR-10 dataset (a) and (b). (c) Same plot for SMAC MNIST.

Method	# centroids	Error (%)
SVM [Coates *et al.*, 2011]	4000	20.40%
SVM [Coates *et al.*, 2011]	1600	22.10%
DNN [Swersky *et al.*, 2013]	400	21.10%
DNN (SMAC)	400	**19.22%**
DNN (TPE)	400	20.18%
DNN (random search)	400	19.90%

Table 2: Comparison of classification results on the k-means features extracted from the CIFAR10 dataset for different optimizers in comparison to previously published results.

seeds and show means and standard deviations of their validation errors across these runs. While all optimizers achieved strong performance for this architecture (around 20% validation error), their computational time requirements to achieve this level of performance differed greatly. As Figure 3a and Figure 3b show, our predictive termination criterion sped up both SMAC and TPE by at least a factor of two for reaching the same validation error as without it. Overall, the average time needed per hyperparamter optimization run was reduced from 40 to 18 hours. After this experiment, we applied the best models found by each optimizer to the test data to compute the classification error. These results—together with a comparison to random search and previous attempts for using k-means preprocessed CIFAR-10—are given in Table 2, showing that all optimizers we used found configurations with slightly better performance than the previously published results for this architecture, with SMAC yielding the overall best result for this experiment.

Figure 4 shows the effect of our predictive termination criterion on the different DNN training runs: the predictive termination criterion successfully terminated runs that do not reach top performance but rather converge slowly to mediocre results. The figure also shows that it was possible to terminate many poor runs quite early.

Results for MNIST

Figure 3c illustrates the speedups that predictive termination yielded for training fully connected networks on MNIST with SMAC. We only ran SMAC (10 runs) for this experiment since it had yielded the best results for CIFAR-10 (cf. Table 2). Consistent with the results on CIFAR-10, SMAC found networks with good performance with and without predictive termination (reaching approximately 1% validation error) and was much faster when using the predictive termination criterion: it reached 1% validation error in about 60% of the time it took a standard SMAC run to achieve the same performance.

Network hyperparameters			
Hyperparameter	min	max	default
init. learning rate (log)	1×10^{-7}	0.5	0.001
momentum	0	0.99	0.6
weight decay (log)	5×10^{-7}	0.05	0.0005
number of pooling layers	2	3	2
learning rate decay γ	0.9	1.	0.9998
Convolutional layer hyperparameters			
Hyperparameter	min	max	default
Gaussian weight init σ (log)	1×10^{-6}	0.1	0.005
weight lr. multiplier (log)	0.1	10.0	1.
number of units (small/large CNN)	16/64	64/192	32/96

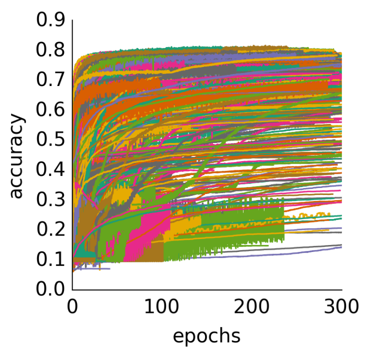
Table 3: Hyperparameters for the CNNs together with their ranges; lr. stands for learning rate, log indicates that the hyperparameter is optimized on a log scale.

4.3 Small Convolutional Neural Networks

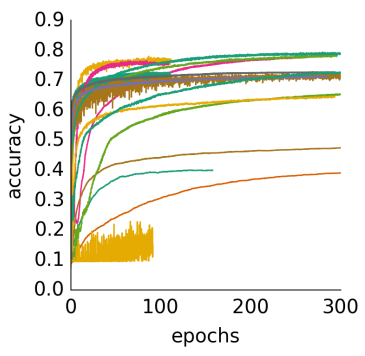
To study the generality of our learning curve extrapolation models, next, we turned to the problem of optimizing convolutional neural networks (CNNs), which constitute the state-of-the-art for visual object recognition (see [Krizhevsky *et al.*, 2012; Jarrett *et al.*, 2009] for recent explanations of convolutional layers). For this experiment we used the CIFAR-10 and CIFAR-100 datasets. The images from both datasets were preprocessed using a whitening transform, following the practice outlined by Goodfellow *et al.* [2013]. Other than that the CNNs were trained on the raw pixel images.

Small CNN Hyperparameters

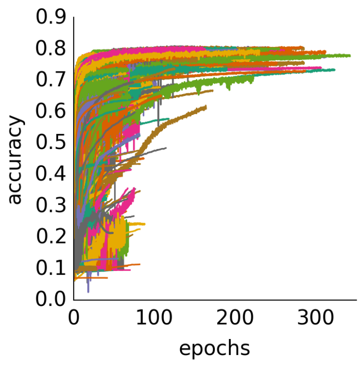
At the core, our CNNs no longer contain fully connected layers but rather use convolutional layers only, followed by a softmax classification layer. These layers extract features by convolving the input image or—for deeper layers—the output of the previous layer with a set of filters. Convolutional layers are regularly followed by dimensionality reduction steps (or pooling steps) which reduce the spatial dimensionality of



(a) Without predictive termination



(b) Random subset of Figure 4a



(c) With predictive termination

Figure 4: Comparison of learning curves for fully connected networks on CIFAR-10 with and without our predictive termination criterion. Best viewed in color. The plots contain all learning curves from the 10 runs of SMAC and TPE.

the feature map. In our first experiment with small CNNs, we model the hyperparameter space after the prominent architecture from Krizhevsky *et al.* [2012]. Concretely, we build a hyperparameter space that mimics the *layers-18pct* config from cuda-convnet[3]. Table 3 summarizes this hyperparameter space. In contrast to the experiments with fully connected networks, we now parameterized the number of layers indirectly, by choosing the number of *pooling layers* $(2-3)$. A convolutional layer is then always placed between these pooling layers. Each pooling layer always halves the input size by using max-pooling. The convolutional kernel size in each layer of our small CNNs is set to 5×5. These restrictions also make training quite fast (approximately 30 minutes per network). Overall, our small CNNs contain 5 network hyperparameters and 3 layer hyperparameters (conditioned on the number of convolutional layers used), resulting in a total of $5 + 3 \times 3 = 14$ configurable hyperparameters.

Results for Small CNNs on CIFAR-10

We optimized the small CNN hyperparameter space using 10 runs of both SMAC and TPE, with and without predictive termination. Each run had a maximum budget of evaluating 150 configurations. The maximum number of epochs $e_{\max}$ was 100. While all optimizers eventually found configurations with good validation performance of around 20% error, SMAC gave slightly better results on average ($19.4\% \pm 0.2\%$ error vs. $20\% \pm 0.4\%$ for TPE). We thus only present the results for SMAC here due to space constraints.

Figure 5a shows that predictive termination again sped up SMAC by a factor of at least two for finding the same validation error as without it. As shown in Figure 5b, predictive termination again consistently stopped bad runs early. When the best resulting configuration in terms of validation error was re-trained on the complete CIFAR-10 training dataset, it achieved a test error of 17.2% – slightly better than the baseline model (*layers-18pct* with 18% test error). A complete comparison of the test error for the best configuration found by the different optimizers is given in Table 4 (top).

[3]http://code.google.com/p/cuda-convnet/

CIFAR-10 classification error

Method	Error (%)
Small CNN + TPE	18.12%
Small CNN + SMAC	**17.47%**
Small CNN + TPE with pred. termination	18.08%
Small CNN + SMAC with pred. termination	**17.20%**

CIFAR-100 classification error

Method	Error (%)
Small CNN + SMAC	42.21%
Small CNN + SMAC with pred. termination	**41.90%**

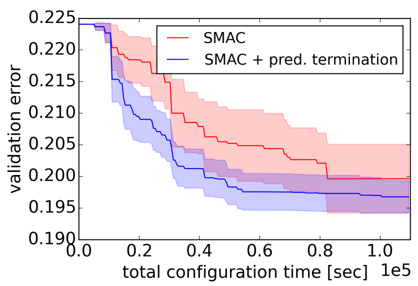
Table 4: Test error on CIFAR-10 and CIFAR-100 for the best hyperparameter configuration found for the small CNN search space.

Results for Small CNNs on CIFAR-100

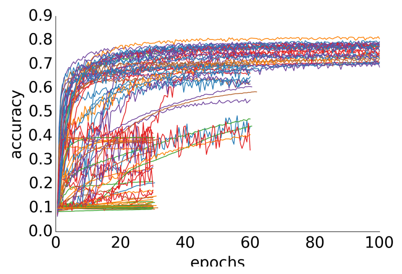
For CIFAR-100, we again optimized the same small CNN hyperparameter space as for CIFAR-10. For this experiment, we only used SMAC (10 runs with and without predictive termination) since it gave the best results in our experiments on CIFAR-10. As for CIFAR-10, each hyperparameter optimization run had a budget of evaluating 150 configurations. Figure 5c gives the results of this experiment. SMAC found configurations with approximately 43% validation error both with and without the predictive termination criterion, but was substantially faster with predictive termination and reached the same quality almost two times faster. Using the best configuration found with and without predictive termination to re-train on the full CIFAR-100 training data yielded slightly better test set performance for predictive termination (41.90% vs. 42.21%); in comparison, adapting the *layers-18pct* to CIFAR-100 yields 45% test error.

4.4 Large Convolutional Networks

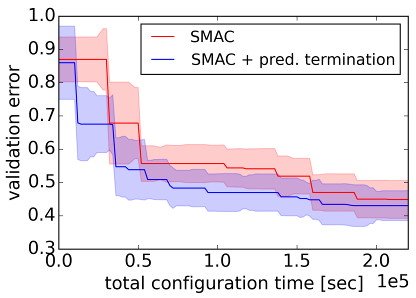
Finally, to test our learning curve prediction on state-of-the-art CNN models, we optimized the hyperparameters of a family of large convolutional networks on CIFAR-10.



(a) Small CNNs CIFAR-10



(b) CNN learning curves with predictive termination (CIFAR-10)



(c) Small CNNs CIFAR-100

Figure 5: Results for optimizing the small CNNs with SMAC on CIFAR-10/100. (a) Benefit of predictive termination for SMAC, for a total run-time of $120,000$ seconds. (b) Learning curves from one exemplary SMAC run with predictive termination on CIFAR-10. (c) Effect of predictive termination for CIFAR-100 (total run-time depicted is $220,000$ seconds).

Large CNN Hyperparameters

To model these larger CNNs, we re-use the hyperparameter space from Table 3 and alter it in several key aspects. Firstly, following the recently proposed All-CNN network [Springenberg *et al.*, 2015], we replaced max-pooling with additional convolutional layers with stride two (each of these layers is also configurable using the convolutional layer hyperparameters from Table 3 (bottom)). Secondly, we no longer fixed the number of convolutional layers between dimensionality reduction (pooling) steps but made it an additional network hyperparameter with range $1-3$. Our large CNNs are thus considerably deeper than our small CNNs. We further allowed more units in each convolutional layer (between 64 to 192) and changed the kernel size to 3×3. The other notable difference to the small CNNs is that the output of the last dimensionality reduction layer is not fed directly to a softmax classifier but rather sent through an additional 3×3 and a final one-by-one convolutional layer followed by the softmax layer (whose output is averaged over the remaining spatial dimensions). This setup is in accordance with the network structure from Springenberg *et al.* [2015]. Overall, our large CNNs have many hyperparameters due to the additional convolutional layers and the newly parameterized pooling layers: $6(\text{network hyperparams}) + 3(\text{layer hyperparams}) \times [3(\text{reduction steps}) \times (3\text{conv. layers} + 1\text{reduction layer}) + 2\text{final layers}] = 48$ hyperparameters.

Results for Large CNNs on CIFAR-10

We tested our predictive termination criterion on the configuration space for large CNNs on CIFAR-10. Due to the large time costs attached to such an experiment—training a single configuration for the All-CNN on CIFAR-10 takes between 6 and 12 hours on a NVIDIA Titan GPU—we restricted ourselves to a single run of SMAC on this hyperparameter configuration space, using 4 GPUs in parallel. We trained all networks for a maximum of 800 epochs and evaluated 100 configurations. Table 5 compares the performance of the best network resulting from this experiment to the state-of-the-art for CIFAR-10 without data augmentation. The best model found by our approach performs comparably with the ALL-CNN results, slightly outperforming the best previously reported results. While the total runtime for this experiment with predictive termination was approximately 8 days on 4 GPUs, the optimization run without predictive termination would have taken more than 20 days on 4 GPUs.

CIFAR-10 classification error

Method	Error (%)
Maxout [Goodfellow *et al.*, 2013]	11.68%
Network in Network [Lin *et al.*, 2014]	10.41%
Deeply Supervised [Lee *et al.*, 2014]	9.69%
ALL-CNN [Springenberg *et al.*, 2015]	9.08%
ALL-CNN + SMAC with pred. termination	**8.81%**

Table 5: Test error on CIFAR-10 for the Large CNN in relation to the state-of-the-art without data augmentation.

5 Conclusion

We presented a method for speeding up the hyperparameter search for deep neural networks by automatically detecting and terminating underperforming hyperparameter evaluations. For this purpose, we introduced a probabilistic learning curve model that—like human experts—can extrapolate performance from only a few steps of stochastic gradient descent and terminate the training of models that are expected to yield poor performance. Our method is agnostic to the hyperparameter optimizer used, and in our experiments for optimizing various network architectures on several benchmarks it consistently sped up two state-of-the-art hyperparameter optimizers by a factor of roughly two, leading to state-of-the-art results on the CIFAR-10 dataset without data augmentation. The code for our learning curve prediction models and its integration into the CAFFE framework is publicly available at https://github.com/automl/pylearningcurvepredictor.

Acknowledgements

This work was supported by the German Research Foundation (DFG), under Priority Programme Autonomous Learning (SPP 1527, grant HU 1900/3-1) and under the BrainLinks-BrainTools Cluster of Excellence (grant number EXC 1086).

References

[Bengio, 2000] Y. Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8):1889–1900, 2000.

[Bergstra and Bengio, 2012] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13(1):281–305, 2012.

[Bergstra *et al.*, 2011] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proc. of NIPS*, pages 2546–2554, 2011.

[Bergstra *et al.*, 2013] J. Bergstra, D. Yamins, and D.D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proc. of ICML*, pages 115–123, 2013.

[Breiman, 2001] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Brochu *et al.*, 2010] E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010.

[Coates *et al.*, 2011] A. Coates, A. Y. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proc. of AISTATS*, pages 215–223, 2011.

[Dahl *et al.*, 2013] G. Dahl, T. Sainath, and G. Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Proc. of ICASSP*, pages 8609–8613. IEEE, 2013.

[Deng *et al.*, 2013] L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *Proc. of ICASSP*, 2013.

[Donahue *et al.*, 2014] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proc. of ICML*, 2014.

[Eggensperger *et al.*, 2013] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS Workshop on Bayesian Optimization in Theory and Practice (BayesOpt'13)*, 2013.

[Foreman-Mackey *et al.*, 2013] D. Foreman-Mackey, D. W. Hogg, D. Lang, and J. Goodman. emcee: The MCMC Hammer. *PASP*, 125:306–312, 2013.

[Frey and Fisher, 1999] L. Frey and D. Fisher. Modeling decision tree performance with the power law. In *Proc. of AISTATS*, 1999.

[Glorot and Bengio, 2010] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proc. of AISTATS*, pages 249–256, 2010.

[Goodfellow *et al.*, 2013] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *Proc. of ICML*, 2013.

[Gu *et al.*, 2001] B. Gu, F. Hu, and H. Liu. Modelling classification performance for large data sets. In *Proc. of WAIM*, pages 317–328. Springer, 2001.

[Hutter *et al.*, 2011] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION*, pages 507–523. Springer, 2011.

[Hutter *et al.*, 2014] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods and evaluation. *AIJ*, 206(0):79 – 111, 2014.

[Jarrett *et al.*, 2009] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *Proc. of ICCV*, 2009.

[Jia *et al.*, 2014] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[Jones *et al.*, 1998] D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[Kolachina *et al.*, 2012] P. Kolachina, N. Cancedda, M. Dymetman, and S. Venkatapathy. Prediction of learning curves in machine translation. In *Proc. of ACL*, pages 22–30, 2012.

[Krizhevsky *et al.*, 2012] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. of NIPS*, pages 1097–1105, 2012.

[Krizhevsky, 2009] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[LeCun *et al.*, 1989] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[Lee *et al.*, 2014] C. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply supervised nets. In *Deep Learning and Representation Learning Workshop, NIPS*, 2014.

[Lin *et al.*, 2014] M. Lin, Q. Chen, and S. Yan. Network in network. In *ICLR: Conference Track*, 2014.

[Maron and Moore, 1994] O. Maron and A. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Proc. of NIPS*, pages 59–66, 1994.

[Montavon *et al.*, 2012] G. Montavon, G. Orr, and K.-R. Müller, editors. *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *LNCS*. Springer, 2012.

[Rasmussen and Williams, 2006] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[Snoek *et al.*, 2012] J. Snoek, H. Larochelle, and R.P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NIPS*, pages 2951–2959, 2012.

[Springenberg *et al.*, 2015] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *arxiv:cs/arXiv:1412.6806*, 2015.

[Srivastava *et al.*, 2014] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 15:1929–1958, 2014.

[Swersky *et al.*, 2013] K. Swersky, D. Duvenaud, J. Snoek, F. Hutter, and M. Osborne. Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces. In *NIPS workshop on Bayesian Optimization in theory and practice (BayesOptâĂŹ13)*, 2013.

[Swersky *et al.*, 2014] K. Swersky, J. Snoek, and R. P. Adams. Freeze-thaw Bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[Yao *et al.*, 2007] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.