Towards Practical Neural Network Meta-Modeling

by

Bowen Baker

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

Author .....
Department of Electrical Engineering and Computer Science
August 18, 2017

Certified by .....
Cèsar Hidalgo
Associate Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Towards Practical Neural Network Meta-Modeling

by

Bowen Baker

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis primarily focuses on the topic of efficient automated procedures for convolutional neural network (CNN) architecture search.

We first introduce a novel approach for CNN architecture architecture using Q-learning, a popular value iteration algorithm from the reinforcement learning community for sequential decision problems. On the task of object classification, the Q-learning agent outperforms all human crafted models that are similar to those in the search space. By analysing the underlying weights of the agent, we are also able to uncover some of the design principles that the agent learned during the search process.

Reinforcement learning is generally very sample inefficient; in the case of architecture search most approaches require thousands of unique models to be trained. In the second part of this thesis we introduce simple sequential regression models (SRM) to predict final performance of a candidate CNN from partially observed learning curves. We use these performance predictors and empirical variance estimates for practical early stopping of online optimization procedures. Our SRMs are state-of-the-art in performance prediction and early stopping.

Thesis Supervisor: Cèsar Hidalgo
Title: Associate Professor

Acknowledgments

I would like to begin by thanking Nikhil Naik, Otkrist Gupta, and Ramesh Raskar for welcoming me into the Camera Culture group in the Media Lab and giving me the freedom explore open ended problems and ideas. Were it not for their support, contributions, and resources, much of this work would not have been possible.

I would also like to thank my parents for all of their love and support, and for providing me many opportunities to further my education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we introduce a reinforcement learning based architecture search method for deep convolutional neural networks. While automated architecture search methods have begun to see some success on both visual and text benchmarks [1, 2, 3], each method is extremely computationally expensive due to the need to repeatedly train many unique architectures. In an attempt to combat this deficiency, we also introduce a practical early stopping algorithm based on performance prediction from partially observed learning curves with simple sequential regression models and variance estimates.

1.1 Deep Learning

Though the study of neural networks is decades old, the recent groundbreaking results in object recognition [4], speech recognition [5], game playing [6], and many other domains have brought the study of deep neural networks (DNN) to the forefront of academic discussion. These deep models are composed by stacking together non-linear functions in a hierarchical manner, which leads to the ability for increasing levels of feature abstraction. Since the deep learning boom, many popular models have surfaced from the ILSVRC image competitions, e.g. AlexNet [4], Inception [7], Residual Networks [8], etc.

While these models have proven extremely powerful and general feature extrac-

tors, they cannot perform well for every problem specification. For instance, in high frame-rate real-time applications with limited hardware or in domains with drastically different input dimensionality these networks cannot be used 'out of the bag.' A typical CNN architecture consists of several convolution, pooling, and fully connected layers. While constructing a CNN, a network designer has to make numerous design choices: the number of layers of each type, the ordering of layers, and the hyperparameters for each type of layer, e.g., the receptive field size, stride, and number of receptive fields for a convolution layer. The number of possible choices makes the design space of CNN architectures extremely large and hence, infeasible for an exhaustive manual search. There are thus many cases where researchers or engineers will not have the time, manpower, or expertise to design a network architecture from scratch specifically for their application. This thesis seeks to introduce a method using reinforcement learning to take the human out of the loop in architecture selection and automatically craft architectures for specific tasks using reinforcement learning.

1.2 Reinforcement Learning

Reinforcement learning is a branch of machine learning concerned with optimizing an agent's sequence of actions such that it maximizes its total future reward in an environment through trial and error. Recently there has been much work at the intersection of reinforcement learning and deep learning. For instance, methods using convolutional neural networks (CNNs) to approximate the Q-learning utility function [9] have been successful in game-playing agents [6, 10] and robotic control [11, 12]. These methods rely on phases of *exploration*, where the agent tries to learn about its environment through sampling, and *exploitation*, where the agent uses what it learned about the environment to find better paths. In traditional reinforcement learning settings, over-exploration can lead to slow convergence times, yet over-exploitation can lead to convergence to local minima [13]. However, in the case of large or continuous state spaces, the *ϵ-greedy strategy* of learning has been empirically shown to converge [14]. Finally, when the state space is large or exploration is costly, the

experience replay technique [15] has proved useful in experimental settings [16, 6]. We incorporate these techniques—Q-learning, the ϵ-greedy strategy and experience replay—in our algorithm design.

1.3 Hyperparameter Optimization

In the context of neural networks, 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, usually through the space of filter types and sizes. A variety of Bayesian optimization methods have been proposed for hyperparameter optimization, including methods based on sequential model-based optimization (SMAC) [17], Gaussian processes (GP) [18], and TPE [19]. To improve on the scalability of Bayesian methods, Snoek et al. [20] utilize neural networks to efficiently model distributions over functions. However, random search or grid search [21] is most commonly used in practical settings. Recently, Li et al. [22] introduced Hyperband, a multi-armed bandit-based efficient random search technique that outperforms state-of-the-art Bayesian optimization methods.

1.4 Meta-Modeling

In the context of neural networks, we define meta-modeling as an algorithmic approach for designing network architectures from scratch. The earliest meta-modeling approaches for neural net design were based on genetic algorithms [23, 24, 25]. Suganuma et al. [26] use Cartesian genetic programming to obtain competitive results on image classification tasks. Saxena and Verbeek [27] use densely connected networks of layers to search for well-performing networks. Another popular tool for meta-modeling is Bayesian optimization [28]. Saliently, Bergstra et al. [19] utilize Tree of Parzen Estimators (TPE) to design feed-forward networks.

More recently, there have been experiments on large scale architecture search.

Published concurrently to our work, Zoph et al. [2] describe using an LSTM controller trained with policy gradients to design CNNs and recurrent cell architectures. For their CNN experiment, they constrain the controller to predict a *fixed* number of hyperparameters and furthermore constrain the controller to select layer hyperparameters in a fixed scaffold, i.e. a set ordering of layer types. For the recurrent cell experiment they similarly fix the number of hyperparameters and scaffold to a tree-like topology. Among others, these constraints allow them to find extremely competitive architectures. In another recent work, Real et al. [3] use a tournament of pairs genetic algorithm to jointly select CNN architecture and optimization hyperparameters. Their formulation has the benefit that no further hyperparameter tuning is required after the final architecture is selected.

While these methods have had success on small benchmark problems such as CIFAR-10 and Penn Treebank, they use between 2500 and 10000 GPU-days to converge. While GPU-days is an informal metric and these numbers are rough, the staggering order of magnitude in runtime makes it doubtful that these methods applicable to domains larger than these toy benchmarks.

1.5 Early Stopping

Early stopping is normally thought of as a type of model regularization, and in some cases can be shown to be equivalent to the standard Tikhonov regularization. In this thesis we do not use early stopping as a regularization technique but rather as a method to accelerate meta-modeling and hyperparameter optimization algorithms. Most hyperparameter optimization and meta-modeling methods suffer computationally because they must train many sub-par configurations. While humans are relatively good at terminating these configurations early on in training, automated methods ignorantly complete the entire optimization schedule. Hyperband [22], for instance, terminates models based on performance comparisons at regular intervals along the training schedule; however, for most methods, completing the entire training schedule is necessary because they use the final performance to update some form

of acquisition function which is used to select the next configuration for consideration. In order to perform automatic early stopping, one must therefore estimate the final performance as well as determine a termination point.

1.6 Neural Network Performance Prediction

There has been limited work on predicting neural network performance during the training process [29, 30]. Domhan et al. [29] introduce a weighted probabilistic model for learning curves utilizing a hand-selected set of basis functions and use this model for speeding up hyperparameter search in small convolutional neural networks (CNNs) and fully-connected networks (FCNs). Their experimental setup contains large search spaces for hyperparameters such as learning rate, batch size, and weight decay, along with limited search spaces for number of layers and number of units. In contrast to our work, this method does not utilize the learning curve information from different neural architectures; it independently models the learning curve for each network. We also note that Swersky et al. [31] develop a Gaussian Process-based method for predicting individual learning curves for logistic regression models (among others), but not for neural networks. Building on Domhan et al. [29], Klein et al. [30] train Bayesian neural networks for predicting unobserved learning curves using a training set of fully and partially observed learning curves. Our method—based on support vector machines—is simpler, more efficient, and more accurate than Bayesian neural nets in a variety of settings, including meta-modeling and hyperparameter optimization. We summarize the related work on these topics next.

1.7 Thesis Contributions

CNN Architecture Search Using Q-Learning[1]: We propose a novel method for neural network architecture search using Q-learning. We show that using this method and a comparatively modest amount of computational resources, we can outperform

[1] An abridged version of this chapter appears in Baker et al. [1]

all prior hand-crafted models that are close to the architecture search space on many standard benchmark image classification datasets.

CNN Performance Prediction Using Simple Sequential Regression Models[2]: We show that simple sequential regression models such as support vector regression or ordinary least squares can vastly outperform many Bayesian models used in the literature in the task of optimization performance extrapolation from partially observed learning curves. We furthermore show that these simple parametric models can extrapolate learning trajectories of vastly different model architectures, which has heretofore not been well studied.

Practical Early Stopping for Meta-Modeling Procedures: Using the simple performance extrapolation models, we experiment with early termination of subpar architecture and hyperparameter configurations during an architecture search or hyperparameter optimization procedures. We show that using an extremely simple early stopping algorithm based around empirical variance estimates, we can increase the efficiency of these searches by factors of up to 6.

[2]An abridged version of this chapter appears in Baker et al. [32]

Chapter 2

Background

Here we present a brief background to Markov Decision Processes, optimal control, policy gradients, and Q-learning.

2.1 Markov Decision Processes and Optimal Control

A Markov Decision Process (MDP) is a formalism used to describe an agent acting in a possible stochastic environment. An MDP is defined by

- $\mathcal{S}$ – the state space which describes all possible states the agent may be in.
- $\mathcal{U}$ – the action space which describes all possible actions from each state.
- $P(s, s'|u)$ – the set of transition probabilities between states given actions
- $r(s, u, s')$ – possibly stochastic reward given to the agent as a function of state, action, and next state.

In general, it is the goal of reinforcement learning and optimal control algorithms to maximize the total discounted cumulative future reward $R = \sum_{t=0}^{\infty} \gamma^t r(s_t, u_t, s_{t+1})$, where $\gamma \in (0, 1)$ is the discount factor on future rewards. The end goal is to come up with an optimal policy $\pi^*(s)$ which maximizes the total future reward. One way to

do this is to estimate the *value function*, defined as

$$J^*(s_i) = \max_{u \in \mathcal{U}(s_i)} \mathbb{E}_{s_j|s_i,u} \left[\mathbb{E}_{r|s_i,u,s_j}[r|s_i, u, s_j] + \gamma J^*(s_j) \right]. \tag{2.1}$$

An optimal policy given J^* is simply the greedy policy. If a model of the dynamics and reward are known (or estimated), one can estimate the true value function using value iteration. If the dynamics or reward function is unknown, then the value function can be estimated using *Q-learning*, which is further discussed in Section 2.2.

Alternatively, one can directly optimize the policy by solving

$$\max_{\pi} \mathbb{E}[R|\pi] \tag{2.2}$$

In large state-action space problems, it is necessary to use a parameterized policy, π_θ, which is often a large neural network. Now to learn the policy, we estimate the gradient of equation 2.2 with respect to θ. A particular π_θ induces a distribution over trajectories τ within the environment, and it is easy to show that

$$\nabla_\theta \mathbb{E}_\tau[R(\tau)|\pi_\theta] = \mathbb{E}_\tau[R(\tau) \nabla_\theta \log p(\tau)] \tag{2.3}$$

While we soley use Q-learning in this thesis, we thought it useful to review the policy gradient algorithm as it is used in other architecture search work [2].

2.2 Q-learning

Our method relies on Q-learning, a popular reinforcement learning algorithm. We now summarize the theoretical formulation of Q-learning, as adopted to our problem. Consider the task of teaching an agent to find optimal paths as a Markov Decision Process (MDP) in a finite-horizon environment. Constraining the environment to be finite-horizon ensures that the agent will deterministically terminate in a finite number of time steps. In addition, we restrict the environment to have a discrete and finite state space $\mathcal{S}$ as well as action space $\mathcal{U}$. For any state $s_i \in \mathcal{S}$, there is a finite

set of actions, $\mathcal{U}(s_i) \subseteq \mathcal{U}$, that the agent can choose from. For each episode, the agent is trying to optimize the total expected reward $R = \sum_{t=0}^{T} \gamma^t r(s_t, u_t, s_{t+1})$ where T is the maximum length of an episode.

For any state $s_i \in \mathcal{S}$ and subsequent action $u \in \mathcal{U}(s_i)$, we define the maximum total expected reward to be $Q^*(s_i, u)$. $Q^*(\cdot)$ is known as the *action-value* function and individual $Q^*(s_i, u)$ are know as *Q-values*. Noting that $J^*(s_i) = \max_{u \in \mathcal{U}(s_i)} Q(s_i, u)$, we may rewrite equation 2.1 as

$$Q^*(s_i, u) = \mathbb{E}_{s_j|s_i,u} \left[\mathbb{E}_{r|s_i,u,s_j}[r|s_i, u, s_j] + \gamma \max_{u' \in \mathcal{U}(s_j)} Q^*(s_j, u') \right]. \tag{2.4}$$

In many cases, it is impossible to analytically solve Bellman's Equation [33], but it can be formulated as an iterative update

$$Q_{t+1}(s_i, u) = (1 - \alpha) Q_t(s_i, u) + \alpha \left[r_t + \gamma \max_{u' \in \mathcal{U}(s_j)} Q_t(s_j, u') \right]. \tag{2.5}$$

Equation 2.5 is the simplest form of Q-learning proposed by [9]. For well formulated problems, $\lim_{t \to \infty} Q_t(s, u) = Q^*(s, u)$, as long as each transition is sampled infinitely many times [33]. The update equation has two parameters: (i) α is a *Q-learning rate* which determines the weight given to new information over old information, and (ii) γ is the *discount factor* which determines the weight given to short-term rewards over future rewards. The Q-learning algorithm is *model-free*, in that the learning agent can solve the task without ever explicitly constructing an estimate of environmental dynamics. In addition, Q-learning is *off policy*, meaning it can learn about optimal policies while exploring via a non-optimal behavioral distribution, i.e. the distribution by which the agent explores its environment.

We choose the behavior distribution using an ϵ-greedy strategy [6]. With this strategy, a random action is taken with probability ϵ and the greedy action, $\max_{u \in \mathcal{U}(s_i)} Q_t(s_i, u)$, is chosen with probability $1 - \epsilon$. We anneal ϵ from $1 \to 0$ such that the agent begins in an *exploration* phase and slowly starts moving towards the *exploitation* phase. In addition, when the exploration cost is large (which is true for our problem setting), it is beneficial to use the *experience replay* technique for faster convergence [34]. In

experience replay, the learning agent is provided with a memory of its past explored paths and rewards. At a given interval, the agent samples from the memory and updates its Q-values via Equation 2.5.

Chapter 3

Designing Neural Networks Using Q-Learning

The majority of this chapter is presented in Baker et al. [1]; however, we also include further results and model architecture embedding visualizations. We seek to automate the process of CNN architecture selection through a meta-modeling procedure based on reinforcement learning. We construct a novel Q-learning agent whose goal is to discover CNN architectures that perform well on a given machine learning task with no human intervention. The learning agent is given the task of sequentially picking layers of a CNN model. By discretizing and limiting the layer parameters to choose from, the agent is left with a finite but large space of model architectures to search from. The agent learns through random exploration and slowly begins to exploit its findings to select higher performing models using the ϵ-greedy strategy [6]. The agent receives the validation accuracy on the given machine learning task as the reward for selecting an architecture. We expedite the learning process through repeated memory sampling using experience replay [15]. We refer to this Q-learning based meta-modeling method as MetaQNN, which is summarized in Figure 3-1.[1]

We conduct experiments with a space of model architectures consisting of only standard convolution, pooling, and fully connected layers using three standard image classification datasets: CIFAR-10, SVHN, and MNIST. The learning agent discovers

[1]For more information, model files, and code, please visit *github.com/bowenbaker/metaqnn*

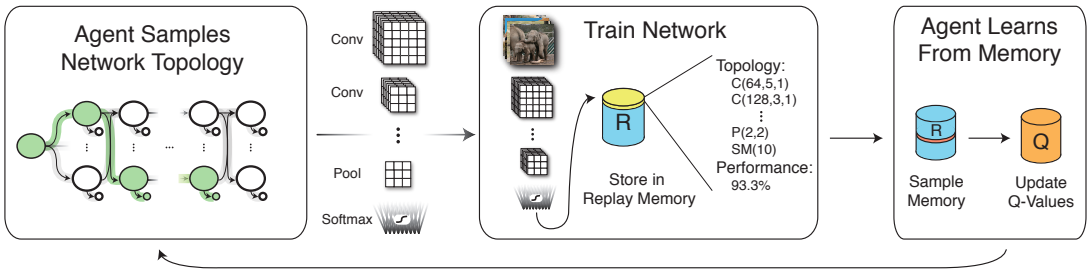


Figure 3-1: **Designing CNN Architectures with Q-learning:** The agent begins by sampling a Convolutional Neural Network (CNN) topology conditioned on a predefined behavior distribution and the agent's prior experience (left block). That CNN topology is then trained on a specific task; the topology description and performance, e.g. validation accuracy, are then stored in the agent's memory (middle block). Finally, the agent uses its memories to learn about the space of CNN topologies through Q-learning (right block).

CNN architectures that beat all existing networks designed only with the same layer types (e.g., [35, 36]). In addition, their performance is competitive against network designs that include complex layer types and training procedures (e.g., [37, 38]). Finally, the MetaQNN selected models comfortably outperform previous automated network design methods [24, 19]. The top network designs discovered by the agent on one dataset are also competitive when trained on other datasets, indicating that they are suited for transfer learning tasks. Moreover, we can generate not just one, but several varied, well-performing network designs, which can be ensembled to further boost the prediction performance.

3.1 Designing Neural Network Architectures with Q-learning

We consider the task of training a learning agent to sequentially choose neural network layers. Figure 3-2 shows feasible state and action spaces (a) and a potential trajectory the agent may take along with the CNN architecture defined by this trajectory (b). We model the layer selection process as a Markov Decision Process with the assumption that a well-performing layer in one network should also perform well

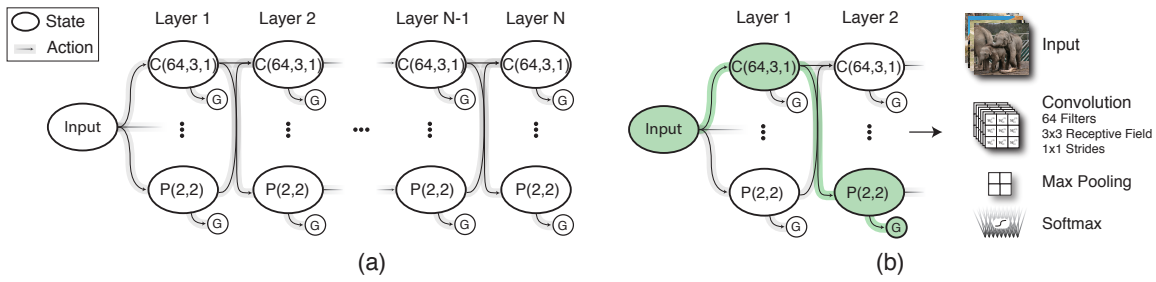


Figure 3-2: **Markov Decision Process for CNN Architecture Generation:** Figure 2(a) shows the full state and action space. In this illustration, actions are shown to be deterministic for clarity, but they are stochastic in experiments. $C(n, f, l)$ denotes a convolutional layer with n filters, receptive field size f, and stride l. $P(f, l)$ denotes a pooling layer with receptive field size f and stride l. G denotes a termination state (Softmax/Global Average Pooling). Figure 2(b) shows a path the agent may choose, highlighted in green, and the corresponding CNN topology.

in another network. We make this assumption based on the hierarchical nature of the feature representations learned by neural networks with many hidden layers [39]. The agent sequentially selects layers via the ϵ-greedy strategy until it reaches a termination state. The CNN architecture defined by the agent's path is trained on the chosen learning problem, and the agent is given a reward equal to the validation accuracy. The validation accuracy and architecture description are stored in a replay memory, and experiences are sampled periodically from the replay memory to update Q-values via Equation 2.5. The agent follows an ϵ schedule which determines its shift from exploration to exploitation.

Our method requires three main design choices: (i) reducing CNN layer definitions to simple state tuples, (ii) defining a set of actions the agent may take, i.e., the set of layers the agent may pick next given its current state, and (iii) balancing the size of the state-action space—and correspondingly, the model capacity—with the amount of exploration needed by the agent to converge. We now describe the design choices and the learning process in detail.

3.1.1 The State Space

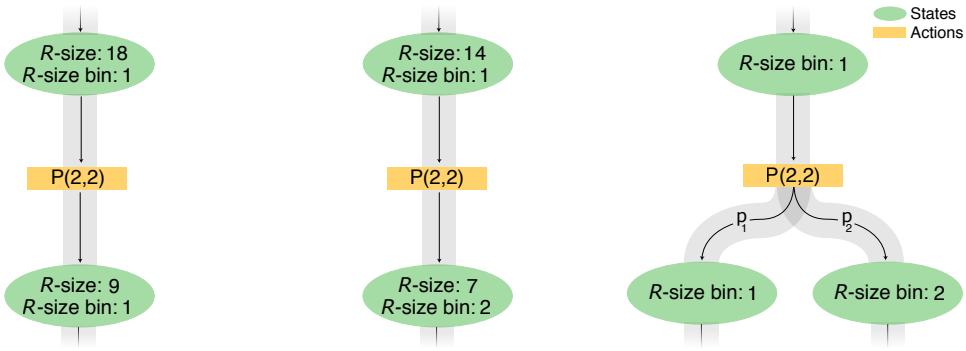
Each state is defined as a tuple of all relevant layer parameters. We allow five different types of layers: convolution (C), pooling (P), fully connected (FC), global average

pooling (GAP), and softmax (SM), though the general method is not limited to this set. Table 3.1 shows the relevant parameters for each layer type and also the discretization we chose for each parameter. Each layer has a parameter *layer depth* (shown as Layer $1, 2, ...$ in Figure 3-2). Adding *layer depth* to the state space allows us to constrict the action space such that the state-action graph is directed and acyclic (DAG) and also allows us to specify a maximum number of layers the agent may select before terminating.

Each layer type also has a parameter called *representation size* (R-size). Convolutional nets progressively compress the representation of the original signal through pooling and convolution. The presence of these layers in our state space may lead the agent on a trajectory where the intermediate signal representation gets reduced to a size that is too small for further processing. For example, five 2×2 pooling layers each with stride 2 will reduce an image of initial size 32×32 to size 1×1. At this stage, further pooling, or convolution with receptive field size greater than 1, would be meaningless and degenerate. To avoid such scenarios, we add the *R-size* parameter to the state tuple s, which allows us to restrict actions from states with R-size n to those that have a receptive field size less than or equal to n. To further constrict the state space, we chose to bin the representation sizes into three discrete buckets. However, binning adds uncertainty to the state transitions: depending on the true underlying representation size, a pooling layer may or may not change the R-size bin. As a result, the action of pooling can lead to two different states, which we model as stochasticity in state transitions. Please see Figure 3-3 for an illustrated example.

3.1.2 The Action Space

We restrict the agent from taking certain actions to both limit the state-action space and make learning tractable. First, we allow the agent to terminate a path at any point, i.e. it may choose a termination state from any non-termination state. In addition, we only allow transitions for a state with layer depth i to a state with layer depth $i + 1$, which ensures that there are no loops in the graph. This constraint



(a) (b) (c)

Figure 3-3: **Representation size binning:** In this figure, we show three example state transitions. The true *representation size* (R-size) parameter is included in the figure to show the true underlying state. Assuming there are two R-size bins, R-size Bin_1: $[8, \infty)$ and R-size Bin_2: $(0, 7]$, Figure 3-3a shows the case where the initial state is in R-size Bin_1 and true *representation size* is 18. After the agent chooses to pool with a 2×2 filter with stride 2, the true *representation size* reduces to 9 but the R-size bin does not change. In Figure 3-3b, the same 2×2 pooling layer with stride 2 reduces the actual *representation size* of 14 to 7, but the bin changes to R-size Bin_2. Therefore, in figures 3-3a and 3-3b, the agent ends up in different final states, despite originating in the same initial state and choosing the same action. Figure 3-3c shows that in our state-action space, when the agent takes an action that reduces the *representation size*, it will have uncertainty in which state it will transition to.

ensures that the state-action graph is always a DAG. Any state at the maximum layer depth, as prescribed in Table 3.1, may only transition to a termination layer.

Next, we limit the number of fully connected (FC) layers to be at maximum two, because a large number of FC layers can lead to too may learnable parameters. The agent at a state with type FC may transition to another state with type FC if and only if the number of consecutive FC states is less than the maximum allowed. Furthermore, a state s of type FC with number of neurons d may only transition to either a termination state or a state s' of type FC with number of neurons $d' \leq d$.

An agent at a state of type convolution (C) may transition to a state with any other layer type. An agent at a state with layer type pooling (P) may transition to a state with any other layer type other than another P state because consecutive pooling layers are equivalent to a single, larger pooling layer which could lie outside of our chosen state space. Furthermore, only states with representation size in bins $(8, 4]$

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 3.1: **Experimental State Space:** For each layer type, we list the relevant parameters and the values each parameter is allowed to take.

and $(4, 1]$ may transition to an FC layer, which ensures that the number of weights does not become unreasonably huge. Note that a majority of these constraints are in place to enable faster convergence on our limited hardware (see Section 3.2) and not a limitation of the method in itself.

3.1.3 Q-learning Training Procedure

For the iterative Q-learning updates (Equation 2.5), we set the Q-learning rate (α) to 0.01. In addition, we set the discount factor (γ) to 1 to not over-prioritize short-term rewards. We decrease ϵ from 1.0 to 0.1 in steps, where the step-size is defined by the number of *unique* models trained (Table 3.2). At $\epsilon = 1.0$, the agent samples CNN architecture with a random walk along a uniformly weighted Markov chain. Every topology sampled by the agent is trained using the procedure described in Section 3.2, and the prediction performance of this network topology on the validation set is recorded. We train a larger number of models at $\epsilon = 1.0$ as compared to other values of ϵ to ensure that the agent has adequate time to *explore* before it begins to *exploit*. We stop the agent at $\epsilon = 0.1$ (and not at $\epsilon = 0$) to obtain a stochastic final policy, which generates perturbations of the global minimum.[2] Ideally, we want to identify

[2] $\epsilon = 0$ indicates a completely deterministic policy. Because we would like to generate several good models for ensembling, we stop at $\epsilon = 0.1$, which represents a stochastic final policy.

ϵ	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
# Models Trained	1500	100	100	100	150	150	150	150	150	150

Table 3.2: **Experimental ϵ Schedule:** The learning agent trains the specified number of unique models at each ϵ.

several well-performing model topologies, which can then be ensembled to improve prediction performance.

During the entire training process (starting at $\epsilon = 1.0$), we maintain a *replay dictionary* which stores (i) the network topology and (ii) prediction performance on a validation set, for all of the sampled models. If a model that has already been trained is re-sampled, it is not re-trained, but instead the previously found validation accuracy is presented to the agent. After each model is sampled and trained, the agent randomly samples 100 models from the replay dictionary and applies the Q-value update defined in Equation 2.5 for all transitions in each sampled sequence. The Q-value update is applied to the transitions in temporally reversed order, which has been shown to speed up Q-values convergence [15].

3.2 Experiment Details

During the model exploration phase, we trained each network topology with a quick and aggressive training scheme. For each experiment, we created a validation set by randomly taking 5,000 samples from the training set such that the resulting class distributions were unchanged. For every network, a dropout layer was added after every two layers. The i^{th} dropout layer, out of a total n dropout layers, had a dropout probability of $\frac{i}{2n}$. Each model was trained for a total of 20 epochs with the Adam optimizer [40] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$. The batch size was set to 128, and the initial learning rate was set to 0.001. If the model failed to perform better than a random predictor after the first epoch, we reduced the learning rate by a factor of 0.4 and restarted training, for a maximum of 5 restarts. For models that started learning (i.e., performed better than a random predictor), we reduced the learning rate by a factor of 0.2 every 5 epochs. All weights were initialized with Xavier initialization [41].

Our experiments using Caffe [42] took 8-10 days to complete for each dataset with a hardware setup consisting of 10 NVIDIA GPUs.

After the agent completed the ϵ schedule (Table 3.2), we selected the top ten models that were found over the course of exploration. These models were then finetuned using a much longer training schedule, and only the top five were used for ensembling. We now provide details of the datasets and the finetuning process.

The **Street View House Numbers (SVHN)** dataset has 10 classes with a total of 73,257 samples in the original training set, 26,032 samples in the test set, and 531,131 additional samples in the extended training set. During the exploration phase, we only trained with the original training set, using 5,000 random samples as validation. We finetuned the top ten models with the original plus extended training set, by creating preprocessed training and validation sets as described by [38]. Our final learning rate schedule after tuning on validation set was 0.025 for 5 epochs, 0.0125 for 5 epochs, 0.0001 for 20 epochs, and 0.00001 for 10 epochs.

CIFAR-10, the 10 class tiny image dataset, has 50,000 training samples and 10,000 testing samples. During the exploration phase, we took 5,000 random samples from the training set for validation. The maximum layer depth was increased to 18. After the experiment completed, we used the same validation set to tune hyperparameters, resulting in a final training scheme which we ran on the entire training set. In the final training scheme, we set a learning rate of 0.025 for 40 epochs, 0.0125 for 40 epochs, 0.0001 for 160 epochs, and 0.00001 for 60 epochs, with all other parameters unchanged. During this phase, we preprocess using global contrast normalization and use moderate data augmentation, which consists of random mirroring and random translation by up to 5 pixels.

MNIST, the 10 class handwritten digits dataset, has 60,000 training samples and 10,000 testing samples. We preprocessed each image with global mean subtraction. In the final training scheme, we trained each model for 40 epochs and decreased learning rate every 5 epochs by a factor of 0.2.

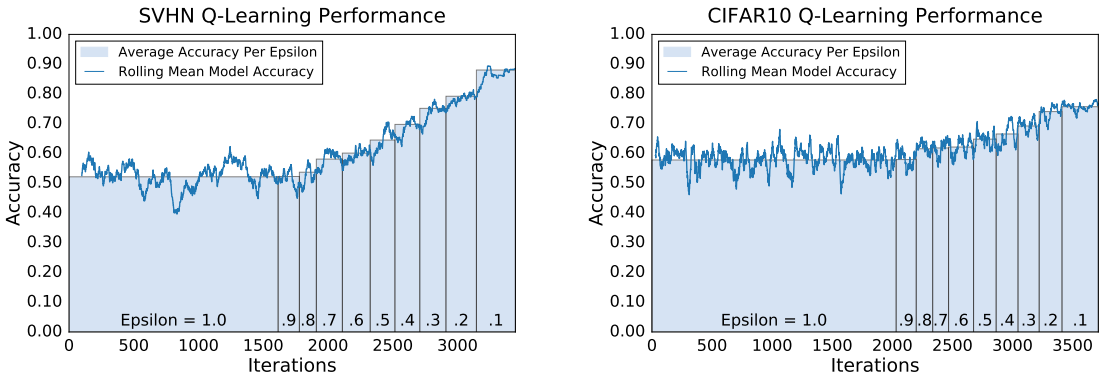


Figure 3-4: ***Q*-Learning Performance:** In the plots, the blue line shows a rolling mean of model accuracy versus iteration, where in each iteration of the algorithm the agent is sampling a model. Each bar (in light blue) marks the average accuracy over all models that were sampled during the exploration phase with the labeled ϵ. As ϵ decreases, the average accuracy goes up, demonstrating that the agent learns to select better-performing CNN architectures.

3.3 Experimental Results

3.3.1 Model Selection Analysis

From Q-learning principles, we expect the learning agent to improve in its ability to pick network topologies as ϵ reduces and the agent enters the exploitation phase. In Figure 3-4, we plot the rolling mean of prediction accuracy over 100 models and the mean accuracy of models sampled at different ϵ values, for the CIFAR-10 and SVHN experiments. The plots show that, while the prediction accuracy remains flat during the exploration phase ($\epsilon = 1$) as expected, the agent consistently improves in its ability to pick better-performing models as ϵ reduces from 1 to 0.1. For example, the mean accuracy of models in the SVHN experiment increases from 52.25% at $\epsilon = 1$ to 88.02% at $\epsilon = 0.1$. Furthermore, we demonstrate the stability of the Q-learning procedure with 10 independent runs on a subset of the SVHN dataset in Section 3.4.1 of the Appendix. Additional analysis of Q-learning results can be found in Section 3.4.2.

The top models selected by the Q-learning agent vary in the number of parameters but all demonstrate high performance (see Appendix Tables 1-3). For example, the number of parameters for the top five CIFAR-10 models range from 11.26 million to

Method	CIFAR-10	SVHN	MNIST	CIFAR-100
Maxout [45]	9.38	2.47	0.45	38.57
NIN [46]	8.81	2.35	0.47	35.68
FitNet [47]	8.39	2.42	0.51	35.04
HighWay [36]	7.72	-	-	-
VGGnet [48]	7.25	-	-	-
All-CNN [35]	7.25	-	-	33.71
MetaQNN (ensemble)	7.32	**2.06**	**0.32**	-
MetaQNN (top model)	**6.92**	2.28	0.44	**27.14***

Table 3.3: **Error Rate Comparison** with CNNs that only use convolution, pooling, and fully connected layers. We report results for CIFAR-10 and CIFAR-100 with moderate data augmentation and results for MNIST and SVHN without any data augmentation.

Method	CIFAR-10	SVHN	MNIST	CIFAR-100
DropConnect [49]	9.32	1.94	0.57	-
DSN [50]	8.22	1.92	0.39	34.57
R-CNN [51]	7.72	1.77	**0.31**	31.75
MetaQNN (ensemble)	7.32	2.06	0.32	-
MetaQNN (top model)	6.92	2.28	0.44	27.14*
Resnet(110) [52]	6.61	-	-	-
Resnet(1001) [53]	**4.62**	-	-	**22.71**
ELU [37]	6.55	-	-	24.28
Tree+Max-Avg [38]	6.05	**1.69**	**0.31**	32.37

Table 3.4: **Error Rate Comparison** with state-of-the-art methods with complex layer types. We report results for CIFAR-10 and CIFAR-100 with moderate data augmentation and results for MNIST and SVHN without any data augmentation. Please note that this comparison was created in December, 2016 and the benchmarks have shifted quite a bit since then.

1.10 million, with only a 2.32% decrease in test error. We find design motifs common to the top hand-crafted network architectures as well. For example, the agent often chooses a layer of type $C(N, 1, 1)$ as the first layer in the network. These layers generate N learnable linear transformations of the input data, which is similar in spirit to preprocessing of input data from RGB to a different color spaces such as YUV, as found in prior work [43, 44].

3.3.2 Top Model Performances

We compare the prediction performance of the MetaQNN networks discovered by the Q-learning agent with state-of-the-art methods on three datasets. We report the

accuracy of our best model, along with an ensemble of top five models. First, we compare MetaQNN with six existing architectures that are designed with standard convolution, pooling, and fully-connected layers alone, similar to our designs. As seen in Table 3.3, our top model alone, as well as the committee ensemble of five models, outperforms all similar models. Next, we compare our results with six top networks overall, which contain complex layer types and design ideas, including generalized pooling functions, residual connections, and recurrent modules. Our results are competitive with these methods as well (Table 3.4). Finally, our method outperforms existing automated network design methods. MetaQNN obtains an error of 6.92% as compared to 21.2% reported by [54] on CIFAR-10; and it obtains an error of 0.32% as compared to 7.9% reported by [25] on MNIST.

The difference in validation error between the top 10 models for MNIST was very small, so we also created an ensemble with all 10 models. This ensemble achieved a test error of **0.28%**—which beats the current state-of-the-art on MNIST without data augmentation.

The best CIFAR-10 model performs 1-2% better than the four next best models, which is why the ensemble accuracy is lower than the best model's accuracy. We posit that the CIFAR-10 MetaQNN did not have adequate exploration time given the larger state space compared to that of the SVHN experiment, causing it to not find more models with performance similar to the best model. Furthermore, the coarse training scheme could have been not as well suited for CIFAR-10 as it was for SVHN, causing some models to under perform.

3.3.3 Transfer Learning Ability

Network designs such as VGGnet [48] can be adopted to solve a variety of computer vision problems. To check if the MetaQNN networks provide similar transfer learning ability, we use the best MetaQNN model on the CIFAR-10 dataset for training other computer vision tasks. The model performs well (Table 3.5) both when training from random initializations, and finetuning from existing weights.

Dataset	CIFAR-100	SVHN	MNIST
Training from scratch	27.14	2.48	0.80
Finetuning	34.93	4.00	0.81
State-of-the-art	24.28 [37]	1.69 [38]	0.31 [38]

Table 3.5: **Transfer Learning Performance** for the top MetaQNN (CIFAR-10) model trained for other tasks. Finetuning refers to initializing training with the weights found for the optimal CIFAR-10 model.

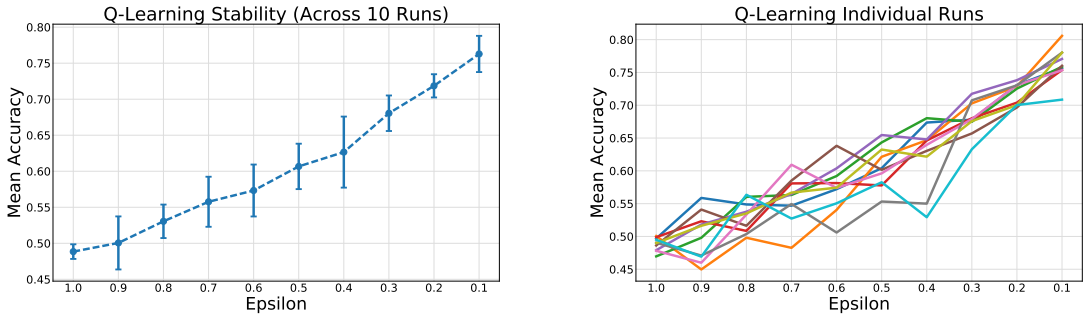
3.4 Further Analysis of Q-Learning

Figure 3-4 show that as the agent begins to exploit, it improves in architecture selection. It is also informative to look at the distribution of models chosen at each ϵ. Figure 3-5 gives further insight into the performance achieved at each ϵ for both experiments.

3.4.1 Q-Learning Stability

Because the Q-learning agent explores via a random or semi-random distribution, it is natural to ask whether the agent can consistently improve architecture performance. While the success of the three independent experiments described in the main text allude to stability, here we present further evidence. We conduct 10 independent runs of the Q-learning procedure on 10% of the SVHN dataset (which corresponds to $\sim$7,000 training examples). We use a smaller dataset to reduce the computation time of each independent run to 10GPU-days, as opposed to the 100GPU-days it would take on the full dataset. As can be seen in Figure 3-6, the Q-learning procedure with the exploration schedule detailed in Table 3.2 is fairly stable. The standard deviation at $\epsilon = 1$ is notably smaller than at other stages, which we attribute to the large difference in number of samples at each stage. Furthermore, the best model found during each run had remarkably similar performance with a mean accuracy of 88.25% and standard deviation of 0.58%, which shows that each run successfully found at least one very high performing model. Note that we did not use an extended training schedule to improve performance in this experiment.

*Results in this column obtained with the top MetaQNN architecture for CIFAR-10, trained from random initialization with CIFAR-100 data.



(a) (b)

Figure 3-6: **MetaQNN Stability:** Figure 3-6a shows the mean model accuracy and standard deviation at each ϵ over 10 independent runs of the Q-learning procedure on 10% of the SVHN dataset. Figure 3-6b shows the mean model accuracy at each ϵ for each independent experiment. Despite some variance due to a randomized exploration strategy, each independent run successfully improves architecture performance.

3.4.2 Q-Value Analysis

We now analyze the actual Q-values generated by the agent during the training process. The learning agent iteratively updates the Q-values of each path during the ϵ-greedy exploration. Each Q-value is initialized at 0.5. After the ϵ-schedule is complete, we can analyze the final Q-value associated with each path to gain insights into the layer selection process. In the left column of Figure 3-7, we plot the average Q-value for each layer type at different layer depths (for both SVHN and CIFAR-10) datasets. Roughly speaking, a higher Q-value associated with a layer type indicates a higher probability that the agent will pick that layer type. In Figure 3-7, we observe that, while the average Q-value is higher for convolution and pooling layers at lower layer depths, the Q-values for fully-connected and termination layers (softmax and global average pooling) increase as we go deeper into the network. This observation matches with traditional network designs.

We can also plot the average Q-values associated with different layer parameters for further analysis. In the right column of Figure 3-7, we plot the average Q-values for convolution layers with receptive field sizes 1, 3, and 5 at different layer depths. The plots show that layers with receptive field size of 5 have a higher Q-value as

compared to sizes 1 and 3 as we go deeper into the networks. This indicates that it might be beneficial to use larger receptive field sizes in deeper networks.

In summary, the Q-learning method enables us to perform analysis on the relative benefits of different design parameters of our state space, and possibly gain insights for new CNN designs.

3.5 All Models Aren't Created Equal

MetaQNN has a limitation that was not addressed in our original paper. All models do not have the same convergence rate, and thus comparing model performances after 20 epochs of training will bias the agent towards specific model depths (number of layers). Zoph et al. [2] somewhat sidestepped this issue by keeping fixed scaffolds for their networks, and Real et al. [3] jointly learn the optimization hyperparameters (including training schedule). Fixing a scaffold requires the user to choose the number of layers prior to the architecture search, which may be limiting in some cases. Jointly learning the optimization hyperparameters may be too costly, as it requires training some configurations for very long durations.

While our method did not have these protections baked in, it turns out the deeper models Q-learning agent picked were even more competitive the model with top performance at 20 epochs. The top model chosen by our Q-learning agent on the CIFAR-10 benchmark was a 9-layer model; to see if this model was the best chosen by our agent, we grouped the models based on number of layers and trained the top model from each group for the final 300 epoch training schedule. Out of these, the 15-layer model performed best achieving **94.7%** accuracy. The top 9-layer model achieved 84.78% at 20 epochs where the top 15-layer model only reached 81.2%, which demonstrates the need for a principled way to compare varying depth models. However, this result makes us more optimistic about using Q-learning for architecture search, as it was able to generate a simple feed forward architecture with no residual connections that outperforms everything in Table 3.4 except a 1001-layer residual network. Table 3.6 compares our updated results to two other meta-modeling approaches. Given we

Method	CIFAR-10 Error	# Architectures Sampled	Complexity Estimate (GPU-days)
MetaQNN (ours)	5.3	2,700	100
Neural Architecture Search [2]	3.65	12,800	10,000
Large Scale Evolution [3]	5.4	-	2,600

Table 3.6: **Comparison with meta-modeling methods**. We report the single model CIFAR-10 Error with minor data augmentation. We only give rough estimates of computation time so as to compare the order of magnitude. It is promising that MetaQNN can outperform Large Scale Evolution with more than an order of magnitude less computation and can come within 2% performance of Neural Architecture Search while using a factor of 100 less computation.

use order of magnitudes less computation than the competing experiments, it is quite promising that MetaQNN beats Large Scale Evolution [3] and comes within 2% of Neural Architecture Search [2].

Our sweep over the best varying depth models chosen by the agent gives much improved results, but it doesn't admit an immediate algorithm that allows the agent to automatically deal with this model depth bias. We leave further investigation of this issue to future work.

3.6 Visualizing The Architecture Space

It is odd that the Q-learning agent is able to learn despite the Markov assumption made, i.e. only the previous layer chosen is included in the state variable. Including variables such as *layer depth* and *representation size* in the state may give the agent enough partial information on the entire model chosen so far to make learning tractable, but we leave this ablative analysis for future work. Using the dataset of models sampled over the course of the agent's training period, we present some very preliminary findings on the space of architectures and why these simplistic assumptions are not detrimental to the architecture search. First, we define the *edit distance* between two models as the minimum number of inserts, deletes, and substitutions required to transform one model into the other. We only consider edits at the layer level, so substituting a convolutional layer with 3x3 receptive field for a convolutional layer with 1x1 receptive field is the same edit distance as swapping that convolutional

layer with any other layer type. Using this metric, we can create a distance matrix for the sampled architectures and embed them in a 2-dimensional space using t-SNE [55]. Figure 3-8 and 3-9 show these 2-dimensional embeddings, where each point is a a unique architecture, for the CIFAR-10 and SVHN experiments, respectively.

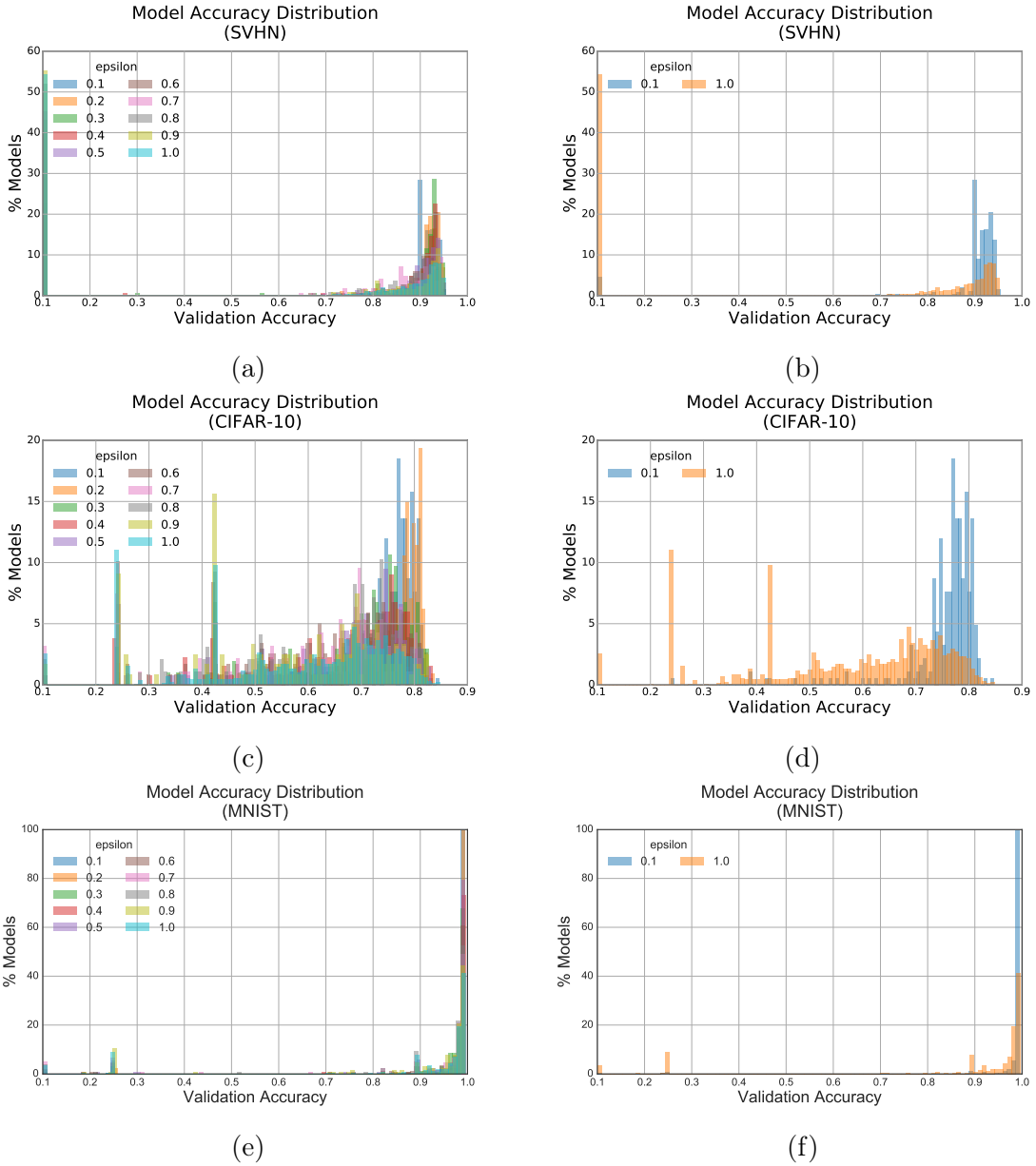


Figure 3-5: **Architecture Accuracy Distributions versus ϵ**: Figures 3-5a, 3-5c, and 3-5e show the accuracy distribution for each ϵ for the SVHN, CIFAR-10, and MNIST experiments, respectively. Figures 3-5b, 3-5d, and 3-5f show the accuracy distributions for the initial $\epsilon = 1$ and the final $\epsilon = 0.1$. One can see that the accuracy distribution becomes much more peaked in the high accuracy ranges at small ϵ for each experiment.

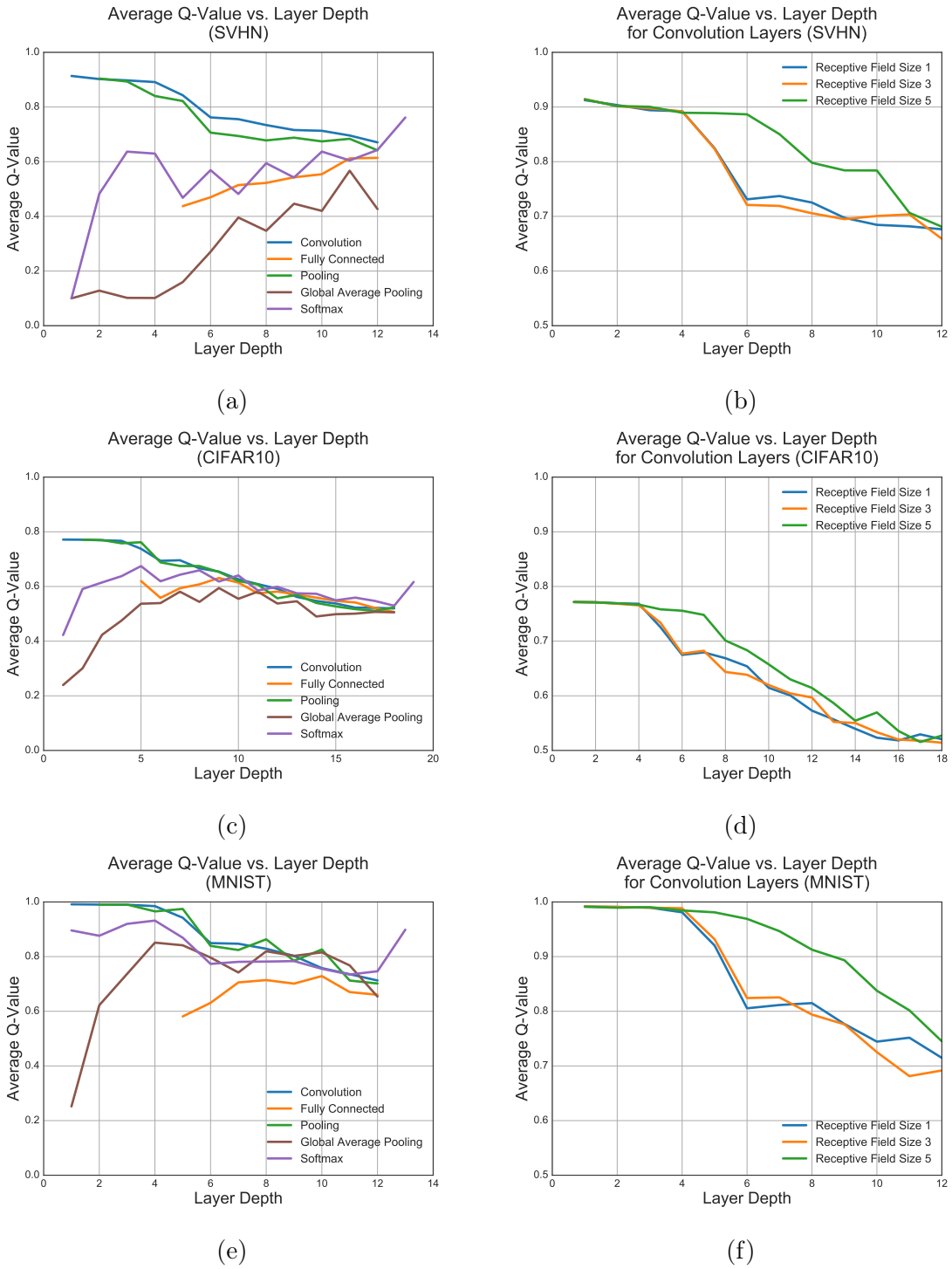


Figure 3-7: **Q-Value Statistics:** Average Q-Value versus Layer Depth for different layer types are shown in the left column. Average Q-Value versus Layer Depth for different receptive field sizes of the convolution layer are shown in the right column.

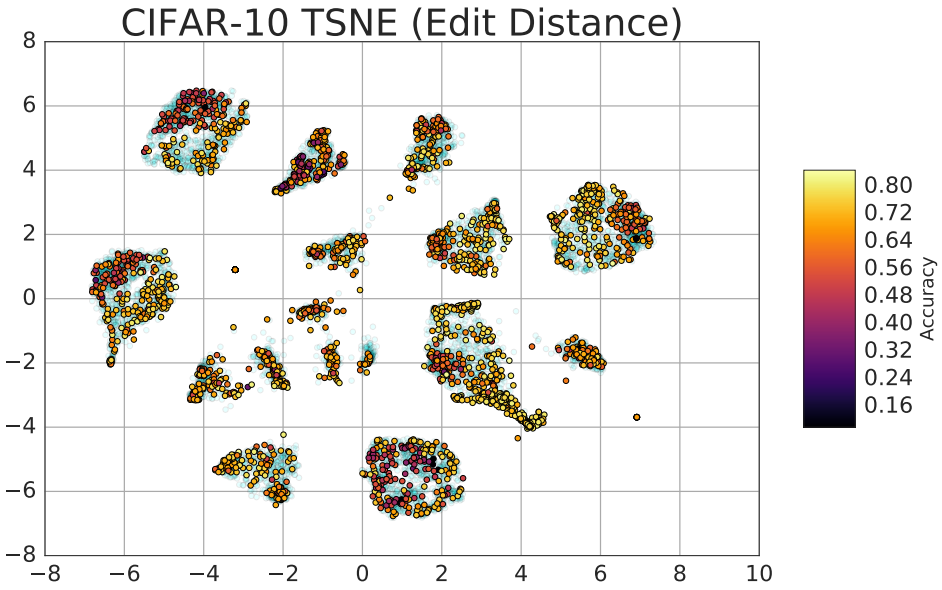


Figure 3-8: CIFAR-10 Architecture Edit Distance t-SNE

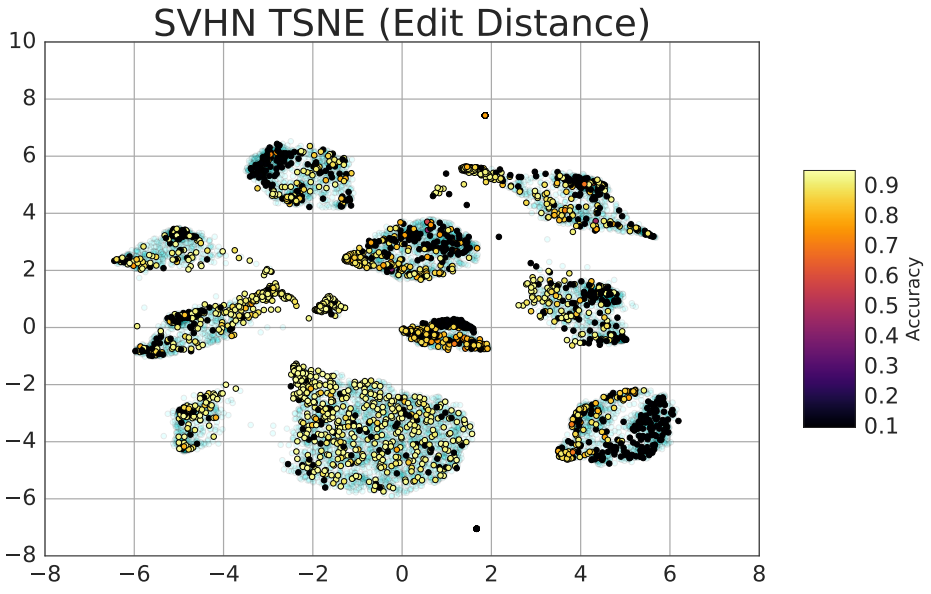


Figure 3-9: SVHN Architecture Edit Distance t-SNE

Chapter 4

Performance Prediction for Practical Early Stopping

Some of this chapter is presented in Baker et al. [32]; we also include a superior algorithm for speeding up Hyperband and further analysis. When sampling many different model configurations, it is likely that many sub-par configurations will be explored. Human experts are quite adept at recognizing and terminating suboptimal model configurations by inspecting their partially observed learning curves. In this chapter we seek to emulate this behavior and automatically identify and terminate sub-par model configurations in order to speedup both meta-modeling and hyperparameter optimization methods for deep neural networks. In Figure 4-1 we show the potential benefits of automated early termination for deep convolutional neural networks. Our method parameterizes learning curve trajectories using simple features derived from model architectures, training hyperparameters, and early time-series measurements from the learning curve. We demonstrate that a simple, fast, and accurate sequential regression model (SRM) can be trained to predict the final validation accuracy of partially observed neural network configurations using a small training set of fully observed curves. We can use these predictions and empirical variance estimates to construct a simple early stopping algorithm that can drastically speedup both meta-modeling and hyperparameter optimization methods.

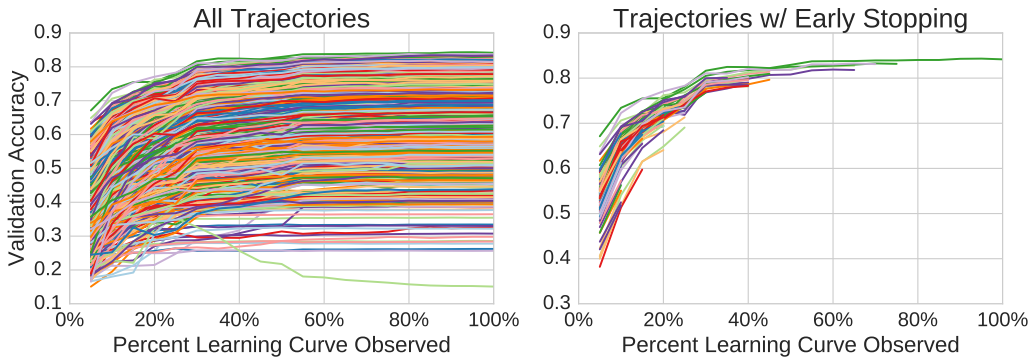


Figure 4-1: **Early Stopping Example:** (Left) 1000 learning curves sampled from the MetaQNN [1] search space. (Right) We see the same trajectories, many of which have been terminated by the early stopping algorithm presented in this work.

4.1 Method Overview

We first describe our model for neural network performance prediction, followed by a method for early termination of under-performing network architectures.

4.1.1 Modeling Learning Curves

Our goal is to model the validation accuracy $v(\mathbf{x}, t)$ of a neural net configuration $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$ at epoch $t \in \mathbb{R}$ using noisy observations $y(\mathbf{x}, t)$ drawn from an IID distribution. 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 $\mathcal{S} = \{(\mathbf{x}^1, y^1(t)), (\mathbf{x}^2, y^2(t)), \ldots, (\mathbf{x}^n, y^n(t))\}$. Figure 4-2 explicitly shows the partially observed learning curves with the target performances. Note that this problem formulation is very similar to Klein et al. [30].

We propose to use a set of features $u_{\mathbf{x}}$, derived from the neural net configuration $\mathbf{x}$, along with a subset of time-series accuracies $y(t)_{1-\tau} = (y_t)_{t=1,2,\ldots,\tau}$ (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 $x_f = \{u_{\mathbf{x}}, y(t)_{1-\tau}\}$. We utilize ν-Support Vector Regression (ν-SVR) [56] for training a model for y_T; however, we note that SVR did not perform significantly better than other simple and efficient models such as kernelized ordinary least squares. We denote this form of model as

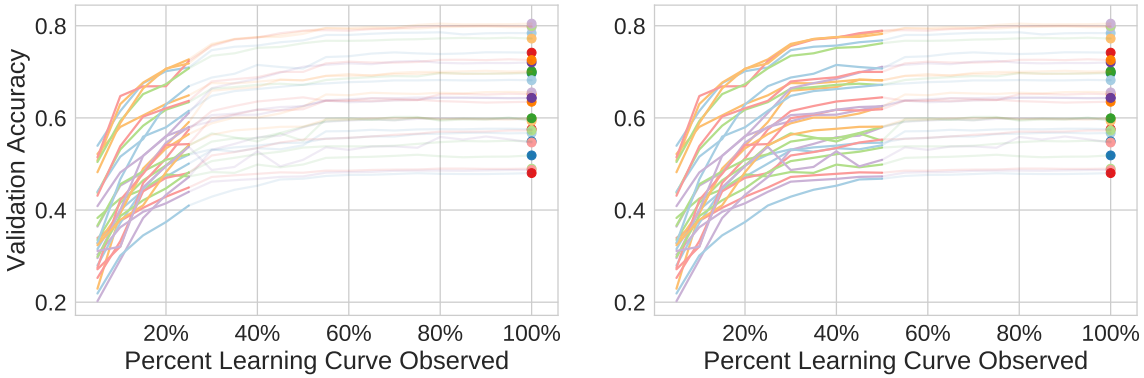


Figure 4-2: **Partial Learning Curve Training Data:** Here we show the partial learning curves (bold lines), and the target final accuracy (circles). Using our Sequential Regression Models, we would train a separate model for both the training set shown in the left and right plots.

a *sequential regression model* (SRM) as we have to train $T-1$ separate models to deal with the varying dimensionality of the partial time series. We note that this is still extremely computationally cheap both in absolute terms and when compared to competing methods as we will see in Section 4.2.4.

4.1.2 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 SRM. 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 $f(x_f)$ with an SRM 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 observed 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 case $f(x_f)$ has poor out-of-sample generalization, we may mistakingly terminate the optimal configuration. Thus, 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 $p(\hat{y}_T \leq y_{\text{BEST}} | \sigma(\mathbf{x}, \tau)) = \Phi(y_{\text{BEST}}; \hat{y}_T, \sigma)$, 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 don't 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. For complete clarity, we train $T-1$ independent performance predictors, the τ^{th} of which only uses the points $y(t)_{1-\tau}$ from the learning curve. This use of *sequential regression models* may seem unreasonably expensive; however, in practice each model is extremely cheap to train and far cheaper to do inference in than the previous state-of-the-art models, e.g. [29] and [30].

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.

Please note, the only difference between this work and the excellent work done by Domhan et al. [29] is our use of sequential regression models and empirical variance estimates. The early termination algorithm and use in SMBO is the same, except as we will see in the next section, our models are much cheaper to train and do inference in, more accurate, and simpler to implement.

Fast Hyperband

Here we present the algorithm for Fast Hyperband (f-Hyperband), which details exactly our method for early termination for improved performance on the Hyperband algorithm, which is detailed in Algorithm 1 from Li et al. [22]. Algorithm 1 of this text replicates Algorithm 1 from [22], 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 4.1.2 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.

The end goal of this procedure is to find the single best model configuration trained to R epochs. Thus, it makes sense to do global early stopping across successive halving blocks *only* for configurations that are being trained to R epochs. This essentially entails remembering the best model seen so far and running the same early stopping algorithm for other models training to R epochs, alleviating the need to train at least one model to R epochs in each iteration of successive halving. This can also be carried out over successive iterations of Hyperband. All results presented use both of these modifications.

4.2 Experiments and Results

We now evaluate the performance of our algorithm in three separate settings. First, we analyze the ability a ν-SVR model to predict the final validation accuracy of a

Algorithm 1: f-Hyperband

input : R – (Max resources allocated to any configuration)
η – (default $\eta = 3$)
Δ – (Probability threshold for early termination)
δ – (Performance offset for early termination)
d – (# points required to train performance predictors)
κ – (Proportion of models to train)

initialize: $D = \text{dict}()$
$M = \text{dict}()$
$s_{\max} = \lfloor \log_\eta(R) \rfloor$
$B = (s_{\max} + 1)R$

for $s \in \{s_{max}, \ldots, 0\}$ **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 $i \in \{0, \ldots, s\}$ **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 ∈ T do
    ℓ = []
    for i ∈ {0, ..., r − 1} do
        ℓ_i = run_one_epoch_return_validation_loss(t)
        ℓ.append(ℓ_i)
        if M[r][i].trained() then
            ŷ_r, σ = M[r][i].predict(ℓ)
            if Φ(max_k(L, κn) − δ; ŷ_r, σ) ≥ Δ then
                L.append(ŷ_r)
                break
            end
        end
        else if i == r − 1 then
            L.append(ℓ_i)
        end
    end
    if length(D[r][0]) < d and length(ℓ) == r then
        {D[r][i].append({ℓ[0, ..., i], ℓ[r]}): i ∈ {0, ..., r − 1}}
        if notM[r][i].trained() then
            M[r][i].train(D[r][i])
        end
    end
end
return L
```

trained neural network. Second, we integrate the SRM-based early stopping model into MetaQNN [1] to show that it can speed up the meta-modeling process without perturbing the reward function to the point that the agent learns suboptimal policies. Finally, we characterize the acceleration achieved by f-Hyperband over the vanilla Hyperband algorithm. We first describe the process we use to train our performance prediction model.

For all experiments, we train the ν-SVR model with random search over 1000 hyperparameter configurations from the space $C \sim \text{LogUniform}(10^{-5}, 10)$, $\nu \sim \text{Uniform}(0, 1)$, and $\gamma \sim \text{LogUniform}(10^{-5}, 10)$ (when using the RBF kernel). We use a combination of features to train the SVR. For all experiments described in this chapter, we use the following time-series features: (i) the validation accuracies $y'(t)_{1\text{–}\tau} = (y_t)_{t=1,2,\ldots,\tau}$ (where $1 \leq \tau < T$), (ii) the first-order differences of validation accuracies (i.e., $y_t{}' = (y_t - y_{t-1})$), and (iii) the second-order differences of validation accuracies (i.e., $y_t{}'' = (y_t{}' - y_{t-1}{}')'$). For experiments in which we vary the CNN architectures (MetaQNN CNNs and Deep Resnet), we include the total number of weights, number of layers, and learning rate into the feature space. For experiments in which we vary the optimization hyperparameters (Cuda-Convnet and AlexNet), we include all hyperparameters for training the neural networks as features.

4.2.1 Datasets and Training Procedures

We now describe the datasets and training procedures used in our experiments. We generate learning curves from randomly sampled models in four different hyperparameter search spaces. We experiment with standard datasets on both small convolutional networks and very deep convolutional architectures.

MetaQNN CNNs: We sample 1,000 model architectures from the search space detailed by Baker et al. [1], which allows for varying the numbers and orderings of convolution, pooling, and fully connected layers. We experiment with both the SVHN and CIFAR-10 datasets and use the same preprocessing and optimization hyperparameters.

Cuda-Convnet: We also experiment with the Cuda-Convnet architecture [57].

We vary 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 experiment with both the CIFAR-10 and SVHN datasets. CIFAR-10 models are trained for 60 epochs and SVHN models are trained for 12 epochs. A total of 8489 and 16582 configurations were randomly sampled for CIFAR-10 and SVHN respectively.

AlexNet: We train the AlexNet [4] model on the ILSVRC12 dataset. It was considerably more expensive to train many configurations on ILSVRC12 because of larger image and dataset size. To compensate for our limited computation resources, we randomly sample 10% of dataset, trained each configuration for 10 epochs, and only vary learning rate and learning rate reduction. We sampled and trained 1,376 hyperparameter configurations.

Deep Resnet: We sample 500 ResNet [52] architectures from a search space similar to Zoph et al. [2]. 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.

4.2.2 Prediction Performance

We now evaluate the ability of ν-SVR, trained with linear and RBF kernels, to predict the final performance of partially trained neural networks. We compare against the Bayesian Neural Network (BNN) presented by Klein et al. [30] using a Hamiltonian Monte Carlo sampler. 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

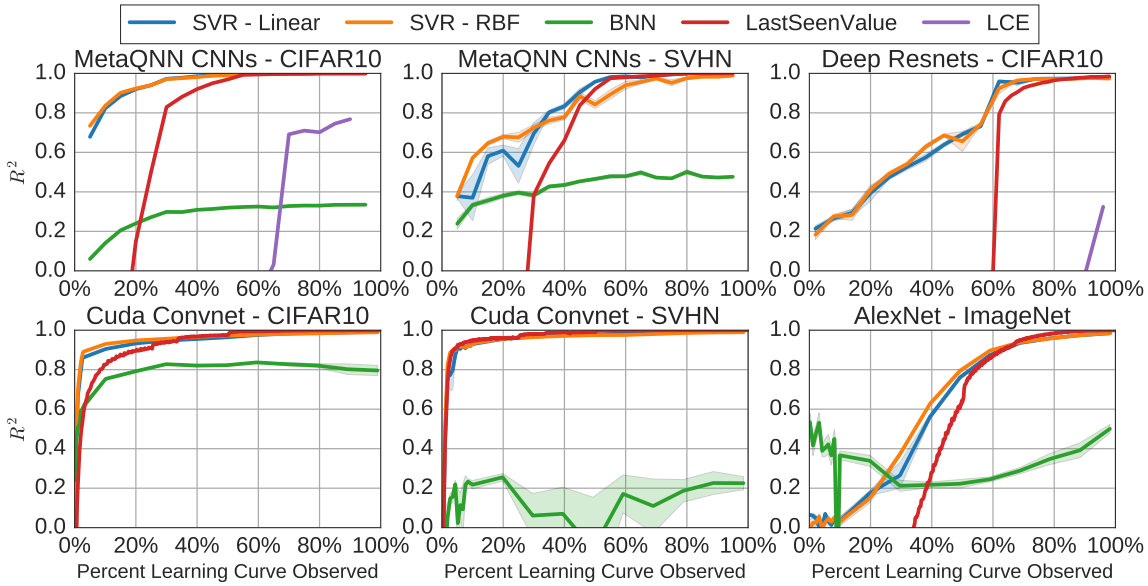


Figure 4-3: **Performance Prediction Results:** We plot the performance of each method versus the percent of learning curve observed. For BNN and ν-SVR (linear and RBF), we sample 10 different training sets, plot the mean R^2, and shade the corresponding standard error. The top row shows results on problems varying the CNN architecture, and the bottom row shows results on problems varying optimization hyperparameters. We compare our method against BNN [30], LCE [29], and a "last seen value" heuristic [22]. Absent results for a model indicate that it did not achieve a positive R^2.

from the training set. While we do not present the partially observed curves to the ν-SVR model for training, we felt this was a fair comparison as ν-SVR uses the entire partially observed learning curve during inference. Furthermore, we compare against the learning curve extrapolation (LCE) method introduced by Domhan et al. [29] and the last seen value (LastSeenValue) heuristic [22], both of which don't incorporate prior learning curves during training. For all experiments, we obtain a training set of 100 neural net configurations randomly sampled from a dataset. We obtain the best performing ν-SVR using random hyperparameter search over 3-fold cross-validation on this training set. We then compute the regression performance over the remaining points in the dataset. We repeat this experiment 10 times and report the results with standard errors in Figure 4-3. We also compare the performance versus the number of training samples (i.e. the number of observed learning curves) in Figure B-1.

Figure 4-3 shows the Coefficient of Determination (R^2) obtained by each method

Feature Set	MetaQNN CIFAR-10 (R^2)	Cuda-Convnet CIFAR10 (R^2)
TS	0.9656	0.9680
TS + AP	0.9703	–
TS + AP + HP	0.9704	0.9473
TS + AP + HP + BoW	0.9625	–

Table 4.1: Ablation Analysis for Feature Importance in Prediction Performance: time-series data (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, hyperparameters (HP) refer to the optimization parameters such as learning rate, and Bag-of-Words (BOW) refers counts of each layer type in a deep CNN. We use TS + AP + HP in all other experiments described in the chapter.

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, both ν-SVR models drastically outperform the competing methods. In fact, ν-SVR achieves $R^2 > 0.8$ on three out of six experiments with only 10% of the learning curve observed. For deeper models (Resnets and Alexnet), ν-SVR obtains $R^2 > 0.6$ after observing only 40% of the learning curve. In comparison, we find that the LastSeenValue heuristic only becomes viable when the models are near convergence, and its performance is much worse than ν-SVR for very deep models. We also find that the LCE model does poorly at the prediction task in all experiments. BNN also has relatively mediocre performance, but tends to do better than LastSeenValue and LCE when only a few iterations have been observed.

To further demonstrate the remarkable fit obtained by the ν-SVR model, we show the predicted versus true values of final validation accuracy for the MetaQNN, Cuda-Convnet, and Resnet search spaces on the CIFAR-10 dataset in Figure 4-4. Each plot is generated using ν-SVR with a Gaussian kernel, using 25% the learning curve as training data, along with the features obtained from the architecture, and hyperparameters. We also compared RBF kernel ordinary least squares and found it to perform almost as well as ν-SVR, further demonstrating the advantage of simple regression models over Bayesian methods for this task. Finally, we analyze which features in our model are the most informative in Table 4.1.

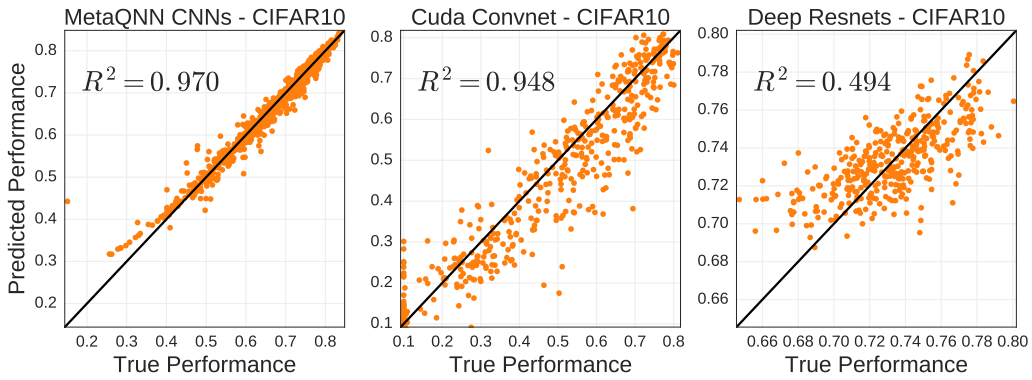


Figure 4-4: **Predicted vs True Values of Final Performance:** We show the shape of the predictive distribution for three network configuration datasets trained on CIFAR-10. Each ν-SVR model is trained with 100 configurations with data from 25% of the learning curve.

4.2.3 Early Stopping for Meta-modeling

We now detail the performance of ν-SVR in speeding up architecture search using sequential configuration selection. 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 the algorithms presented in Section 4.1.2. We compare against the early stopping algorithm proposed by Domhan et al. [29] as a baseline, which has a similar probability threshold termination criterion. The ν-SVR model 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 the ν-SVR model, it is still able to outperform the LCE model quite drastically, as can be seen in Figure 4-5. The plotted results are only in terms of training the CNNs; however, in our experience it takes anywhere from 1 to 3 minutes to fit the LCE model to a learning curve on a modern CPU due to expensive MCMC sampling. Additionally, each time a new point on the learning curve is observed, a new LCE model must be fit. Thus, for an experiment on the order of that of [1, 2], many days of computation time would be added as overhead to fit the LCE models when compared to our simple model.

We furthermore simulate early stopping for the deep resnets search space. In our experiment we found that only the probability threshold $\Delta = 0.99$ resulted in recov-

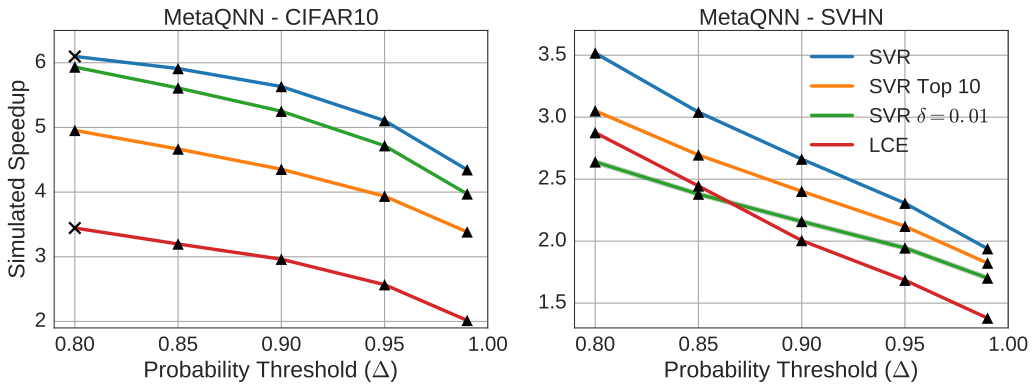


Figure 4-5: **Simulated Speedup in MetaQNN Search Space:** We compare the three variants of the early stopping algorithm presented in Section 4.1.2. Each ν-SVR model is trained using the first 100 learning curves, and each algorithm is tested on 10 independent orderings of the model configurations. Triangles 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.

ering 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—even with this speedup—to run the full experiment from Zoph et al. [2], we believe that this is a promising result for future large scale architecture searches.

It is not enough, however, to simply simulate the speedup because sequential configuration selection algorithms typically use the observed performance in order to update some type of acquisition function. 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 sub-par policies. We replicate the MetaQNN experiment on the CIFAR-10 dataset (Figure 4-6). We find that integrating early stopping with the Q-learning procedure does not disrupt learning and resulted in a speedup of 3.8x; note that for this experiment we set $\Delta = 0.99$ which explains why the speedup is relatively low. After training the top models to 300 epochs, we also find that the resulting performance is on par with the originally published numbers without early stopping (just under 93%).

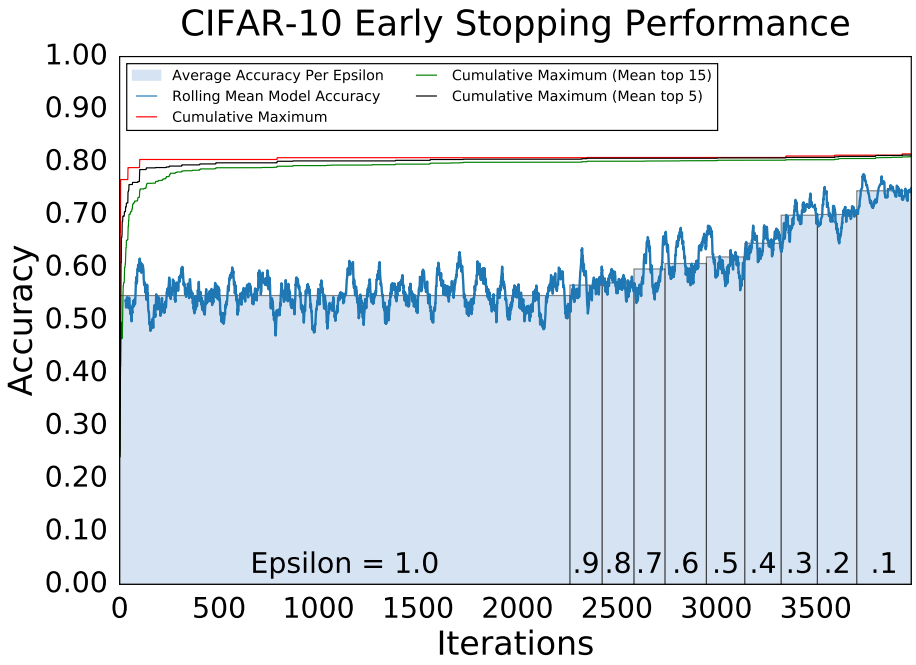


Figure 4-6: **MetaQNN on CIFAR-10 with Early Stopping:** A full run of the MetaQNN algorithm [1] on the CIFAR-10 dataset with early stopping. We use the ν-SVR model with a probability threshold $\Delta = 0.99$. Light blue bars indicate the average model accuracy per decrease in ϵ, which represents the 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.

4.2.4 Early Stopping for Hyperparameter Optimization

For our final empirical test, we turn towards the task of searching over optimization hyperparameters, such as learning rate, regularization weight, etc. In this section we present results on the Hyperband algorithm [22]. Within each block of successive halving, we found that discarding model configurations based on predicted final performance led to degraded performance due to noise in the predictions. Accordingly, we predict the final accuracy *within each block*, i.e. if in the current block of successive halving we are training each model in the population to r iterations, then we train models to predict y_r instead of y_T. Not only does this reduce the variance of our predictions, but it also allows us to begin early stopping within initial blocks of successive halving sooner. Figure 4-7 shows that our early stopping algorithm evaluates the same number of unique configurations as Hyperband 2 to 3.5 times faster, while often coming within standard error of the vanilla Hyperband performance. For all

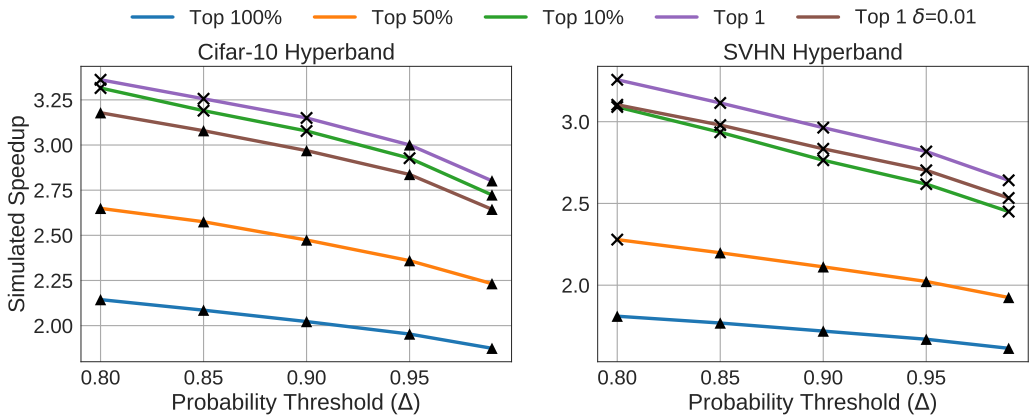


Figure 4-7: **Simulated Speedup on Hyperband with Earlier Stopping:** Both experiments show hyperparameter search over the optimization parameters of Cuda-Convnet. We repeat each experiment 10 times and plot the mean and standard error. Triangles indicate an algorithm that recovered a model within standard error of Hyperband without early stopping, and X's indicate those that did not. Note that the standard error for these problems is less than 0.1% on the validation set. Top $X\%$ indicate algorithms that try to recover the top $X\%$ of where X is the number of models considered in the next block of successive halving.

results, we make sure that the configuration sampling seeds are kept the same.

Figure 4-8 shows the max validation performance versus total amount of computation used over 40 consecutive Hyperband runs. Because in the early Hyperbands we do not yet have enough training data to create the performance predictors, we plot the speedup over vanilla Hyperband as a function of the number of Hyperbands run in Figure 4-9.

Training and Inference Speed

We have empirically shown the power of sequential regression models (SRM). However, it is also a concern how efficient SRMs are to train and do inference in as they must be run in parallel with a meta-modeling procedure. Empirically we found *nu*-SVR to take around 0.006s to train on 100 datapoints on a single core of an Intel 6700k CPU, and around 5×10^{-5}s to do inference. Even if we measure the performance of each configuration 1,000 times, it would only take 20 minutes to do a 1,000 hyperparameter random search for each of the 1,000 regression models on a modern

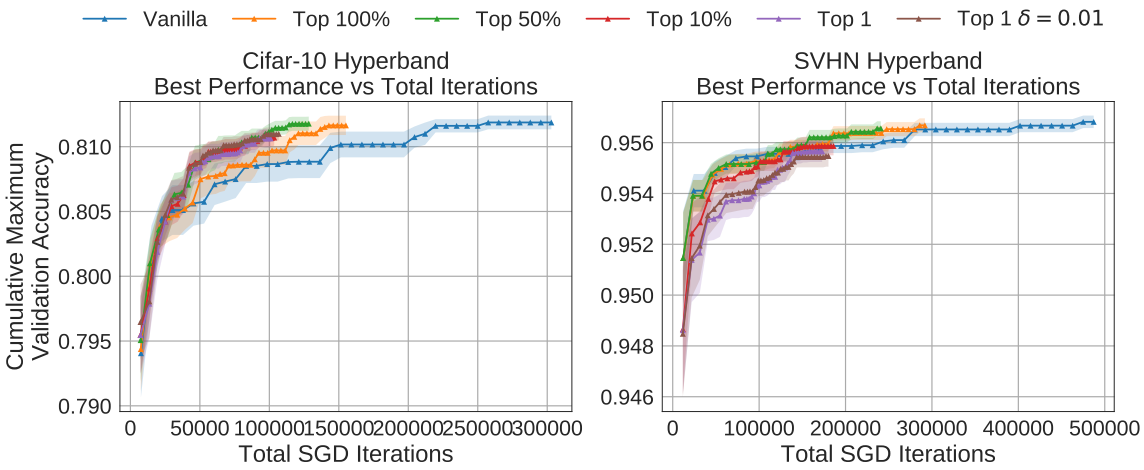


Figure 4-8: **Simulated Max Accuracy vs SGD Iterations for Hyperband:** We show the trajectories of the maximum performance so far versus total computational resources used for 40 consecutive Hyperband runs with $\eta = 3.0$ and $\Delta = 0.95$. As can be seen in the CIFAR-10 experiment (left), our method remains above the vanilla 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.

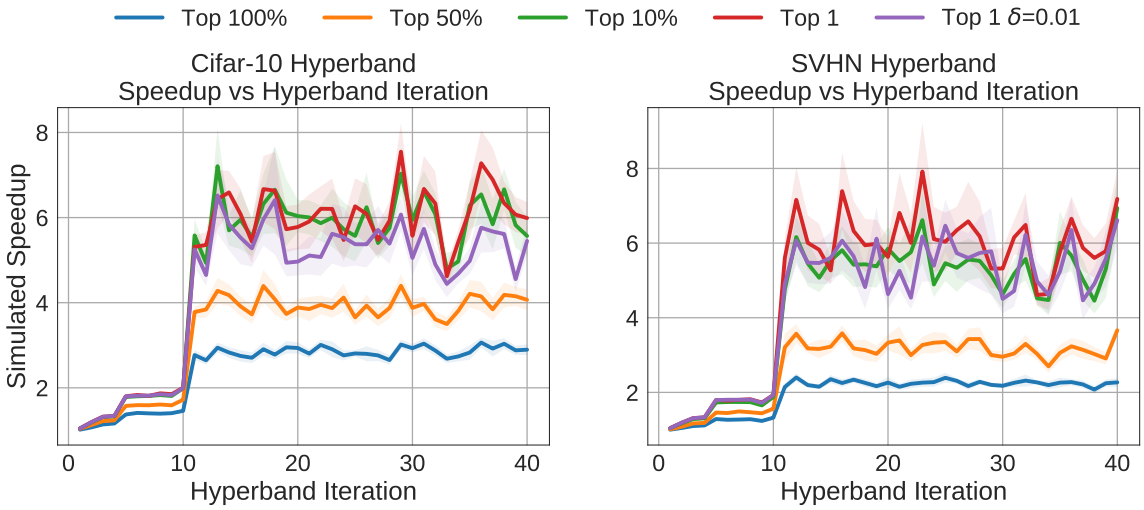


Figure 4-9: **Simulated Speedup on Hyperband vs Hyperband Iteration:** We show the speedup using the f-Hyperband algorithm over vanilla Hyperband on 40 consecutive Hyperband runs with $\eta = 3.0$ and $\Delta = 0.95$. The major jump in speedup comes at iteration 10, where we have trained more than 100 models to the full R iterations.

8 core CPU. In comparison, the open source code from Domhan et al. [29]* takes 60s on the same CPU for *each* inference, which will need to do *each* time we query the model on whether to terminate training for early stopping. Of course, for training deep networks on GPUs, these inferences can happen in parallel. Even in this case, the method from [29] would waste around 8.8 GPU-Days on an experiment the scale of that of Zoph et al. [2], whereas our method would waste less than GPU-Second.

*Code found at https://github.com/automl/pylearningcurvepredictor

Chapter 5

Conclusion and Future Directions

Neural networks are being used in an increasingly wide variety of domains, which calls for scalable solutions to produce problem-specific model architectures. We take a step towards this goal and show that a meta-modeling approach using reinforcement learning is able to generate tailored CNN designs for different image classification tasks. We introduced a reinforcement learning based approach to architecture search, MetaQNN, which generates networks networks outperform hand-crafted networks which use the same types of layers.

Despite the success of this and other architecture search methods, they are all prohibitively expensive except for small scale problems, e.g. CIFAR-10, or for institutions with inordinate amounts of computational resources. In an effort to bring the cost down, 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. Using our simple algorithm, we can speedup hyperparameter search techniques with complex acquisition functions, such as a Q-learning agent, by a factor of 3-6x and Hyperband—a state-of-the-art hyperparameter search method—by a factor of 2-3x, without disturbing the search procedure.

Future Directions: While we report results for image classification problems, our method could be applied to different problem settings, including supervised (e.g., classification, regression) and unsupervised (e.g., autoencoders). The MetaQNN

method could also aid constraint-based network design, by optimizing parameters such as size, speed, and accuracy. For instance, one could add a threshold in the state-action space barring the agent from creating models larger than the desired limit. In addition, one could modify the reward function to penalize large models for constraining memory or penalize slow forward passes to incentivize quick inference. This will become increasingly important as more and more mobile and embedded system applications require fast light-weight architectures.

Section 3.5 presents the issue of intersecting training curves. There is currently no solution other than to restrict the search to a class of models that should have similar learning rates, or to jointly learn the training schedule and the architecture. Both have their limitations; the first being that the deep learning engineer must create a more constrained search space, and the latter incurring higher computational cost to search through fewer architectures.

Finally, we saw the power of using sequential regression models over Bayesian mixture models with handcrafted basis functions. However, the models explored in this thesis do not admit a natural estimate of uncertainty, which forces us to use empirical variance estimates that are independent of the learning curve in question. We leave it to further work to explore models that do not sacrifice performance to obtain these estimates.

In conclusion, this thesis has explored a new frontier in hyperparameter optimization: large scale architecture search. We first show that tabular Q-learning can effectively learn to produce neural architectures, and then show that simple sequential regression models can be used to do state-of-the-art performance prediction and early stopping. We just scratched the surface in making architecture search practical, and we believe that this space has a bright future for alleviating the arduous task of architecture design that has until now be manual.

Appendix A

Tables

A.0.1 Top Model Architectures

In Tables A.1 through A.3, we present the top five model architectures selected with Q-learning for each dataset, along with their prediction error reported on the test set, and their total number of parameters. To download the Caffe solver and prototext files, please visit *https://bowenbaker.github.io/metaqnn/* .

Model Architecture	Test Error (%)	# Params (10^6)
[C(512,5,1), C(256,3,1), C(256,5,1), C(256,3,1), P(5,3), C(512,3,1), C(512,5,1), P(2,2), SM(10)]	6.92	11.18
[C(128,1,1), C(512,3,1), C(64,1,1), C(128,3,1), P(2,2), C(256,3,1), P(2,2), C(512,3,1), P(3,2), SM(10)]	8.78	2.17
[C(128,3,1), C(128,1,1), C(512,5,1), P(2,2), C(128,3,1), P(2,2), C(64,3,1), C(64,5,1), SM(10)]	8.88	2.42
[C(256,3,1), C(256,3,1), P(5,3), C(256,1,1), C(128,3,1), P(2,2), C(128,3,1), SM(10)]	9.24	1.10
[C(128,5,1), C(512,3,1), P(2,2), C(128,1,1), C(128,5,1), P(3,2), C(512,3,1), SM(10)]	11.63	1.66

Table A.1: Top 5 model architectures: CIFAR-10.

Model Architecture	Test Error (%)	# Params (10^6)
[C(128,3,1), P(2,2), C(64,5,1), C(512,5,1), C(256,3,1), C(512,3,1), P(2,2), C(512,3,1), C(256,5,1), C(256,3,1), C(128,5,1), C(64,3,1), SM(10)]	2.24	9.81
[C(128,1,1), C(256,5,1), C(128,5,1), P(2,2), C(256,5,1), C(256,1,1), C(256,3,1), C(256,3,1), C(256,5,1), C(512,5,1), C(256,3,1), C(128,3,1), SM(10)]	2.28	10.38
[C(128,5,1), C(128,3,1), C(64,5,1), P(5,3), C(128,3,1), C(512,5,1), C(256,5,1), C(128,5,1), C(128,5,1), C(128,3,1), SM(10)]	2.32	6.83
[C(128,1,1), C(256,5,1), C(128,5,1), C(256,3,1), C(256,5,1), P(2,2), C(128,1,1), C(512,3,1), C(256,5,1), P(2,2), C(64,5,1), C(64,1,1), SM(10)]	2.35	6.99
[C(128,1,1), C(256,5,1), C(128,5,1), C(256,5,1), C(256,5,1), C(256,1,1), P(3,2), C(128,1,1), C(256,5,1), C(512,5,1), C(256,3,1), C(128,3,1), SM(10)]	2.36	10.05

Table A.2: Top 5 model architectures: SVHN. Note that we do not report the *best* accuracy on test set from the above models in Tables 3.3 and 3.4 from the main text. This is because the model that achieved 2.28% on the test set performed the best on the validation set.

Model Architecture	Test Error (%)	# Params (10^6)
[C(64,1,1), C(256,3,1), P(2,2), C(512,3,1), C(256,1,1), P(5,3), C(256,3,1), C(512,3,1), FC(512), SM(10)]	0.35	5.59
[C(128,3,1), C(64,1,1), C(64,3,1), C(64,5,1), P(2,2), C(128,3,1), P(3,2), C(512,3,1), FC(512), FC(128), SM(10)]	0.38	7.43
[C(512,1,1), C(128,3,1), C(128,5,1), C(64,1,1), C(256,5,1), C(64,1,1), P(5,3), C(512,1,1), C(512,3,1), C(256,3,1), C(256,5,1), C(256,5,1), SM(10)]	0.40	8.28
[C(64,3,1), C(128,3,1), C(512,1,1), C(256,1,1), C(256,5,1), C(128,3,1), P(5,3), C(512,1,1), C(512,3,1), C(128,5,1), SM(10)]	0.41	6.27
[C(64,3,1), C(128,1,1), P(2,2), C(256,3,1), C(128,5,1), C(64,1,1), C(512,5,1), C(128,5,1), C(64,1,1), C(512,5,1), C(256,5,1), C(64,5,1), SM(10)]	0.43	8.10
[C(64,1,1), C(256,5,1), C(256,5,1), C(512,1,1), C(64,3,1), P(5,3), C(256,5,1), C(256,5,1), C(512,5,1), C(64,1,1), C(128,5,1), C(512,5,1), SM(10)]	0.44	9.67
[C(128,3,1), C(512,3,1), P(2,2), C(256,3,1), C(128,5,1), C(64,1,1), C(64,5,1), C(512,5,1), GAP(10), SM(10)]	0.44	3.52
[C(256,3,1), C(256,5,1), C(512,3,1), C(256,5,1), C(512,1,1), P(5,3), C(256,3,1), C(64,3,1), C(256,5,1), C(512,3,1), C(128,5,1), C(512,5,1), SM(10)]	0.46	12.42
[C(512,5,1), C(128,5,1), C(128,5,1), C(128,3,1), C(256,3,1), C(512,5,1), C(256,3,1), C(128,3,1), SM(10)]	0.55	7.25
[C(64,5,1), C(512,5,1), P(3,2), C(256,5,1), C(256,3,1), C(256,3,1), C(128,1,1), C(256,3,1), C(256,5,1), C(64,1,1), C(256,3,1), C(64,3,1), SM(10)]	0.56	7.55

Table A.3: Top 10 model architectures: MNIST. We report the top 10 models for MNIST because we included all 10 in our final ensemble. Note that we do not report the *best* accuracy on test set from the above models in Tables 3.3 and 3.4 from the main text. This is because the model that achieved 0.44% on the test set performed the best on the validation set.

Appendix B

Figures

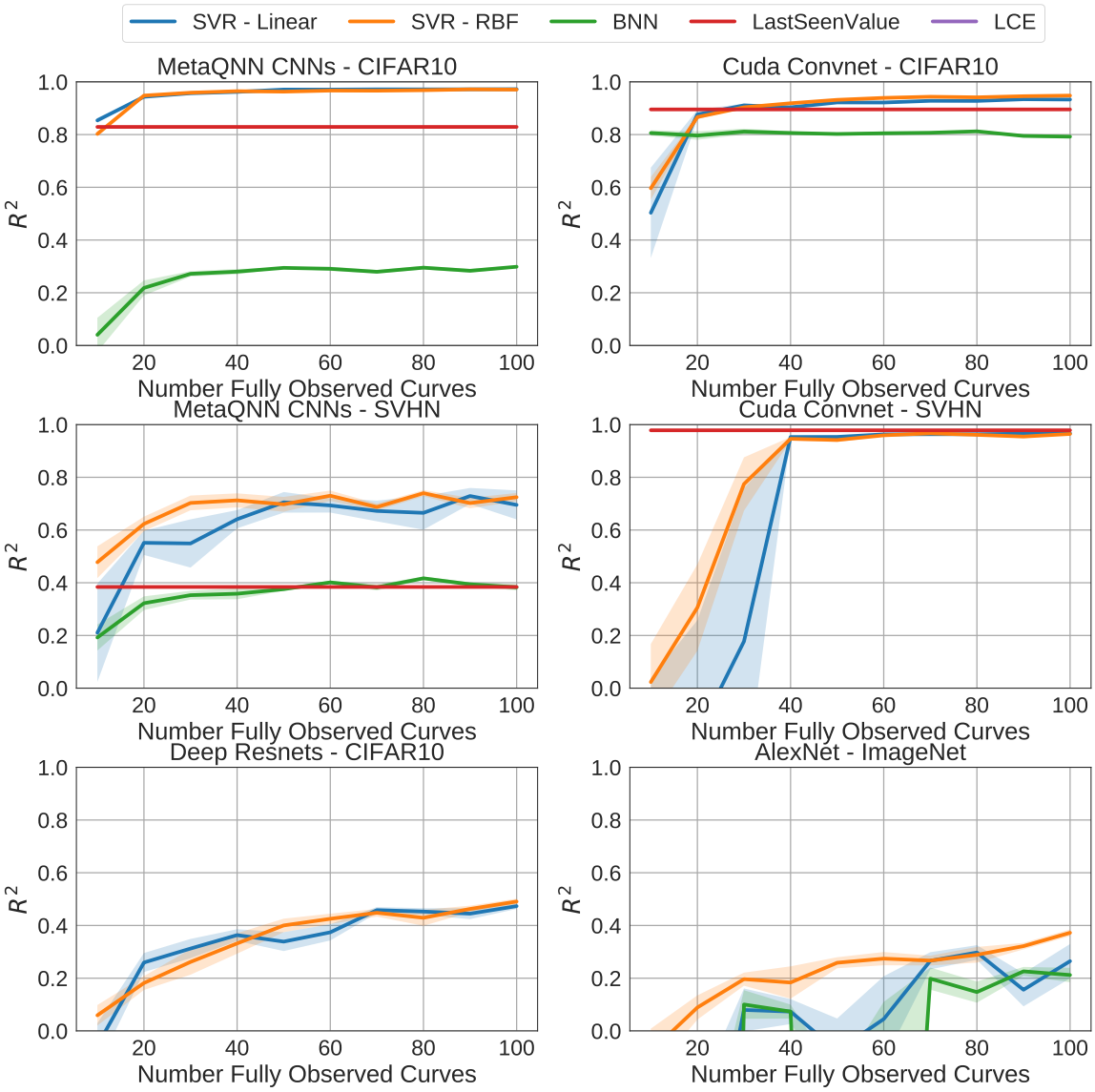


Figure B-1: **Performance Prediction Results Versus Training Set Size:** We plot the performance of each method versus the number of full learning curves observed where 25% of the learning curve is observed. For BNN and ν-SVR (linear and RBF), we sample 10 different training sets, plot the mean R^2, and shade the corresponding standard error. The left column shows results on problems varying the CNN architecture, and the right column shows results on problems varying optimization hyperparameters. We compare our method against BNN [30], LCE [29], and a "last seen value" heuristic [22]. Absent results for a model indicate that it did not achieve a positive R^2.

Bibliography

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

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

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

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, pages 1097–1105, 2012.

[5] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2012.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CVPR*, pages 1–9, 2015.

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

[9] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, England, 1989.

[10] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[11] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[12] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *JMLR*, 17(39):1–40, 2016.

[13] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[14] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. *European Conference on Machine Learning*, pages 437–448, 2005.

[15] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[16] Sander Adam, Lucian Busoniu, and Robert Babuska. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):201–212, 2012.

[17] 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*, pages 507–523. Springer, 2011.

[18] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *NIPS*, pages 2951–2959, 2012.

[19] 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.

[20] 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*, pages 2171–2180, 2015.

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

[22] 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.

[23] 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.

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

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

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

[27] Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. In *Advances in Neural Information Processing Systems 29*, pages 4053–4061. 2016.

[28] 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.

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

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

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

[32] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Practical neural network performance prediction for early stopping. *arXiv preprint arXiv:1705.10823*, 2017.

[33] Dimitri P Bertsekas. *Convex optimization algorithms*. Athena Scientific Belmont, 2015.

[34] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[35] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[36] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[37] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *arXiv preprint arXiv:1511.07289*, 2015.

[38] Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. *International Conference on Artificial Intelligence and Statistics*, 2016.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[40] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, 9:249–256, 2010.

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[43] Pierre Sermanet, Soumith Chintala, and Yann LeCun. Convolutional neural networks applied to house numbers digit classification. *ICPR*, pages 3288–3291, 2012.

[44] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. *CVPR*, pages 3626–3633, 2013.

[45] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C Courville, and Yoshua Bengio. Maxout networks. *ICML (3)*, 28:1319–1327, 2013.

[46] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[47] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

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

[49] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. *ICML*, pages 1058–1066, 2013.

[50] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. *AISTATS*, 2(3):6, 2015.

[51] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. *CVPR*, pages 3367–3375, 2015.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[53] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[54] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *NIPS*, pages 2546–2554, 2011.

[55] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[56] Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. New support vector algorithms. *Neural computation*, 12(5):1207–1245, 2000.

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