Proceedings of Machine Learning Research 129:273–288, 2020 ACML 2020

Deep-n-Cheap: An Automated Search Framework for Low Complexity Deep Learning

Sourya Dey SOURYADE@USC.EDU
Saikrishna C. Kanala KANALA@USC.EDU
Keith M. Chugg CHUGG@USC.EDU
Peter A. Beerel PABEEREL@USC.EDU
Ming Hsieh Department of Electrical and Computer Engineering
University of Southern California

Editors: Sinno Jialin Pan and Masashi Sugiyama

Abstract

We present Deep-n-Cheap – an open-source AutoML framework to search for deep learning models. This search includes both architecture and training hyperparameters, and supports convolutional neural networks and multilayer perceptrons. Our framework is targeted for deployment on both benchmark and custom datasets, and as a result, offers a greater degree of search space customizability as compared to a more limited search over only preexisting models from literature. We also introduce the technique of 'search transfer', which demonstrates the generalization capabilities of our models to multiple datasets.

Deep-n-Cheap includes a user-customizable complexity penalty which trades off performance with training time or number of parameters. Specifically, our framework results in models offering performance comparable to state-of-the-art while taking 1-2 orders of magnitude less time to train than models from other AutoML and model search frameworks. Additionally, this work investigates and develops various insights regarding the search process. In particular, we show the superiority of a greedy strategy and justify our choice of Bayesian optimization as the primary search methodology over random / grid search.

Keywords: Automated Machine Learning, Complexity Reduction, Bayesian Optimization, Neural Architecture Search

1. Introduction

Artificial neural networks (NNs) in deep learning systems are critical drivers of emerging technologies such as computer vision, text classification, and autonomous applications. In particular, convolutional neural networks (CNNs) are used for image related tasks while multilayer perceptrons (MLPs) can be used for general purpose tasks. Designing NNs involves decisions to be made regarding *hyperparameters*. As opposed to trainable parameters like weights and biases, hyperparameters are not learned by the network. Therefore, the challenging task of manually searching for, specifying and adjusting the values of large numbers of NN hyperparameters needs to be performed by an external entity, *i.e.,* the designer. Hyperparameters can be broadly grouped into two categories – a) architectural hyperparameters, such as the type of each layer and the number of nodes in it, and b) training hyperparameters, such as the learning rate and batch size.

© 2020 S. Dey, S.C. Kanala, K.M. Chugg & P.A. Beerel.

1.1. Motivation and Related Work

The problem of manually designing good NNs has resulted in several efforts towards automating this process. These include **AutoML frameworks** such as Auto-Keras (Jin et al. (2019)), AutoGluon (AWSLabs (2020)) and Auto-PyTorch (Mendoza et al. (2018)), which are open source software packages applicable to a variety of tasks and NN types. AutoML frameworks primarily focus on providing toolkits to search for good hyperparameter values.

Several other efforts place more emphasis on novel techniques for the hyperparameter search process. These can be broadly grouped into architecture search efforts such as Pham et al. (2018); Liu et al. (2019, 2018); Real et al. (2019); Xie and Yuille (2017); Tan and Le (2019); Cai et al. (2019a); He et al. (2018), and efforts that place a larger emphasis on training hyperparameters over architecture (Cai et al. (2019b); Snoek et al. (2012); Bergstra et al. (2013)). An alternate grouping is on the basis of search methodology – a) reinforcement learning (Pham et al. (2018); Zoph et al. (2018)), b) evolution (Real et al. (2019); Xie and Yuille (2017)), and c) Bayesian Optimization (Kandasamy et al. (2018); Snoek et al. (2012); Swersky et al. (2013)). Although the efforts described in this paragraph often come with publicly available software, they are typically not intended for general purpose use, *e.g.,* the code release for Cai et al. (2019a) only allows reproducing NNs on two datasets. This differentiates them from AutoML frameworks.

Deep NNs often suffer from **complexity** bottlenecks – either in storage, quantified by the *total number of trainable parameters* N_p, or computational, such as the number of FLOPs or the time taken to perform training and/or inference. Prior efforts on NN search penalize inference complexity in specific ways – latency in Cai et al. (2019a), FLOPs in Tan and Le (2019), and both in He et al. (2018). However, inference complexity is significantly different from training since the latter includes backpropagation and parameter updates every batch. For example, the resulting network for CIFAR-10 in Cai et al. (2019a) takes a minute to perform inference, but hours to train. Moreover, while there is considerable interest in popular benchmark datasets, in most real-world applications deep learning models need to be trained on custom datasets for which readymade, pre-trained models do not exist (Baldi et al. (2014); Santana and Hotz (2016)). This leads to an increasing number of resource-constrained devices needing to perform training on the fly, *e.g.,* self-driving cars.

The computing platform is also important, *e.g.,* changing batch size has a greater effect on training time per epoch on GPU than CPU. Therefore, calculating the FLOP count is not always an accurate measure of the time and resources expended in training a NN. Some previous works have proposed pre-defined sparsity (Dey et al. (2019); Dey et al. (2017)) and stochastic depth (Huang et al. (2016)) to reduce training time, while Page (2019) focuses on finding the quickest training time to get to a certain level of performance. Note that these are all manual methods, not automated search frameworks.

1.2. Overview and Contributions

This paper introduces *Deep-n-Cheap (DnC)* – an open-source AutoML framework to search for deep learning models[1]. We specifically target the training complexity bottleneck by including a penalty for *training time per epoch* t_{tr} in our search objective. The penalty

1. Source code available at https://github.com/souryadey/deep-n-cheap.

coefficient can be varied by the user to obtain a family of networks trading off performance and complexity. Additionally, we also support storage complexity penalties for N_p.

DnC searches for both architecture and training hyperparameters. While the architecture search derives some ideas from literature, DnC offers the user considerable customizability in specifying the search space. This is important for training on custom datasets that have significantly different requirements than those associated with benchmark datasets.

DnC primarily uses Bayesian Optimization (BO) and supports classification tasks using CNNs and MLPs. A notable aspect is *search transfer*, where we found that the best NNs obtained from searching over one dataset give good performance on a different dataset. This helps to improve generalization in NNs – such as on custom datasets – instead of purely optimizing for specific problems.

The following are the key contributions of this paper:

1. **Complexity**: To the best of our knowledge, DnC is the only AutoML framework targeting training complexity reduction. We show results on several datasets on both GPU and CPU. Our models achieve performance comparable to state-of-the-art, with training times that are 1-2 orders of magnitude less than those for models obtained from other AutoML frameworks and search efforts.

2. **Usability**: DnC offers a highly customizable three-stage search interface for both architecture and training hyperparameters. As opposed to Auto-Keras and AutoGluon, our search includes a) batch size that affects training times, and b) architectures beyond pre-existing ones found in literature. As a result, our target users include those who want to train quickly on custom datasets. As an example, our framework achieves the highest performance and lowest training times on the custom Reuters RCV1 dataset (Dey et al. (2019)). We also introduce *search transfer* to explore generalization capabilities of architectures to multiple datasets under different training hyperparameter settings.

3. **Insights**: We conduct investigations into the search process and draw several insights that will help guide a deeper understanding of NNs and search methodologies. We introduce a new similarity measure for BO and a new distance function for NNs. We empirically justify the value of our greedy three-stage search approach over less greedy approaches, and the superiority of BO over random and grid search.

The paper is structured as follows — Sec. 2 outlines our search methodology, Sec. 3 our experimental results, Sec. 4 includes additional investigations and insights, Sec. 5 compares with related work, and Sec. 6 concludes the paper.

2. Our Approach

Given a dataset, our framework searches for NN configurations (configs) through sequential stages in multiple search spaces. Each config is trained for the same number of epochs, *e.g.,* 100. There have been works on extrapolating NN performance from limited training (Baker et al. (2017); Liu et al. (2018)), however we train for a large number of epochs to predict with significant confidence the final performance of a NN after convergence. Configs

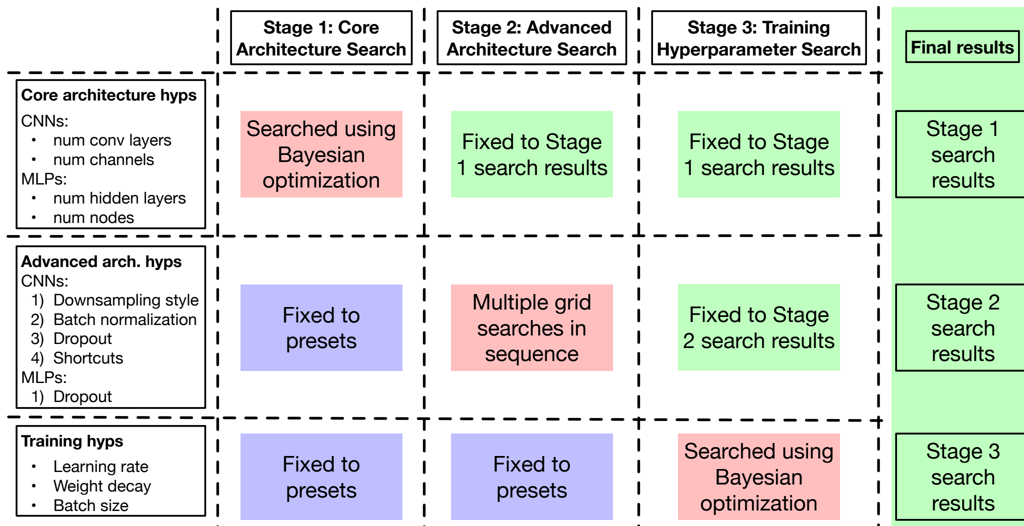

Figure 1: Three-stage search process for DnC.

are mapped to objective values using:

$$f(\text{Config}) = \log\left(f_p + w_c f_c\right) \tag{1}$$

where w_c controls the importance given to reducing complexity. The goal of the search is to minimize f. Its components are:

$$\text{Performance term:} \quad f_p = 1 - (\text{Best Validation Accuracy}) \tag{2a}$$

$$\text{Complexity term:} \quad f_c = \frac{c}{c_0} \tag{2b}$$

where c is the complexity metric for the current config (either t_{tr} or N_p), and c_0 is a reference value for the same metric (typically obtained for a high complexity config in the space). Lower values of w_c focus more on performance, *i.e.*, improving accuracy. One key contribution of this work is characterizing higher values of w_c that lead to reduced complexity NNs that train fast – these also reduce the search cost by speeding up the overall search process.

2.1. Three-stage search process

Stage 1 – Core architecture search: For CNNs, the combined search space consists of the number of convolutional (conv) layers and number of channels in each, while for MLPs, it is the number of hidden layers and number of nodes in each. Other architectural hyperparameters such as batch normalization (BN) and dropout layers and all training hyperparameters are fixed to presets that we found to work well across a variety of datasets and network depths. BO is used to minimize f and produce the resulting best config.

Stage 2 – Advanced architecture search: This stage starts from the resulting architecture from Stage 1 and searches over the following CNN hyperparameters – 1) whether

to use strides or max pooling layers for downsampling, 2) amount of BN layers, 3) amount of dropout layers and drop probabilities, and 4) amount of shortcut/skip connections. We use grid search instead of BO since the number of options in the search space is limited. Moreover, using a combined space yielded inferior results, so we use sequential sub-stages, *i.e.,* grid search first picks the downsampling choice leading to the minimum f value, then freezes that and searches over BN, and so on. This ordering yielded good empirical results, however, reordering is supported by the framework. For MLPs, there is a single grid search for dropout probabilities. As before, training hyperparameters are fixed to presets.

Stage 3 – Training hyperparameter search: The architecture is finalized after Stage 2. In Stage 3 – identical for CNNs and MLPs – we search over the combined space of initial learning rate η, weight decay λ, and batch size, using BO to minimize f. The final config after Stage 3 comprises both architecture and training hyperparameters. The complete process is summarized in Fig. 1.

Insights: While BO is an excellent choice for our search problem, it is known to perform poorly for highly multi-dimensional search spaces (Brochu et al. (2010)). This is why we break up the search space into three stages in a way such that latter stage hyperparameters depend on prior ones, *e.g.,* weight decay from Stage 3 depends on the core architecture in Stage 1. Note that the converse is also true, this is why the hyperparameters to be searched in latter stages automatically adapt their values based on each config as it is sampled in earlier stages. For example, sampled configs with more layers in Stage 1 automatically have more shortcut connections (these presets are also customizable). Note that we explored different orderings of the stages, however, these yielded inferior results.

2.2. Bayesian Optimization

Bayesian Optimization is useful for optimizing functions that are black-box and/or expensive to evaluate such as f, which requires NN training. The initial step when performing BO is to sample n_1 configs from the search space, $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{n_1}\}$, calculate their corresponding objective values, $\{f(\boldsymbol{x}_1), \cdots, f(\boldsymbol{x}_{n_1})\}$, and form a Gaussian prior. The mean vector $\boldsymbol{\mu}$ comprises the mean of the f values, and covariance matrix $\boldsymbol{\Sigma}$ is such that $\Sigma_{ij} = \sigma(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where $\sigma(\cdot,\cdot)$ is a *kernel function* that takes a high value $\in [0,1]$ if $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are similar.

Then the algorithm continues for n_2 steps, each step consisting of sampling n_3 configs, picking the config with the maximum *expected improvement*, computing its f value, and updating $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ accordingly. The reader is referred to Brochu et al. (2010) for a complete tutorial on BO – where eq. (4) in particular has details of expected improvement. Note that BO explores a total of $n_1 + n_2 n_3$ states in the search space, but the expensive f computation only occurs for $n_1 + n_2$ states.

2.2.1. Similarity between NN configurations

We begin by defining the *distance* between values of a particular hyperparameter k for two configs $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Larger distances denote dissimilarity. We have developed the *ramp* distance function:

$$d(x_{ik}, x_{jk}) = \omega_k \left(\frac{|x_{ik} - x_{jk}|}{u_k - l_k} \right)^{r_k} \tag{3a}$$

where u_k and l_k are respectively the upper and lower bounds for k, ω_k is a scaling coefficient, and r_k is a fractional power used for stretching differences. We experimented with similar distance functions defined in Hutter and Osborne (2013) and found ramp distance to be superior since it achieves similar performance with less functional hyperparameters to tune.

Note that d is 0 when $x_{ik} = x_{jk}$, and reaches a maximum of ω_k when they are the furthest apart. x_{ik} and x_{jk} are computed in different ways depending on k:

- If k is batch size or number of layers, x_{ik} and x_{jk} are the actual values.
- If k is η or λ, x_{ik} and x_{jk} are the logarithms of the actual values.
- When k is the hidden node configuration of a MLP, we sum the nodes together across all hidden layers. We found that the sum has a greater impact on f than considering hidden layers individually, *e.g.,* a config with three 300-node layers has a closer f value to a config with one 1000-node layer than a config with three 100-node layers.
- When k is the conv channel configuration of a CNN, we calculate individual distances for each layer. If the number of layers is different, the distance is maximum for each of the extra layers, *i.e.,* ω. This idea is inspired from Hutter and Osborne (2013), as compared to alternative similarity measures in Kandasamy et al. (2018); Jin et al. (2019). We follow this layer-by-layer comparison because our prior experiments showed that the representations learned by a certain conv layer in a CNN are similar to those learned by layers at the same depth in different CNNs. Additionally, this approach performed better than the summing across layers as in MLPs.

Each individual distance $d\left(x_{ik}, x_{jk}\right)$ is converted to its kernel value $\sigma\left(x_{ik}, x_{jk}\right)$ using the squared exponential function, then we take their convex combination for all K hyperparameters using coefficients $\{s_k\}$ to finally get $\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$. Fig. 2 shows an example.

$$\sigma\left(x_{ik}, x_{jk}\right) = \exp\left(-\frac{d^2(x_{ik}, x_{jk})}{2}\right) \tag{3b}$$

$$\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \sum_{k=1}^{K} s_k \sigma\left(x_{ik}, x_{jk}\right) \tag{3c}$$

3. Experimental Results

This section presents results of our search framework on different datasets for both CNN and MLP classification problems, along with the search settings used. Note that most of these *settings can be customized by the user* – this leads to one of our key contributions of using limited knowledge from literature to enable wider exploration of NNs for various custom problems. We used the Pytorch library on two platforms: a) *GPU* – an Amazon Web Services p3.2xlarge instance that uses a single NVIDIA V100 GPU with 16 GB memory and 8 vCPUs, and b) *CPU* – a mid-2014 Macbook Pro CPU with 2.2 GHz Intel Core i7 processor and 16GB 1.6 GHz DDR3 RAM.

The BO settings are $n_1 = n_2 = 15$ and $n_3 = 1000$. We later show the effects of varying these in Section 4.3. The hyperparameters not searched over include using the Adam optimizer and ReLU activation function.

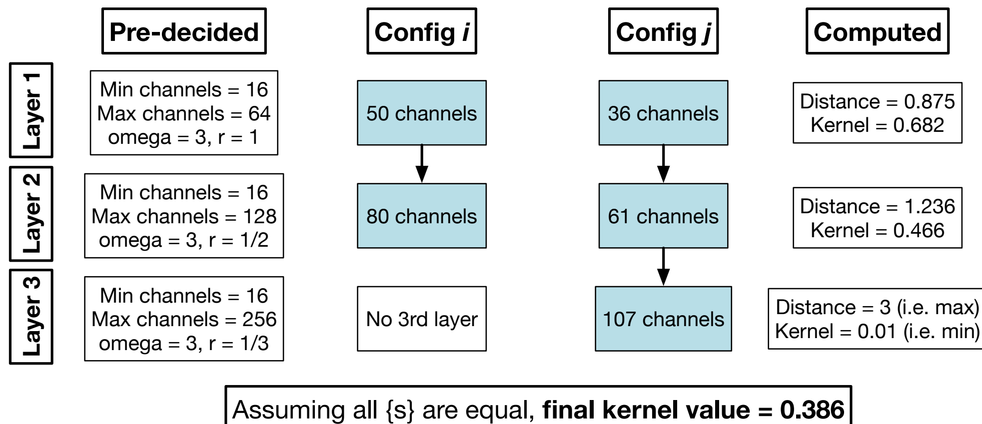

Figure 2: Calculating Stage 1 similarity for two conv channel configs: $\boldsymbol{x}_i = [50, 80]$ and $\boldsymbol{x}_j = [36, 61, 107]$. Taking the 1st conv layer as an example, the pre-decided values are $u_1 = 64$, $l_1 = 16$, $\omega_1 = 3$ and $r_1 = 1$. Distance $d_1 = 3 \times [(50-36)/(64-16)]^1 = 0.875$, and kernel value $\sigma_1 = \exp\left(-0.5 \times 0.875^2\right) = 0.682$. Similarly we get $\sigma_2 = 0.466$ and $\sigma_3 = 0.01$ (note that $d_3 = \omega_3$ due to the absence of a 3rd layer in $\boldsymbol{x}_i$). Combining these using $s_1 = s_2 = s_3 = 1/3$ yields $\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = 0.386$.

3.1. CNNs

All CNN experiments are on GPU. The datasets used are CIFAR-10 and -100 with train-validation-test splits of 40k-10k-10k, and Fashion MNIST (FMNIST) with 50k-10k-10k splits. Standard augmentation is always used – channel-wise normalization, random crops from 4 pixel padding on each side, and random horizontal flips. Augmentation requires Pytorch data loaders that incur timing overheads, so we also show results on unaugmented CIFAR-10 where the whole dataset is pre-loaded into memory and t_{tr} reduces as a result.

Stage 1 searches over CNNs with 4–16 conv layers, the first of which has $c_1 \in \{16, 17, \cdots, 64\}$ channels and each subsequent layer has $c_{i+1} \in \{c_i, c_i + 1, \cdots, \min(2c_i, 512)\}$ channels. We allow the number of channels in a layer to have arbitrary integer values, not just fixed to multiples of 8. For **Stage 2**, the first grid search is over all possible combinations of using either strides or max pooling for the downsampling layers. Second, we vary the fraction of BN layers through $[0, 1/4, 1/2, 3/4]$. Third, we vary the fraction of dropout layers in a manner similar to BN, and drop probabilities over $[0.1, 0.2]$ for the input layer and $[0.15, 0.3, 0.45]$ for all other layers. Fourth, we search over shortcut connections – none, every 4th layer, or every other layer. Finally in **Stage 3**, we usearch over a) $\eta \in \{10^x\}$ for $x \in [1, 5]$, b) $\lambda \in \{10^x\}$ for $x \in [-6, -3]$, with λ converted to 0 when $x < -5$, and c) batch sizes in $[32, 33, \cdots, 512]$. We found that batch sizes that are not powers of 2 did not lead to any slowdown on the platforms used.

The complexity term f_c uses normalized t_{tr}, since this is the major bottleneck in developing CNNs. Each config was trained for 100 epochs on the train set and evaluated on the validation set to obtain f_p. We ran experiments for 5 values of w_c: $[0, 0.01, 0.1, 1, 10]$. The

best network from each search was then trained on the combined training and validation set, and evaluated on the test set for 300 epochs to get final test accuracies and t_{tr} values.

As shown in Fig. 3, we obtain a family of networks by varying w_c. Performance in the form of test accuracy trades off with complexity in the form of t_{tr}. The latter is correlated with search cost and N_p. The last row of figures directly plot the performance-complexity tradeoff. These curves rise sharply on the left and flatten out on the right, indicating **diminishing performance returns as complexity is increased**. This highlights one of our key contributions – allowing the user to choose fast training NNs that perform well.

Taking augmented CIFAR-10 as an example, DnC found the following best config for $w_c = 0$: 14 conv layers with $\{c\} = (50, 52, 53, 59, 95, 96, 97, 120, 193, 239, 351, 385, 488, 496)$, the 4th layer has a stride of 2 while max pooling follows layers 8 and 10, BN follows all conv layers, dropout with drop probability 0.3 follows every other conv block, and skip connections are present for every other conv block. The best found η is 10^{-3}, batch size is 120 and λ is 3.35×10^{-5}. We note that we achieve good performance with a NN that has irregular $\{c\}$ values and is also not very deep – this is consistent with the findings in Zagoruyko and Komodakis (2016).

On the other hand, the best config found for $w_c = 10$ only has 4 conv layers.

3.2. MLPs

While CNNs are useful for image-related tasks, MLPs are more generalized and can be used for other tasks, or images in permutation-invariant format. We ran CPU experiments on the MNIST and FMNIST datasets in this format without any augmentation, and GPU experiments on the custom Reuters RCV1 dataset constructed as given in Dey et al. (2019). Each dataset is loaded into memory in its entirety, eliminating data loader overheads.

For **Stage 1**, we search over 0–2 hidden layers for MNIST and FMNIST, number of nodes in each being 20–400. These numbers change for RCV1 to 0–3 and 50–1000 since it is a larger dataset. For **Stage 2**, we do a grid search over drop probabilities in $[0, 0.1, 0.3, 0.4, 0.5]$, and for **Stage 3**, the training hyperparameter search is identical to CNNs.

We ran separate searches for individual penalty functions – normalized t_{tr} and normalized N_p. The latter is owing to the fact that MLPs often massively increase the number of parameters and thereby storage complexity of NNs (Krizhevsky et al. (2012)). The train-validation-test splits for MNIST and FMNIST are 50k-10k-10k, and 178k-50k-100k for RCV1. Candidate networks were trained for 60 epochs and the final networks tested after 180 epochs. As before, $w_c \in [0, 0.01, 0.1, 1, 10]$ for MNIST and FMNIST. For RCV1, the results for $w_c = 10$ were mostly similar to $w_c = 1$, so we replace 10 with 0.03. Plots are shown in Fig. 4, where pink dots and black crosses are respectively for t_{tr} and N_p penalties.

The trends in Fig. 4 are qualitatively similar to those in Fig. 3. When penalizing N_p, the two lowest complexity networks in each case have no hidden layers, so they both have exactly the same N_p (results differ due to different training hyperparameters). Of interest is the subfigure on the bottom right, indicating much longer search times when penalizing N_p as compared to t_{tr}. This is because time is not a factor when penalizing N_p, so the search picks smaller batch sizes that increase t_{tr} with a view to improving performance. Interestingly enough, this does not actually lead to performance benefit as shown in the subfigure on the top-right, where the black crosses occupy similar locations as the pink dots.

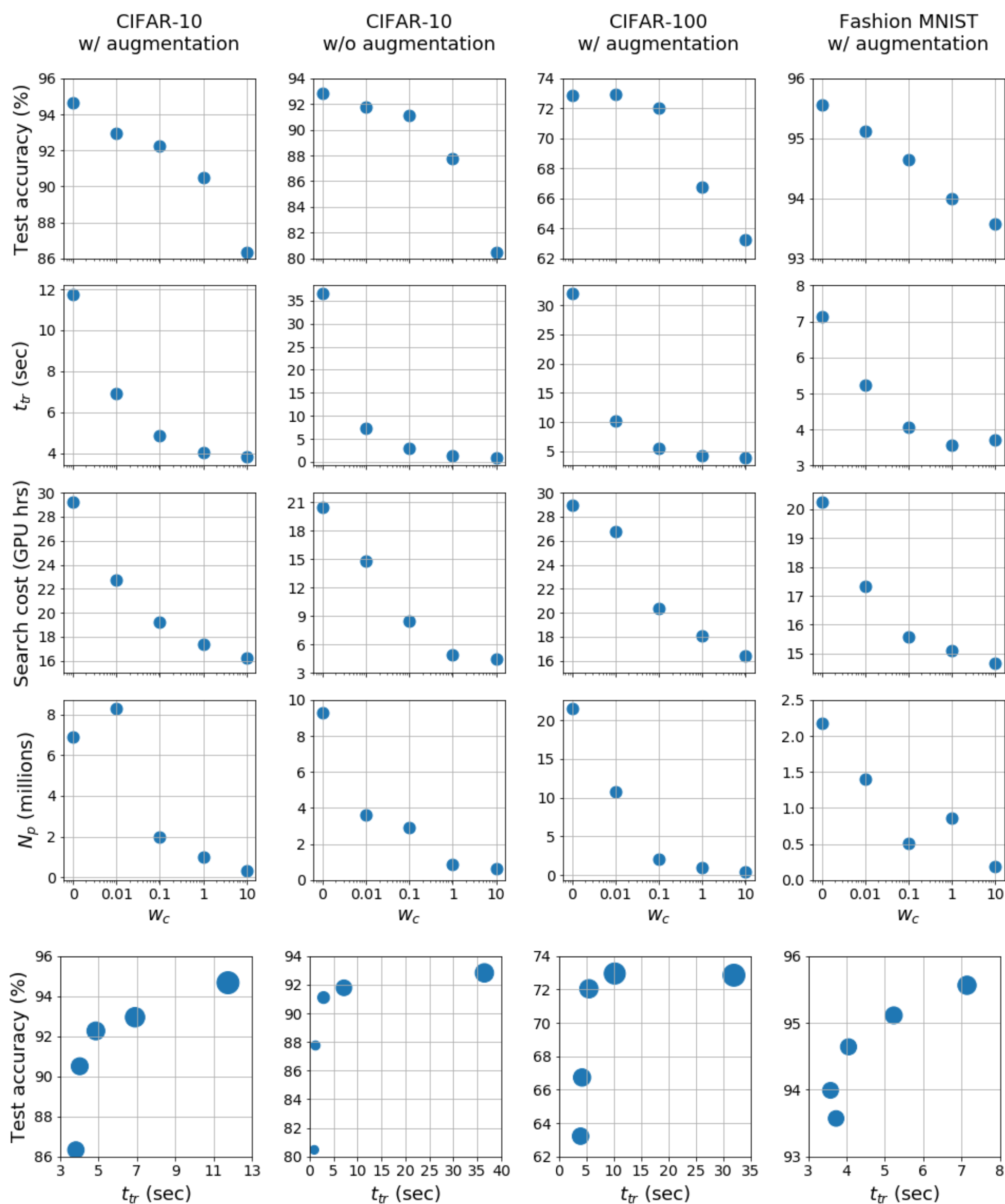

Figure 3: Characterizing a family of NNs for CIFAR-10 augmented (1st column), unaugmented (2nd column), CIFAR-100 augmented (3rd column) and FMNIST augmented (4th column), obtained from DnC for different w_c. We plot test accuracy in 300 epochs (1st row), t_{tr} on combined train and validation sets (2nd row), search cost (3rd row) and N_p (4th row), all against w_c. The 5th row shows the performance-complexity tradeoff, with dot size proportional to search cost.

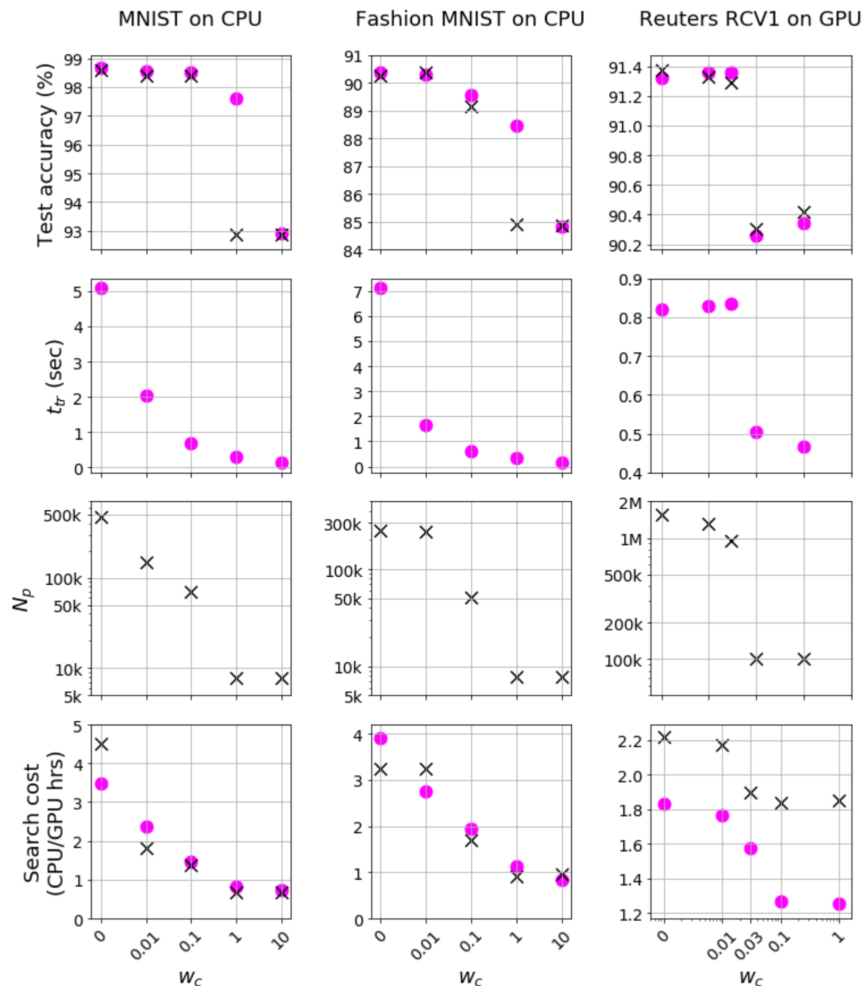

Figure 4: Characterizing a family of NNs for MNIST (1st column) and FMNIST (2nd column) on CPU, and RCV1 (3rd column) on GPU, obtained from DnC for different w_c. We plot test accuracy in 180 epochs (1st row), t_{tr} on combined train and validation sets (2nd row), N_p (3rd row), and search cost (4th row), all against w_c. The search penalty is t_{tr} for the pink dots and N_p for the black crosses.

4. Investigations and insights

4.1. Search transfer

One goal of our search framework is to find models that are applicable to a wide variety of problems and datasets suited to different user requirements. To evaluate this aspect, we experimented on whether a NN architecture found from searching through Stages 1 and 2 on dataset A can be applied to dataset B after searching for Stage 3 on it. In other words, how does transferring an architecture compare to 'native' configs, *i.e.,* those searched for through all three stages on dataset B. This process is shown on the left in Fig. 5. Note

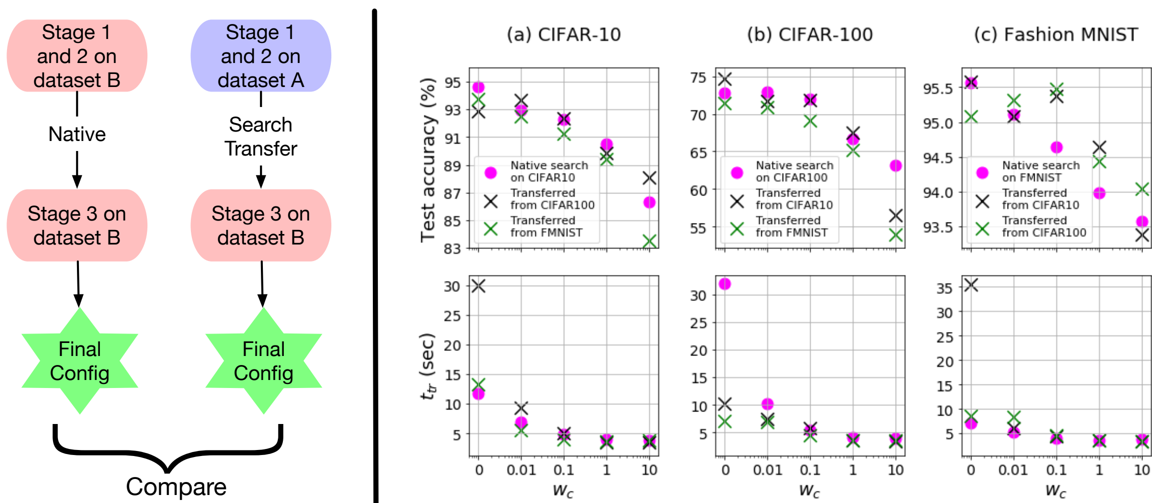

Figure 5: *Left*: Process of search transfer – comparing configs obtained from native search with those where Stage 3 is done on a dataset different from Stages 1 and 2. *Right*: Results of CNN search transfer to (a) CIFAR-10, (b) CIFAR-100, (c) FMNIST. All datasets are augmented. Pink dots denote native search.

that we repeat Stage 3 of the search since it optimizes training hyperparameters such as weight decay, which are related to the capacity of the network to learn a new dataset. This is contrary to simply transferring the architecture as in Zoph et al. (2018).

We took the best CNN architectures found from searches on CIFAR-10, CIFAR-100 and FMNIST (as depicted in Fig. 3) and transferred them to each other for Stage 3 searching. The results for test accuracy and t_{tr} are shown on the right in Fig. 5. We note that the architectures generally transfer well. In particular, transferring from FMNIST (green crosses in subfigures (a) and (b)) results in slight performance degradation since those architectures have N_p around 1M-2M, while some architectures found from native searches (pink dots) on CIFAR have $N_p > 20$M. However, architectures transferred between CIFAR-10 and -100 often exceed native performance. Moreover, almost all the architectures transferred from CIFAR-100 (green crosses in subfigure (c)) exceed native performance on FMNIST, which again is likely due to bigger N_p. We also note that t_{tr} values remain very similar on transferring, except for the $w_c = 0$ case where there is absolutely no time penalty.

4.2. Greedy strategy

Our search methodology is greedy in the sense that it preserves only the best config resulting in the minimum f value from each stage and sub-stage. We also experimented with a non-greedy strategy. Instead of one, we picked the three best configs from Stage 1 – $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$, then ran separate grid searches on each of them to get three corresponding configs at the end of Stage 2, and finally picked the three best configs for each of their Stage 3 runs for a total of nine different configs – $\{\boldsymbol{x}_{11}, \boldsymbol{x}_{12}, \boldsymbol{x}_{13}, \boldsymbol{x}_{21}, \cdots, \boldsymbol{x}_{33}\}$. Following a purely greedy

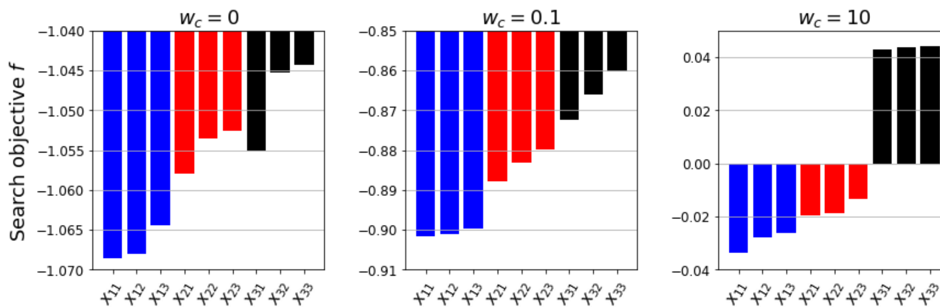

Figure 6: Search objective values (lower the better) for three best configs from Stage 1 (blue, red, black), optimized through Stages 2 and 3 and three best configs chosen for each in Stage 3. Results shown for different w_c on CIFAR-10 unaugmented.

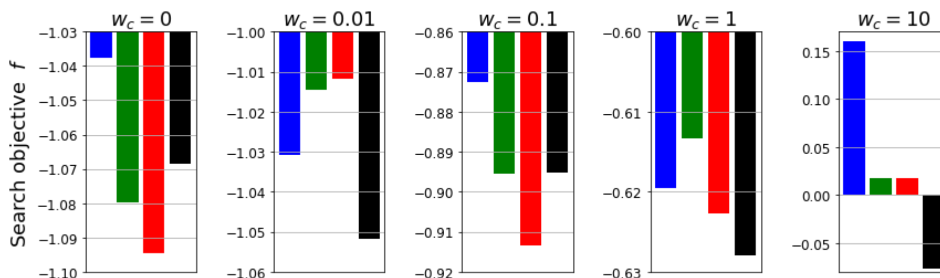

Figure 7: Search objective values (lower the better) for purely random search (blue) vs purely grid search via Sobol sequencing (green) vs balanced BO (red) vs extreme BO (black). Results shown for different w_c on CIFAR-10 unaugmented.

approach would have resulted in only $\boldsymbol{x}_{11}$, while following a greedy approach for Stages 1 and 2 but not Stage 3 would have resulted in $\{\boldsymbol{x}_{11}, \boldsymbol{x}_{12}, \boldsymbol{x}_{13}\}$. We plotted the losses for each config for five different values of w_c on CIFAR-10 unaugmented (Fig. 6 shows three of these). In each case we found that following a purely greedy approach yielded best results, which justifies our choice for DnC.

4.3. Bayesian optimization vs random and grid search

We use Sobol sequencing – a space-filling method that selects points similar to grid search – to select initial points from the search space and construct the BO prior. We experimented on the usefulness and properties of BO by comparing the final search loss f achieved by performing the Stage 1 and 3 searches in four different ways. These are a) Random search – pick 30 prior points randomly, then 0 optimization steps, b) Grid search – pick 30 prior points via Sobol sequencing, then 0 optimization steps, c) Balanced BO (DnC default) – pick 15 prior points via Sobol sequencing, then 15 optimization steps, and d) Extreme BO – pick 1 initial point, then 29 optimization steps.

Table 1: Comparison of features of AutoML frameworks

Framework	Architecture search space	Training hyp search	Adjust model complexity
Auto-Keras	Only pre-existing architectures	No	No
AutoGluon	Only pre-existing architectures	Yes	No
Auto-PyTorch	Customizable by user	Yes	No
Deep-n-Cheap	Customizable by user	Yes	Penalize t_{tr}, N_p

The results in Fig. 7 are for different w_c on CIFAR-10. A form of BO outperforms random and grid search on each occasion. In particular, more optimization steps are beneficial for low complexity models, while the advantages of BO are not significant for high performing models. We believe that this is due to the fact that many deep nets are fairly robust to training hyperparameter settings (Zagoruyko and Komodakis (2016)).

5. Comparison to related work

Table 1 compares features of different AutoML frameworks. To the best of our knowledge, only DnC allows the user to specifically penalize complexity of the resulting models. This allows our framework to find models with performance comparable to other state-of-the-art methods, while significantly reducing the computational burden of training. This is shown in Table 2, which compares the search process and metrics of the final model found for CNNs on CIFAR-10, and Table 3, which does the same for MLPs on FMNIST and RCV1 for DnC and Auto-PyTorch only, since Auto-Keras and AutoGluon do not have explicit support for MLPs at the time of writing.

Table Metrics: Auto-Keras and AutoGluon do not support explicitly obtaining the final model from the search, which is needed to perform separate inference on the test set after the search. As a result, in order to have a fair comparison, Tables 2 and 3 use metrics from the search process – t_{tr} is for the train set and the performance metric is best validation accuracy. These are reported for the best model found from each search.

Note that ProxylessNAS (Cai et al. (2019a)) is not an AutoML framework, and hence out of scope for DnC comparisons. The primary point of including ProxylessNAS is to compare to a model with state-of-the-art accuracy that has been highly optimized for CIFAR-10. This model was trained in the original paper using stochastic depth and additional cutout augmentation, yielding 97.92% accuracy on their test set. We obtained this model from the authors (Cai and Authors (2020)) and, in order to make a fair comparison in Table 2, trained it without cutout and stochastic depth and report the best validation accuracy.

Auto-Keras and AutoGluon use fixed batch sizes across all models, while Auto-PyTorch and DnC search over batch sizes. We have included batch size since it affects t_{tr}. Each config for each search is run for the same number of epochs, as described in Sec. 3. The exception is Auto-PyTorch, where a key feature is variable number of epochs.

Analyzing Results: We note that for CNNs, DnC results in both the fastest t_{tr} and highest performance. The performance of ProxylessNAS is comparable, while taking 43X more time to train. This highlights one of our key features – the ability to find models with

Table 2: Comparing frameworks on CNNs for CIFAR-10 augmented on GPU

Framework	Additional settings	Search cost (GPU hrs)	Best model found from search			
			Architecture	t_{tr} (sec)	Batch size	Best val acc (%)
ProxylessNAS	Proxyless-G	96	537 conv layers	429	64	93.22
Auto-Keras	Default run	14.33	Resnet-20 v2	33	32	74.89
AutoGluon	Default run	**3**	Resnet-20 v1	37	64	88.6
	Extended run	101	Resnet-56 v1	46	64	91.22
Auto-Pytorch	'tiny cs'	6.17	30 conv layers	39	64	87.81
	'full cs'	6.13	41 conv layers	31	106	86.37
Deep-n-Cheap	$w_c = 0$	29.17	14 conv layers	10	120	**93.74**
	$w_c = 0.1$	19.23	8 conv layers	4	459	91.89
	$w_c = 10$	16.23	4 conv layers	**3**	256	83.82

Table 3: Comparing AutoML frameworks on MLPs for FMNIST and RCV1 on GPU

Framework	Additional settings	Search cost (GPU hrs)	Best model found from search				
			MLP layers	N_p	t_{tr} (sec)	Batch size	Best val acc (%)
Fashion MNIST							
Auto-Pytorch	'tiny cs'	6.76	50	27.8M	19.2	125	**91**
	'medium cs'	5.53	20	3.5M	8.3	184	90.52
	'full cs'	6.63	12	122k	5.4	173	90.61
Deep-n-Cheap (penalize t_{tr})	$w_c = 0$	0.52	3	263k	0.4	272	90.24
	$w_c = 10$	**0.3**	1	**7.9k**	**0.1**	511	84.39
Deep-n-Cheap (penalize N_p)	$w_c = 0$	0.44	2	317k	0.5	153	90.53
	$w_c = 10$	0.4	1	**7.9k**	0.2	256	86.06
Reuters RCV1							
Auto-Pytorch	'tiny cs'	7.22	38	19.7M	39.6	125	88.91
	'medium cs'	6.47	11	11.2M	22.3	337	90.77
Deep-n-Cheap (penalize t_{tr})	$w_c = 0$	1.83	2	1.32M	0.7	503	**91.36**
	$w_c = 1$	**1.25**	1	**100k**	**0.4**	512	90.34
Deep-n-Cheap (penalize N_p)	$w_c = 0$	2.22	2	1.6M	0.6	512	**91.36**
	$w_c = 1$	1.85	1	**100k**	5.54	33	90.4

performance comparable to state-of-the-art while massively reducing training complexity. The search cost is lowest for the default AutoGluon run, which only runs 3 configs. We also did an extended run for ~ 100 models on AutoGluon to make it match with DnC and Auto-Keras – this results in the longest search time without significant performance gain.

For MLPs, DnC has the fastest search times and lowest t_{tr} and N_p values – this is a result of it searching over simpler models with few hidden layers. While Auto-PyTorch performs slightly better for the benchmark FMNIST, our framework gives better performance for the more customized Reuters RCV1 dataset.

6. Conclusion and Future Work

In this paper we introduced Deep-n-Cheap – the first AutoML framework that specifically considers training complexity of the resulting models during searching. While our framework can be customized to search over any number of layers, it is interesting that we obtained competitive performance on various datasets using models significantly less deep than those obtained from other efforts in literature. We also found that it is possible to transfer a family

of architectures found using different w_c values between different datasets. The framework uses Bayesian optimization and a three-stage greedy search process – these were empirically demonstrated to be superior to other search methods and less greedy approaches.

The first release of DnC supports classification using CNNs and MLPs. The 'bleeding edge' version extends DnC to regression, while additional network types such as segmentation and recurrent are currently under development. Our future plans and subsequent releases will also expand the set of hyperparameters searched over. The framework is open source and offers considerable customizability to the user. We hope that DnC becomes widely used and provides efficient NN design solutions to many.

References

AWSLabs. AutoGluon: AutoML toolkit for deep learning. https://autogluon.mxnet.io/#, 2020.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. In *Proc. ICLR*, 2017.

P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308, 2014.

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

Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

Han Cai and Anon. Authors. Private communication with proxylessNAS authors, Mar 2020.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proc. ICLR*, 2019a.

Han Cai, Ligeng Zhu, and Song Han. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *Proc. ICLR*, 2019b.

Sourya Dey, Yinan Shao, Keith Chugg, and Peter Beerel. Accelerating training of deep neural networks via sparse edge processing. In *Proc. ICANN*, pages 273–280, 2017.

Sourya Dey, Kuan-Wen Huang, Peter A. Beerel, and Keith M. Chugg. Pre-defined sparse neural networks with hardware acceleration. *IEEE JETCAS*, 9(2):332–345, June 2019.

Yihui He, Ji Lin, and et al. AMC: AutoML for model compression and acceleration on mobile devices. In *Proc. ECCV*, pages 784–800, 2018.

Gao Huang, Yu Sun, and et al. Deep networks with stochastic depth. In *Proc. ECCV*, pages 646–661, 2016.

Frank Hutter and Michael A. Osborne. A kernel for hierarchical parameter spaces. *arXiv preprint arXiv:1310.5738*, 2013.

Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proc. KDD*, pages 1946–1956, 2019.

Kirthevasan Kandasamy, Willie Neiswanger, and et al. Neural architecture search with bayesian optimisation and optimal transport. In *Proc. NeurIPS*, page 2020–2029, 2018.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proc. NeurIPS*, pages 1097–1105, 2012.

Chenxi Liu, Barret Zoph, and et al. Progressive neural architecture search. In *Proc. ECCV*, pages 19–35, 2018.

H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *Proc. ICLR*, 2019.

Hector Mendoza, Aaron Klein, and et al. Towards automatically-tuned deep neural networks. In *AutoML: Methods, Sytems, Challenges*, chapter 7, pages 141–156. 2018.

David Page. How to train your resnet. https://myrtle.ai/how-to-train-your-resnet/, 2019.

Hieu Pham, Melody Guan, and et al. Efficient neural architecture search via parameter sharing. In *Proc. ICML*, pages 4095–4104, 2018.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *Proc. AAAI*, pages 4780–4789, 2019.

Eder Santana and George Hotz. Learning a driving simulator. *arXiv preprint arXiv:1608.01230*, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proc. NeurIPS*, page 2951–2959, 2012.

Kevin Swersky, David Duvenaud, and et al. Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces. In *NeurIPS workshop on Bayesian Optimization in Theory and Practice*, 2013.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

L. Xie and A. Yuille. Genetic CNN. In *Proc. ICCV*, pages 1388–1397, 2017.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *Proc. BMVC*, pages 87.1–87.12, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proc. CVPR*, pages 8697–8710, 2018.