SN Computer Science (2021) 2:265
https://doi.org/10.1007/s42979-021-00646-0

ORIGINAL RESEARCH

Deep-n-Cheap: An Automated Efficient and Extensible Search Framework for Cost-Effective Deep Learning

Sourya Dey[1] · **Sara Babakniya**[1] · **Saikrishna C. Kanala**[1] · **Marco Paolieri**[1] · **Leana Golubchik**[1] · **Peter A. Beerel**[1] · **Keith M. Chugg**[1]

Received: 22 February 2021 / Accepted: 13 April 2021 / Published online: 8 May 2021

Abstract
Artificial neural networks (NNs) in deep learning systems are critical drivers of emerging technologies such as computer vision, text classification, and natural language processing. Fundamental to their success is the development of accurate and efficient NN models. In this article, we report our work on Deep-n-Cheap—an open-source automated machine learning (AutoML) search framework for deep learning models. The search includes both architecture and training hyperparameters and supports convolutional neural networks and multi-layer perceptrons, applicable to multiple domains. 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 pre-existing models from literature. We also introduce the technique of 'search transfer', which demonstrates the generalization capabilities of the models found by our framework 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 can find models with 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, we investigate and develop insight into the search process that should aid future development of deep learning models.

Keywords Automated machine learning · Complexity reduction · Bayesian optimization · Neural architecture search

Introduction

Artificial NNs in deep learning systems are critical drivers of emerging technologies such as computer vision, text classification, and autonomous applications. In particular, one and two dimensional convolutional neural networks (CNNs) are used for text and image related tasks respectively, while multilayer perceptrons (MLPs) can be used for general purpose classification tasks. Manually designing these NNs is challenging since they typically have a large number of interconnected layers [23, 40] and require a large number of decisions to be made regarding hyperparameters. These hyperparameters, as opposed to trainable parameters like weights and biases, are not learned by the network. They need to be specified and adjusted by an external entity, i.e., the designer. They 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. The difficulty of manually designing hyperparameters to find a good NN is exacerbated by the fact that several hyperparameters interact with each other to have a combined effect on the final performance.

Motivation and Related Work: The problem of searching for good NNs has resulted in several efforts towards automating this process. These efforts include AutoML frameworks such as Auto-Keras [19], AutoGluon [1] and Auto-PyTorch [27], which are open source software packages applicable to a variety of tasks and types of NNs. The

Supported by the Defense Threat Reduction Agency (DTRA), USA and by the NSF CCF-1763747 and the NSF CNS-1816887 awards.

This article is part of the topical collection "ACML 2020" guest edited by Masashi Sugiyama, Sinno Jialin Pan, Thanaruk Theeramunkong and Wray Buntine.

✉ Sourya Dey
souryade@usc.edu

[1] University of Southern California, Los Angeles, USA

major focus of these efforts is on providing user-friendly toolkits to search for good hyperparameter values.

Several other efforts place more emphasis on novel techniques for the search process. These can be broadly grouped into neural architecture search (NAS) efforts such as Refs. [2, 9, 15, 24, 25, 28, 30, 31, 35, 37], and efforts that place a larger emphasis on training hyperparameters over architecture [5, 8, 33, 36]. An alternate grouping is on the basis of search methodology: (a) reinforcement learning [2, 30, 42], (b) evolution/genetic operations [28, 31, 37], (c) gradient-based optimization [9, 25, 38] and (d) Bayesian optimization [22, 33, 34, 36]. 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 Ref. [9] 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 Ref. [9], FLOPs in Ref. [35], and both in Ref. [15]. 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 Ref. [9] 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 [4, 26, 32]. 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 [10, 12, 43] and stochastic depth [16] to reduce training time, while [29] focuses on finding the quickest training time to get to a certain level of performance. Note that these are all manual methods, not search frameworks.

Overview and contributions: This paper describes DnC—an open-source[1] AutoML framework to search for deep learning models, initially introduced by us in Ref. [11]. We specifically target the training complexity bottleneck by including a penalty for training time per epoch t_{tr} in our search objective. The penalty coefficient can be varied by the user to obtain a family of networks trading off performance and complexity. Additionally, we also support storage complexity (number of parameters) penalties for N_p.

DnC searches for both architecture and training hyperparameters. While the architecture search derives some ideas from literature, we have striven to offer the user a considerable amount of customizability in specifying the search space. This is important for training on custom datasets which can have significantly different requirements than those associated with benchmark datasets.

DnC primarily uses Bayesian optimization (BO) and currently 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 and search efforts.
2. Usability: DnC offers a highly customizable three-stage search interface for both architecture and training hyperparameters. As opposed to AutoKeras and AutoGluon, our search includes batch size (which affects training times) and architectures beyond predefined ones. Our target users include those who want to train quickly on custom datasets, using custom architectures. We show that DnC can find MLPs with state-of-the-art accuracy and minimum training time for text categorization on the custom Reuters RCV1 dataset [10]; we also explore the use of DnC on a custom natural language processing (NLP) architecture (inspired by [21]) for sentiment analysis on Yelp datasets [41] of user reviews. We also present search transfer to illustrate how architectures found by DnC can be used on multiple datasets after a quick search over the hyperparameters used during training.
3. Insights: We conduct investigations into the search process and provide several insights that can lead to a deeper understanding of model search methodologies. In particular, 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. We also provide a new similarity measure for BO and new distance functions for MLPs, CNNs, and NLP models.

[1] The code and documentation are available at https://github.com/usc-hal/deep-n-cheap.

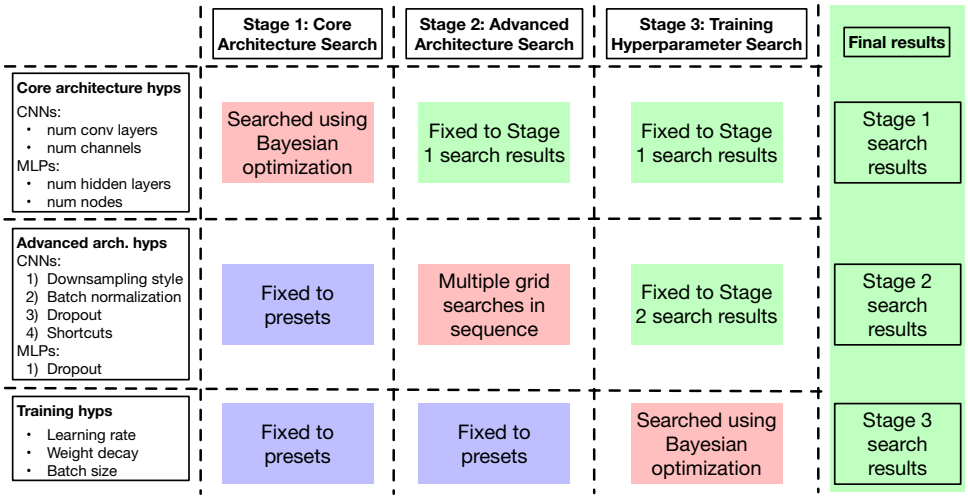

Fig. 1 Three-stage search process for DnC

The paper is structured as follows: "Our Approach" outlines our search methodology, "Experimental Results", "Investigations and Insights", "Comparison to Related Work", and "Conclusion and Future Work".

Our Approach

Given a dataset, our framework searches for neural network configurations through sequential stages in multiple search spaces. During a stage, each neural network configuration x is trained for the same number of epochs; to obtain a good estimate of its accuracy when trained until convergence, we use a large number of epochs (e.g., 100). Although other works [3, 24] try to predict NN performance from limited training, we found that accurate estimates are important for our model search method.

The goal of the search process is to minimize the objective function:

$$f(x) = \log(f_p(x) + w_c f_c(x)), \tag{1}$$

where the performance term $f_p(x)$ is the best validation error achieved by an NN configuration x (including architecture choices and training hyperparameters), w_c controls the importance given to reducing complexity, and the complexity term $f_c(x) = c(x)/c_0$ gives a complexity metric of x (based on either the per-epoch training time, t_{tr}, or the number of parameters, N_p) relative to a reference value c_0 (typically obtained for a high complexity configuration in the search space). Lower values of w_c place a greater emphasis on improving accuracy.

One key contribution of this work is exploring model search with higher values of w_c, which lead to reduced complexity NNs that train fast, also reducing the search cost by speeding up each evaluation of $f_p(x)$ in the search process.

Three-Stage Search Process

The search is divided into three stages, summarized in Fig. 1.

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. Bayesian optimization is used to minimize f and the corresponding best configuration is the result of Stage 1.

Stage 2 (advanced architecture search): This stage starts from the architecture found in Stage 1 and uses grid search to search for the following CNN hyperparameters through a sequence of sub-stages: (1) whether to use strides or max pooling layers for downsampling layers; (2) what fraction of layers should use BN; (3) what fraction of layers should use dropout, with specific drop probabilities; and (4) whether to add shortcut connections. This is not a combined space; instead 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 in the previous stage, training hyperparameters are fixed to presets. The result from Stage 2 is the result from the final sub-stage.

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 B, using BO to minimize f. The final configuration after Stage 3 comprises both architecture and training

hyperparameters. The complete process is summarized in Fig. 1.

Bayesian Optimization

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

Then the algorithm continues for n_2 steps, each step consisting of sampling n_3 configurations, picking the configuration with the maximum expected improvement, computing its f value, and updating $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ accordingly. For a complete tutorial on BO, the reader is referred to Ref. [6], 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 evaluation of f only occurs on a total of $n_1 + n_2$ states.

Similarity Between NN Configurations

We begin by defining the distance between values of a particular hyperparameter k for two configurations $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Larger distances denote dissimilarity. We initially considered the distance functions defined in Sects. 2 and 3 of Ref. [17], but then adopted an alternate one that resulted in similar performance with less tuning. We call it the ramp distance:

$$d(x_{ik}, x_{jk}) = \omega_k \left(\frac{\left| x_{ik} - x_{jk} \right|}{u_k - l_k} \right)^{r_k}, \tag{2}$$

where u_k and l_k are respectively the upper and lower bounds for parameter k, ω_k is a scaling coefficient, and r_k is a fractional power used for stretching small differences. Note that d is 0 when $x_{ik} = x_{jk}$, and reaches a maximum of ω_k when they are the farthest apart. x_{ik} and x_{jk} are computed in different ways depending on k:

- If k is batch size B or the 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 an MLP, we sum the nodes together across all hidden layers. This is because we found that the sum has a greater impact on f than considering layers individually, e.g., a configuration with three 300-node hidden layers has a closer f value to a configuration with one 1000-node hidden layer than a configuration with three 100-node hidden 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 by Ref. [17], as compared to alternative similarity measures in Refs. [19, 22]. 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 $d(x_{ik}, x_{jk})$ is converted to its kernel value $\sigma(x_{ik}, x_{jk})$ using the squared exponential function, then we take their convex combination for all K hyperparameters using coefficients $\{s_k\}$ to finally get $\sigma(\boldsymbol{x}_i, \boldsymbol{x}_j)$:

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

$$\sigma(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_{k=1}^{K} s_k \sigma(x_{ik}, x_{jk}). \tag{4}$$

An example is illustrated in Fig. 2.

Note that, in the NLP models considered in "Experimental Results", while the initial layer is an embedding layer, i.e., a fully connected layer converting one-hot word vectors to dense feature vectors, the following layers are convolutional layers with max-pools and shortcuts. In these models, to measure the similarity between two points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ of the search space, we use a combination of the distances defined for MLP and CNN layers. In particular, the distance $d(x_{ik}, x_{jk})$ between embedding layers ($k = 1$) is computed as in MLPs (i.e., as the difference in size of embedding vectors), while the distance of the following layers ($k > 1$) is computed as in CNNs using Eq. (2).

Experimental Results

This section presents results obtained with our search framework on different datasets using MLP, CNN, and NLP model architectures, along with the search settings used. Note that most of the settings can be customized by the user; this is one of the key contributions of our framework, which uses limited knowledge from literature to enable wider exploration of neural network architectures for custom tasks. We used the PyTorch library on three platforms: (a) cloud GPU

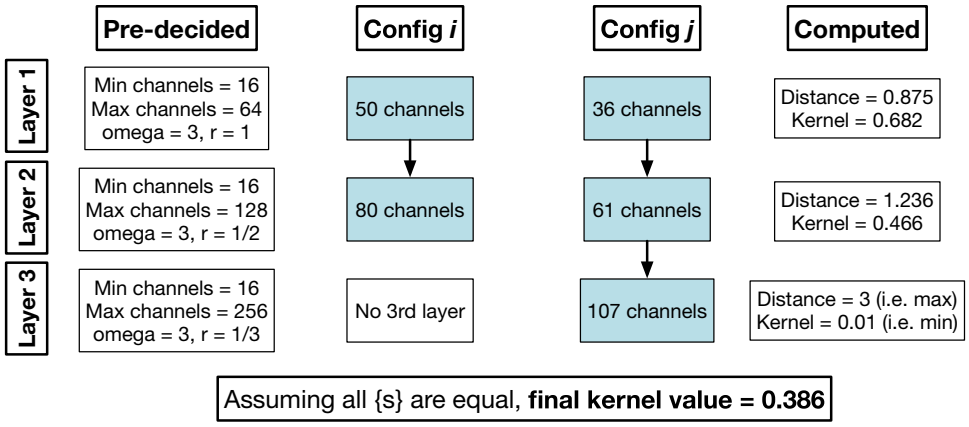

Fig. 2 Calculating Stage 1 similarity for two conv channel configurations: $\boldsymbol{x}_i = [50, 80]$ and $\boldsymbol{x}_j = [36, 61, 107]$. Taking the 1st conv layer as an example, the pre-decided values are $u_1 = 64$, $l_1 = 16$, $\omega_1 = 3$ and $r_1 = 1$ (more details on these choices in "Experimental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

instances, in particular AWS `p3.2xlarge` instances with a single Nvidia V100 GPU, 16 GB of memory and 8 vCPUs; (b) a private GPU cluster, where each node is equipped with four Nvidia GeForce RTX 2080 Ti GPUs, an Intel i9-9940X CPU, and 128 GB of RAM; (c) a laptop CPU, specifically the Intel i7 4870HQ CPU of a mid-2014 MacBook Pro with 16GB of memory. For Bayesian optimization, we used $n_1 = n_2 = 15$ and $n_3 = 1000$ in all experiments.

CNNs

All CNN experiments were conducted on cloud GPUs. The datasets used are CIFAR-10 and CIFAR-100 with a split in train-validation-test of 40k–10k–10k, and Fashion MNIST (FMNIST) with 50k–10k–10k. Standard augmentation is always used: channel-wise normalization, random crops from four pixel padding on each side, and random horizontal flips. Augmentation requires PyTorch data loaders that incur timing overheads, so we also show results on CIFAR-10 without augmentation, loading the entire dataset into memory when model search starts. As a result, t_{tr} is reduced.

For Stage 1, we use BO to search over CNNs with 4–16 conv layers, the first of which has $c_1 \in \{16, 17, \ldots, 64\}$ channels and each subsequent layer has $c_{i+1} \in \{c_i, c_i + 1, \ldots, \min\left(2c_i, 512\right)\}$ channels. We allow the number of channels in a layer to have arbitrary integer values, not just fixed to multiples of 8. Kernel sizes are fixed to 3×3. Downsampling precedes layers where c_i crosses 64, 128 and 256 (this is due to GPU memory limitations). During Stage 1, all conv layers are followed by BN and dropout with 30% drop probability. Configs with more than eight conv layers have shortcut connections. Global average pooling and a softmax classifier follow the conv portion. There are no hidden classifier layers, since we empirically obtained no performance benefit. For both Stages 1 and 2, we used the default Adam optimizer with $\eta = 10^{-3}$, decayed by 80% at the half and three-quarter points of training, batch size $B = 256$, and weight decay $\lambda = N_p \geq 10^6) \times N_p/10^{11}$, $\mathbb{I}$ being the indicator function. We empirically found this rule to work well.

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, 0.25, 0.5, 0.75]. For example, if there are seven conv layers, a setting of 0.5 will place BN layers after conv layers 2, 4, 6 and 7. 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. Finally, we search over shortcut connections—none, every 4th layer, or every other layer. Note that any shortcut connection skips over two layers.

For Stage 3, we used BO to search 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 $B \in [32, 33, \ldots, 512]$. We found that batch sizes that are not powers of 2 did not lead to any slowdown on the platforms used.

The penalty function f_c uses normalized t_{tr}, since this is the major bottleneck in developing CNNs. Each configuration was trained for 100 epochs on the train set and evaluated on the validation set to obtain f_p. We ran experiments for five different 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, where higher w_c values trade off test accuracy

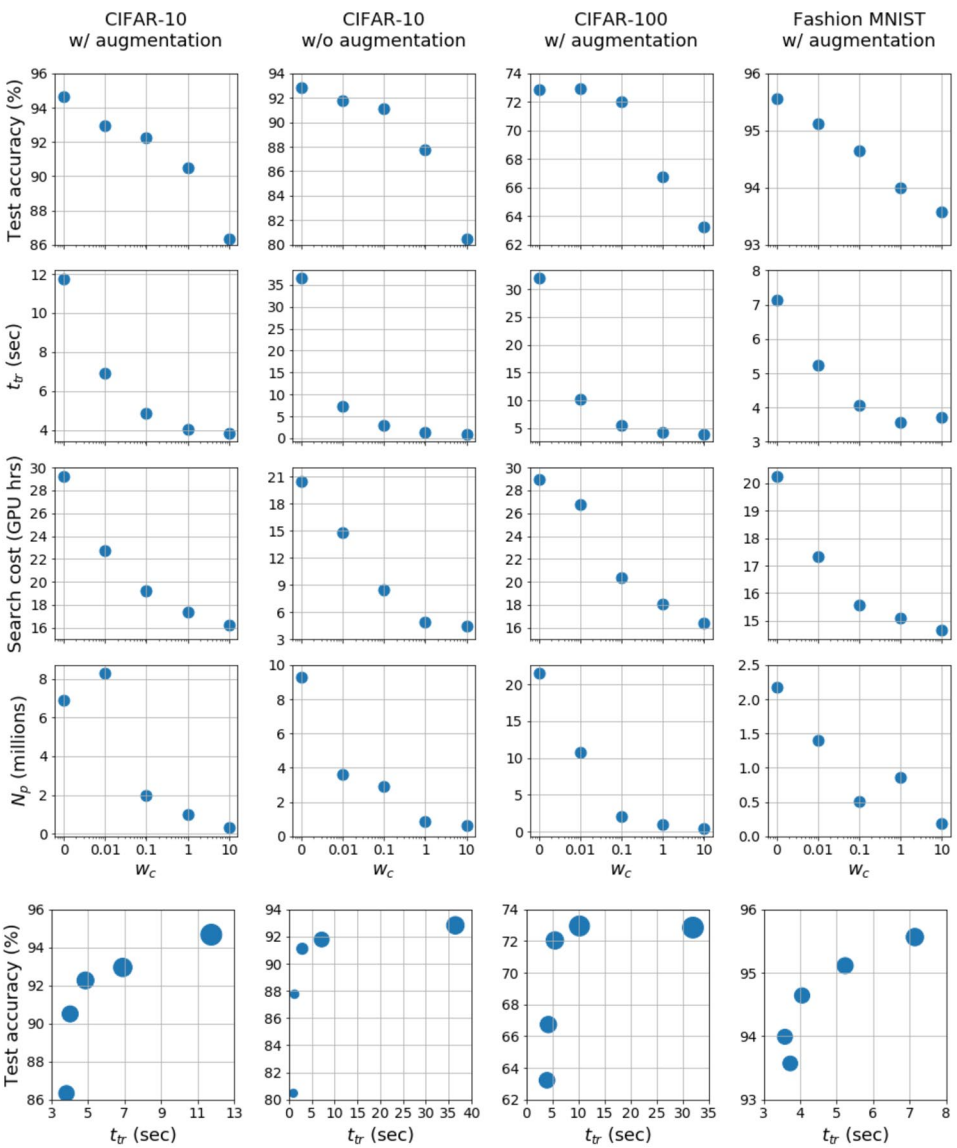

Fig. 3 Characterizing a family of NNs for CIFAR-10 augmented (1st column), unaugmented (2nd column), CIFAR-100 augmented (3rd column) and FMNIST augmented (4th column), obtained from DnC for different w_c. We plot test accuracy in 300 epochs (1st row), t_{tr} on combined train and validation sets (2nd 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

$1 - f_p(x)$ for lower computational cost 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 towards the left and flatten out towards 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 configuration 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 remains 10^{-3}, batch size B is 120 and weight decay $\lambda = 3.35 \times 10^{-5}$. Note that we achieve good performance with a NN that has irregular $\{c\}$ values and is also not very deep—the latter is consistent with the findings in Ref. [40]. Also note that the best configuration found for $w_c = 10$ has only four conv layers.

MLPs

For MLP models, we ran CPU experiments on the MNIST and FMNIST datasets using a permutation-invariant format (i.e., input images are flattened to a single layer of 784 pixels) without any augmentation, and GPU experiments on the Reuters RCV1 dataset constructed as in Ref. [10]. Each dataset is loaded

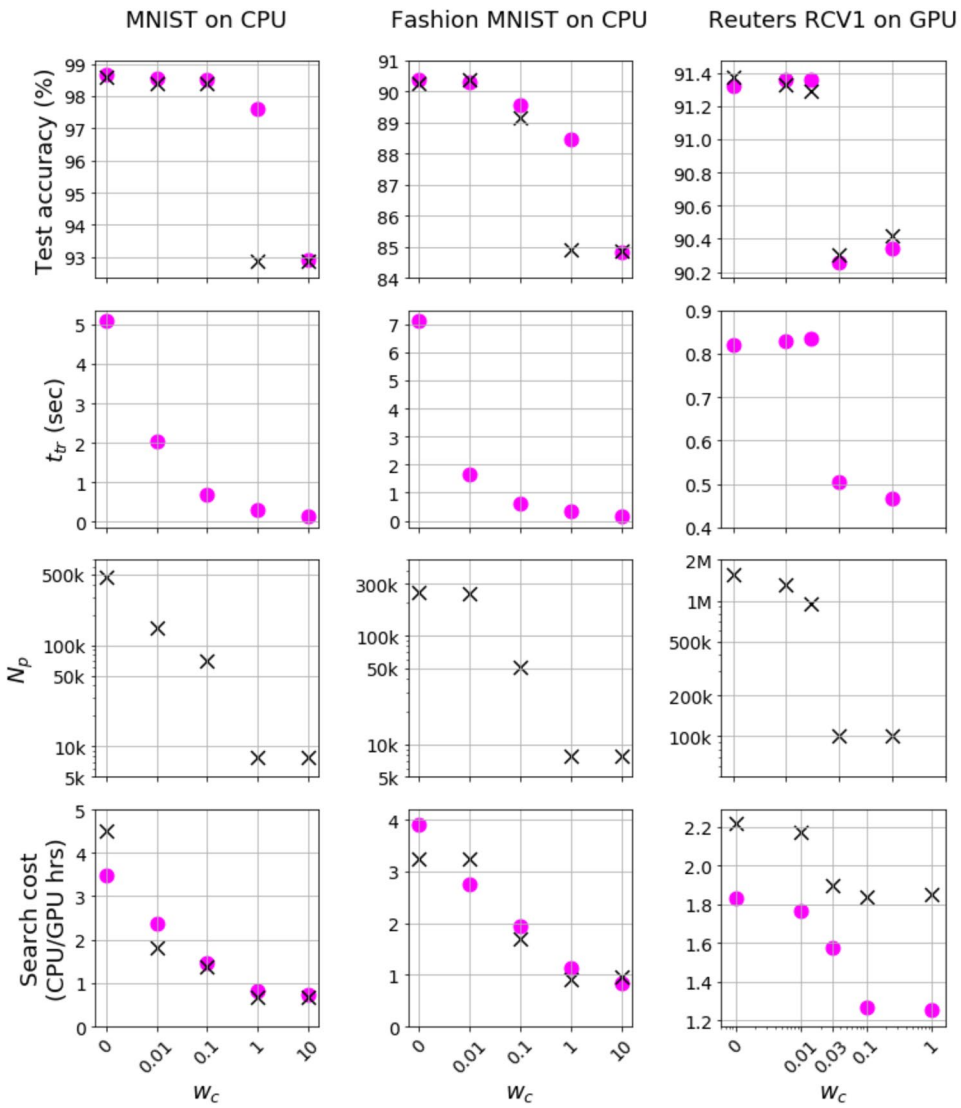

Fig. 4 Characterizing a family of NNs for MNIST (1st column) and FMNIST (2nd column) on CPU, and RCV1 (3rd column) on GPU, obtained from DnC for different w_c. We plot test accuracy in 180 epochs (1st row), t_{tr} on combined train and validation sets (2nd row), N_p (3rd row), and search cost (4th row),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

into memory in its entirety, eliminating data loading 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. Every layer is followed by a dropout layer with 20% drop probability. Training hyperparameters are fixed as in the case of CNNs, with the difference that $\lambda = \mathbb{I}(N_p \geq 10^4) \times N_p/10^9$ for MNIST and FMNIST and $\lambda = \mathbb{I}(N_p \geq 10^5) \times N_p/10^{10}$ for RCV1. 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 [23]. 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. The plots against w_c are shown in Fig. 4, where pink dots are for t_{tr} penalty and black crosses are for N_p penalty.

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.

Natural Language Models

To evaluate our automated model search approach on a different class of NN architectures and datasets and demonstrate DnC's extensibility, we consider the task of sentiment analysis, a common benchmark for natural language models. While only very large models based on transformers [39] reach state-of-the-art accuracy on this task, smaller models using word embeddings and conv layers such as deep pyramid CNNs (DPCNNs) [21] still achieve excellent accuracy with lower training times and inference requirements (both memory and computation). This class of architectures provides an interesting challenge for Deep-n-Cheap, as it presents significant differences with respect to CNNs used for image classification, while keeping training times reasonably low and manageable for automated model search.

Our main sentiment analysis dataset is the Yelp Reviews Full dataset (Yelp-5) [41], a balanced subset of the 2015 Yelp Challenge. Yelp-5 includes a training set of 650k user reviews, each with a score of 1–5 stars, and a test set of 50k reviews; each class has the same number of examples and the average number of words in a review is 155. As validation set for model search, we reserve 52k training examples.

To preprocess the dataset, we follow the same approach as in Ref. [21]: we remove HTML tags and stopwords from the reviews, and then use the 30k most common words as our vocabulary. We also use two special words to represent "out of vocabulary" words and "padding."

Our search space of NN architectures for sentiment analysis is also inspired by the work on DPCNNs [21], since these models achieve 68.2% accuracy in our experiments on Yelp-5, close to the 72.9% state-of-the-art accuracy of Ref. [39] (a transformer model, which can be quickly fine-tuned after long pretraining but requires considerable inference resources). Some key characteristics of DPCNNs are as follows:

- Overlapping regions of 1, 3 or 5 words centered on each word of a review are encoded as bag-of-word vectors of size 30,000 (the vocabulary size).
- An initial fully connected layer (the embedding layer) converts each bag-of-word vector to an embedding vector of size 250. Embedding vectors of each input region are concatenated into a variable-length one-dimensional (1D) vector.
- Models repeat up to seven blocks consisting of a downsampling layer (1D max-pooling with stride of 2, not included in the first block) and two conv layers (1D, kernel size of 3). Similarly to ResNets [13], shortcut connections are included; in particular, downsampled tensors are added to the output of the block.
- In contrast with ResNets, all conv layers have the same number of channels, 250, and stride of 1; thus, there is no need for additional operations on shortcut connections (e.g., padding or 1×1 projections) to match the dimension of the block output.
- ReLU pre-activations are used as in Ref.[14], but without batch normalization. By applying ReLUs to the input of each conv layer (instead of its output), max-pooling layers and tensor additions on shortcut connections receive linear activation signals.
- An array of max-pooling units with variable input size reduces the output of the last block to a fixed size, which is then used for classification in a fully-connected layer.

While borrowing the idea of using CNNs for NLP tasks from [21], we explore a large space of architecture variants where the output size of the embedding layer, the number of conv layers, and the number of channels of each layer are all configurable. To allow layers with variable channel sizes, we use padding on shortcut connections.

In Stage 1, to search over these architecture parameters, we use a strategy similar to DnC's search strategy for CNNs. Through BO, we select an embedding output size from [70, 300], and a number of conv layers from [2, 20]. For the first conv layer, we impose a number of channels $c_1 \in [50, 300]$; for each layer $i > 1$, we allow $c_i \in [c_{i-1}, 350]$ channels. Kernel sizes are fixed at 3; after the first layer, we add a stride-2 max-pooling layer with probability 0.5 to each layer that is not preceded by another max-pooling layer. During this stage, we use 30% dropout on each conv layer, add shortcut connections after every other conv layer (except the last one), and train using Adam (with batch size $B = 100$ and other hyperparameters as in "CNNs"). The input region size is set to 1 (one-hot word encoding).

With respect to DnC's search over image classification CNNs, in addition to changing the type of convolutions used (1D instead of 2D), we also exclude batch normalization (as in DPCNNs), and pad all reviews of a batch to the longest one (different batches can have different length). As described in "Similarity Between NN Configurations", we use distinct distance functions for the embedding layer and for conv layers.

In Stage 2, since DPCNNs use fixed convolution strides without batch normalization, we perform grid searches only to add or remove shortcut connections, to include or exclude dropout, and to select dropout probabilities from $[0.15,\ 0.45]$. We also consider dropout for the embedding layer with probability from $[0.1,\ 0.2]$.

In Stage 3, we search over learning rate and weight decay using BO with the same constraints as described in "CNNs". We did not search over batch size because different batch

Table 1 Comparison of the different models found by DnC for different values of the complexity parameter w_c

Model	Search (h)	Test Acc.	t_{tr} (s)	η	λ	N_p (M)	Depth	Emb.	Channels of convolutional layers
DPCNN		67.7	207.0	1.0e − 3	1.0e − 4	10.1	15	250	(14 layers) × (250 ch.)
$w_c = 0$	93.9	67.4	171.5	1.0e − 3	1.0e − 4	8.5	20	113	97, 127, 135, 143, 152, 295, 347, 349, 11 × (350 ch.)
$w_c = 0.01$	89.4	67.0	169.4	7.0e − 4	1.2e − 4	8.5	20	113	97, 127, 135, 143, 152, 295, 347, 349, 11 × (350 ch.)
$w_c = 0.1$	72.8	66.7	108.7	1.4e − 3	1.3e − 4	3.8	6	106	89, 151, 250, 250, 267
$w_c = 1$	50.5	65.4	58.5	1.6e − 3	1.2e − 4	4.6	3	150	137, 152
$w_c = 10$	38.9	64.3	32.3	2.2e − 3	1.5e − 4	2.3	3	76	57, 100

For each model, we report search time (h), test accuracy, training time t_{tr} (s), learning rate η, weight decay λ, number of parameters N_p (millions), depth, size of the embedding layer, and channels in each conv layer. All the models are trained using Adam with the same learning rate schedule; note that the DPCNN model of [21] can reach 68.2 accuracy when trained using the author's SGD method and hyperparameters

sizes require different padding of the input examples (which have variable length).

To train candidate architectures at each stage, we use at most 25 epochs, stopping when six consecutive epochs do not produce an improvement in validation accuracy. To avoid overfitting, we also use early stopping: models are selected using their best (instead of final) validation accuracy. The selected model is then evaluated on the test dataset.

We explored different tradeoffs between accuracy and model complexity in our search space using different values of w_c, the weight of model complexity in Eq. (1); in particular, we used $w_c \in \{0, 0.01, 0.1, 1, 10\}$ and adopted model training time t_{tr} as a complexity metric. The results, presented in Table 1 and Fig. 5, clearly illustrate these tradeoffs: as w_c increases, the framework selects models that have lower test accuracy and lower training time (our complexity metric). Note that search cost decreases similarly to training time, while the number of model parameters N_p is higher when $w_c = 1$ rather than $w_c = 0.1$. In fact, the model selected for $w_c = 1$ has a higher number of parameters but lower training time than the one of $w_c = 0.1$, because more parameters are assigned to the embedding layer (a fully-connected layer, which results in less computation per parameter than conv layers). We also observe from Table 1 that model depth is reduced from 20 to 3 when w_c increases, and DnC selects a higher learning rate η for these shallow models.

Table 1 also reports the DPCNN architecture and hyperparameters introduced in Ref. [21]: this model, with 15 layers and 10.1 million parameters, achieves best test accuracy in our experiments (67.7%) but also incurs the longest per-epoch training time (207 s). With $w_c = 0$, DnC finds a model with similar accuracy but 17% lower per-epoch training time (171.5 s), fewer parameters and higher depth (20); with $w_c = 0.1$, the reduction of test accuracy to 66.7% allows reducing per-epoch training time to 108.7 s. With $w_c = 1$, DnC achieves test accuracy of 65.4% and per-epoch training time of 58.5 s: ShallowCNNs [20] achieve similar accuracy to this model (65.8%) by also using only two conv layers but with more parameters overall (7.8M instead of 4.6M). Also note that, while [21] used unsupervised embeddings (trained with additional data) and manually tuned SGD to improve accuracy, all results in Table 1 use Adam with the same learning rate schedule, without unsupervised embeddings. In future efforts, we plan to further optimize training strategies in Stage 3.

We also experimented with the easier Yelp-2 dataset, where reviews are only labeled with two classes (positive for 1–2 stars, negative for 3–4); 560k reviews are present for training (45k reserved for validation) and 38k for testing. Figure 6 illustrates the results of model search for multiple values of w_c, using the same search space, but training each model only for 20 epochs (as this was sufficient for the easier classification task). We observe a similar trend, where higher values of w_c allow DnC to select models where test accuracy is decreased to reduce the training time t_{tr} (our complexity metric). While test accuracies are all within a margin of 0.6%, training time

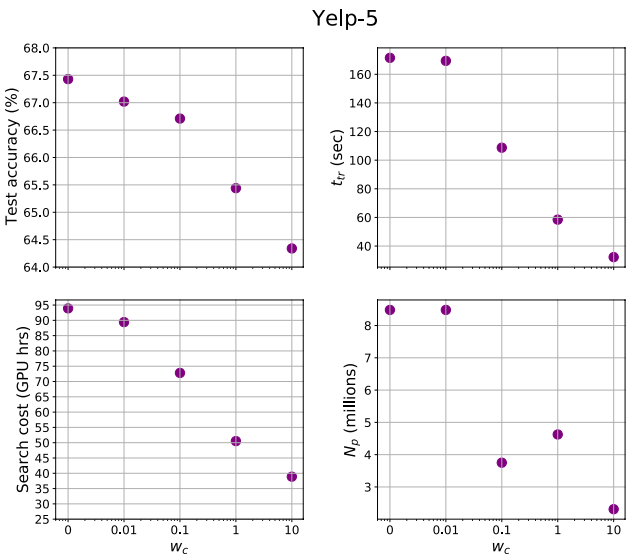

Fig. 5 Characterizing a family of NLP models for sentiment analysis on the Yelp-5 dataset, for different values of w_c (complexity weight)

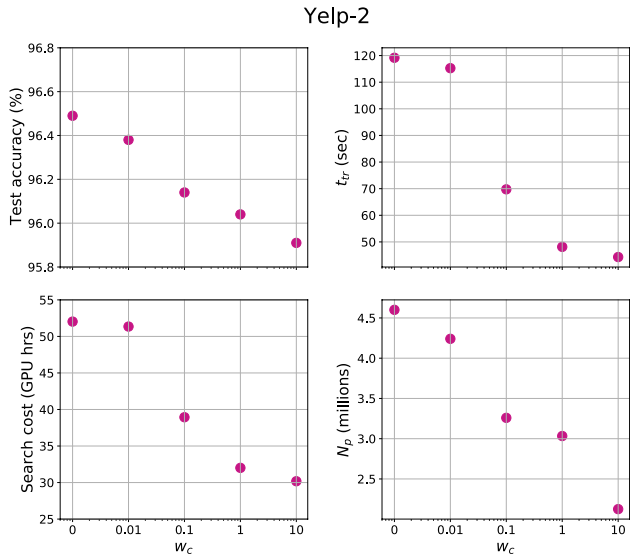

Fig. 6 Characterizing a family of NLP models for sentiment analysis on the Yelp-2 dataset, for different values of w_c (complexity weight). Note that, the best accuracy reported on this dataset in Ref. [21] without use of unsupervised embedding is 96.7% which is very close to our best result (wc = 0, accuracy = 96.5%); however, the model identified by DnC has fewer layers (10 layers) and parameters as compared to the model in Ref. [21] (15 layers)

and model parameters are reduced by more than half when $w_c = 10$.

Investigations and Insights

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' configurations, i.e., those searched for through all three stages on dataset B. This process is shown on the left in Fig. 7. Note 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 Ref. [42].

We took the best CNN architectures found from searches on CIFAR-10, CIFAR-100 and FMNIST (as depicted

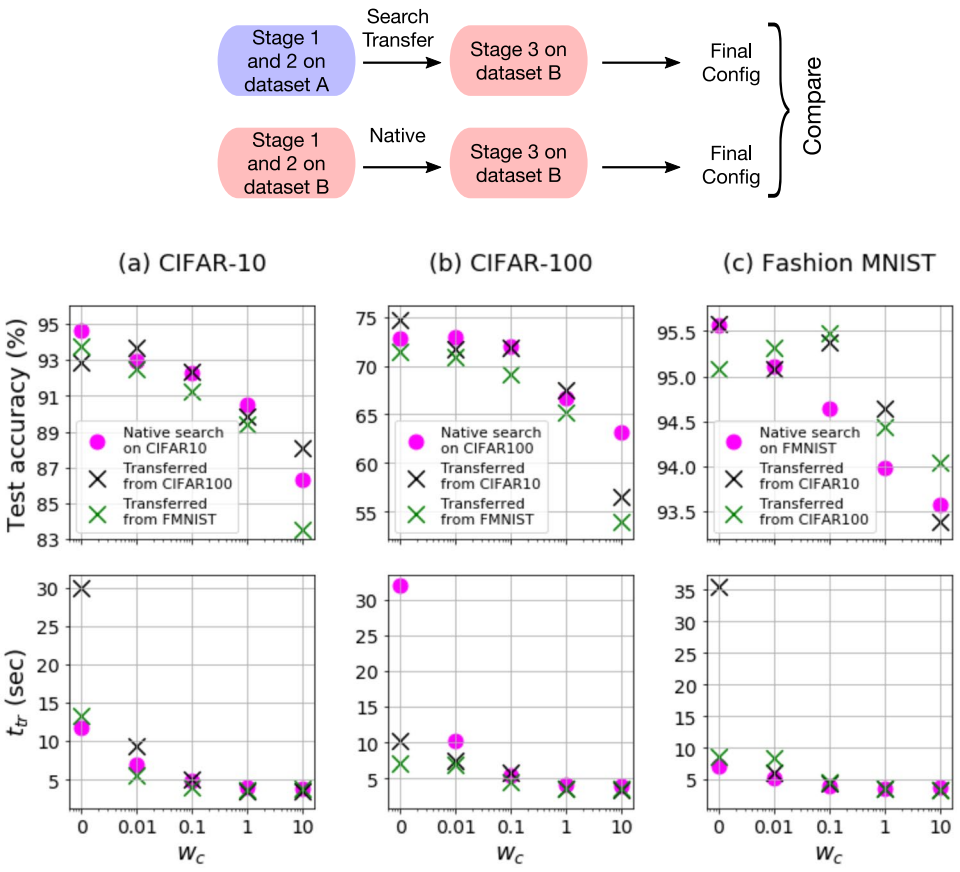

Fig. 7 Top: Process of search transfer: comparing configurations obtained from native search with those where Stage 3 is done on a dataset different from Stages 1 and 2. Bottom: Results of CNN search transfer to **a** CIFAR-10, **b** CIFAR-100, **c** FMNIST. All datasets are augmented. Pink dots denote native search

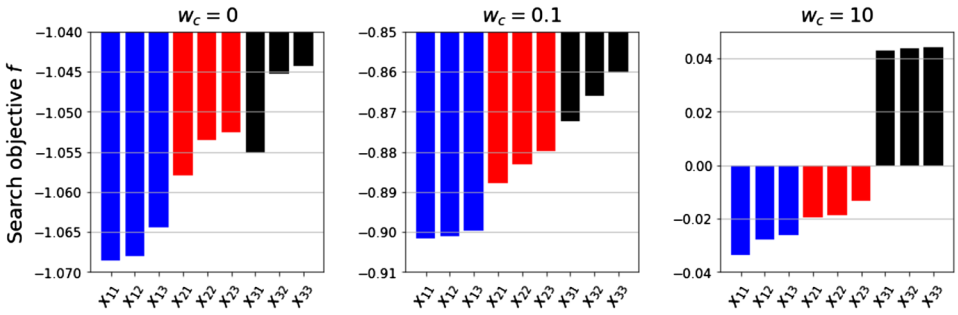

Fig. 8 Search objective values (lower the better) for three best configurations from Stage 1 (blue, red, black), optimized through Stages 2 and 3 and three best configurations chosen for each in Stage 3. Results shown for different w_c on CIFAR-10 unaugmented

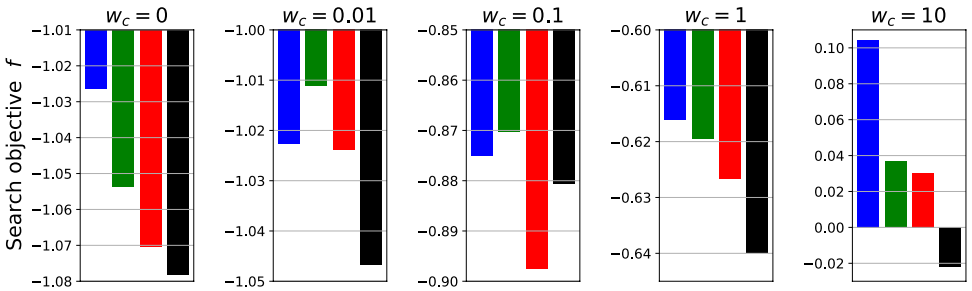

Fig. 9 Search objective values (lower the better) for purely random search (30 samples, blue) vs purely grid search via Sobol sequencing (30 samples, green) vs balanced BO (15 initial samples, 15 optimized samples, red) vs extreme BO (1 initial sample, 29 optimized samples, black). Results shown for different w_c on CIFAR-10 unaugmented, averaged over two runs

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. 7. 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.

Greedy Strategy

Our search methodology is greedy in the sense that it keeps, at the end of each stage and substage, only the configuration with minimum f value in Eq. (1). We also experimented with a non-greedy strategy: instead of one, we picked the three best configurations 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 configurations at the end of Stage 2, and finally picked the three best configurations for each of their Stage 3 runs for a total of nine configurations, $\{\boldsymbol{x}_{11}, \boldsymbol{x}_{12}, \boldsymbol{x}_{13}, \boldsymbol{x}_{21}, \ldots, \boldsymbol{x}_{33}\}$. Following a purely greedy 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 configuration for five different values of w_c on CIFAR-10 unaugmented (Fig. 8 shows three of these w_c values). In each case, we found that following a purely greedy approach yielded best results, which justifies our choice of a gready strategy in DnC.

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 of BO by comparing the final search loss f achieved by performing the Stage 1 and 3 searches in four different ways:

- Random search: pick 30 prior points randomly, no optimization steps.
- Grid search: pick 30 prior points via Sobol sequencing, no optimization steps.
- Balanced BO (DnC default): pick 15 prior points via Sobol sequencing, 15 optimization steps.
- Extreme BO: pick 1 initial point, 29 optimization steps.

Table 2 Comparison of features of AutoML frameworks

Framework	Architecture search space	Training hyp search	Adjust model complexity
Auto-Keras	Pre-existing architectures	No	No
AutoGluon	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. 9 are for different w_c on CIFAR-10, averaged over multiple runs. BO outperforms random and grid search on each occasion. In most cases, more optimization steps are beneficial (black bar).

Extensibility of Deep-n-Cheap

One of the virtues of the Deep-n-Cheap framework is the ability to adapt to new datasets and model architectures. While our initial experiments focused on CNNs and MLPs for image classification [11], we were able to quickly extend our search to NLP models for sentiment analysis. This could be achieved quite easily due to the flexibility of DnC, as follows:

- Support for the Yelp-2 and Yelp-5 datasets was included as an input parameter by following the same conventions used for other datasets, i.e., by preparing a Python dictionary with train, test and validation data loaders.
- Since the size of the embedding layer was added to the search space, a new metric was necessary to evaluate model similarities; we were able to easily define this metric from the existing layer distance metrics already implemented in DnC for MLPs and CNNs.
- Finally, by implementing a new search strategy in DnC, we were able to alter the search space to include only models where (1) the first layer is an embedding, (2) subsequent layers use stride-2 max-pooling (for downsampling) and 1D stride-1 convolutions, (3) batch normalization is not used. To change the search bounds for the number of layers, convolutional channels and fully-connected units, we were able to use the command-line options provided by DnC.

Given the differences between image classification and sentiment analysis datasets and architectures, we believe that these changes represent the minimum amount of work required from the user of an AutoML framework.

Comparison to Related Work

Table 2 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

Table 3 Comparing AutoML results (CNNs, CIFAR-10, augmented)

Framework	Additional settings	Search cost (GPU h)	Best model found from search			
			Architecture	t_{tr} (s)	Batch size	Best val acc (%)
ProxylessNAS[a]	Proxyless-G	96	537 conv layers	429	64	93.22
Auto-Keras[b]	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

Bold values in a column indicate best results

[a]Since ProxylessNAS is a search methodology as opposed to an AutoML framework, we trained the final best model provided to us by the authors [7]. This model was trained in Ref. [9] using stochastic depth and additional cutout augmentation [7]—yielding an impressive 97.92% accuracy on their test set. The result shown here was obtained without cutout or stochastic depth, and the validation accuracy is reported to compare with the metrics available from Auto-Keras and AutoGluon. 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

[b]Auto-Keras does not support image augmentation at the time of writing this paper [18], so we used an unaugmented dataset

Table 4 Comparing AutoML results (MLPs, FMNIST/RCV1)

Framework	Additional settings	Search cost (GPU h)	Best model found from search				
			MLP layers	N_p	t_{tr} (s)	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	$w_c = 0$	0.52	3	263k	0.4	272	90.24
(penalize t_{tr})	$w_c = 10$	**0.3**	1	**7.9k**	**0.1**	511	84.39
Deep-n-Cheap	$w_c = 0$	0.44	2	317k	0.5	153	90.53
(penalize N_p)	$w_c = 10$	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	$w_c = 0$	1.83	2	1.32M	0.7	503	**91.36**
(penalize t_{tr})	$w_c = 1$	**1.25**	1	**100k**	**0.4**	512	90.34
Deep-n-Cheap	$w_c = 0$	2.22	2	1.6M	0.6	512	**91.36**
(penalize N_p)	$w_c = 1$	1.85	1	**100k**	5.54	33	90.4

Bold values in a column indicate best results

is shown in Table 3, which compares the search process and metrics of the final model found for CNNs on CIFAR-10, and Table 4, 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 .

Note that 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, to have a fair comparison, Tables 3 and 4 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. Auto-Keras and AutoGluon use fixed batch sizes across all models, however, Auto-PyTorch and DnC also do a search over batch sizes. We have included batch size since it affects t_{tr}. Each configuration for each search is run for the same number of epochs, as described in "Experimental Results". The exception is Auto-PyTorch, where a key feature is variable number of epochs.

We note that for CNNs, DnC results in both the fastest t_{tr} and highest performance. The performance of Proxyless-NAS is comparable, while taking 43× more time to train. This highlights one of our key features—the ability to find models with 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 three configurations. 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 RCV1 dataset.

Conclusion and Future Work

In this paper, we described 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 AutoML and search frameworks in literature. We also found that it is possible to transfer a family of architectures found using different w_c values between different datasets without performance degradation. 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. We also demonstrated how DnC can be extended successfully for different datasets and model architectures, illustrating our process on NLP models for sentiment analysis.

DnC currently supports image and text classification using CNNs, MLPs and embedding layers. Our future plans are to extend to other types of networks such as recurrent and other applications of deep learning such as

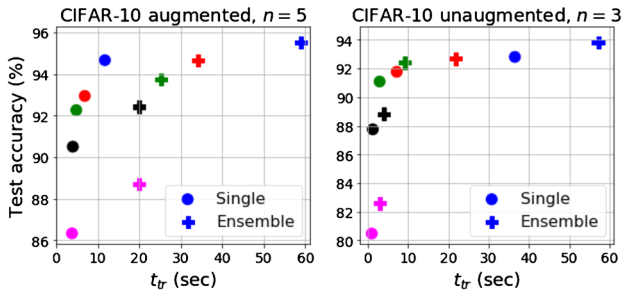

Fig. 10 Performance-complexity tradeoff for single configurations (circles) vs ensemble of configurations (pluses) for $w_c = 0$ (blue), 0.01 (red), 0.1 (green), 1 (black), 10 (pink). Results using ensemble of 5 for CIFAR-10 augmented, and 3 for CIFAR-10 unaugmented

segmentation, which would also require expanding 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 users. The framework can be found at https://github.com/usc-hal/deep-n-cheap.

Appendix: Validity of Our Covariance Kernel

The validity of our covariance kernel can be proved as follows. We note that since x_{ik} and x_{jk} are scalars, d in Eq. (2) is the Euclidean distance. It follows from the properties of the squared exponential kernel that $\sigma(x_{ik}, x_{jk})$ in Eq. (3) is a valid kernel function. So if a kernel matrix $\boldsymbol{\Sigma_k}$ were to be formed such that $\Sigma_{k_{ij}} = \sigma(x_{ik}, x_{jk})$, then $\boldsymbol{\Sigma_k}$ would be positive semi-definite. Writing Eq. (4) in matrix form gives $\boldsymbol{\Sigma} = \sum_{k=1}^{K} s_k \boldsymbol{\Sigma_k}$. Since a convex combination of positive semi-definite matrices is also positive semi-definite, it follows that $\boldsymbol{\Sigma}$ is a valid covariance matrix.

Appendix: Ensembling

One way to increase performance such as test accuracy is by having an ensemble of multiple networks vote on the test set. This comes at a complexity cost since multiple NNs need to be trained. We experimented on ensembling by taking the n best networks from BO in Stage 3 of our search. Note that this does not increase the search cost as long as $n \leq n_1 + n_2$. However, it does increase the effective number of parameters by a factor of exactly n (since each of the n best configurations have the same architecture), and t_{tr} by some indeterminate factor (since each of the n best configuration might have a different batch size).

We experimented on CIFAR-10 unaugmented using $n = 3$ and augmented using $n = 5$. The impact on the performance-complexity tradeoff is shown in Fig. 10. Note how the plus markers—ensemble results—have slightly better performance at the cost of significantly increased complexity as compared to the circles—single results. However, we did not use ensembling in other experiments since the slight increases in accuracy do not usually justify the significant increases in t_{tr}.

Appendix: Changing Hyperparameters of Bayesian Optimization

The BO process itself has several hyperparameters that can be customized by the user, or optimized using marginal likelihood or Markov chain Monte Carlo methods [34]. This section describes the default values we used. Expected improvement involves an exploration–exploitation tradeoff variable ξ. The recommended default is $\xi = 0.01$ [6], however, we tried different values and empirically found $\xi = 10^{-4}$ to work well. Secondly, f is a noisy function since the computed values of network performance are noisy due to random initialization of weights and biases for each new state. Accordingly, and also considering numerical stability for the matrix inversions involved in BO, our algorithm incorporates a noise term σ_n^2. We calculated its value from the variance in f values as $\sigma_n^2 = 10^{-4}$, which worked well compared to other values we tried.

Appendix: Adaptation to Various Platforms

While most deep NNs are run on GPUs, situations may arise where GPUs are not readily or freely available and it is desirable to run simpler experiments such as MLP training on CPUs. DnC can adapt its penalty metrics to any platform. For example, the FMNIST results shown in Fig. 4 were on CPU, while Table 4 shows results on GPU (to do a fair comparison with other frameworks). As a result, the t_{tr} values are an order of magnitude faster, while the performance is the same as expected.

Acknowledgements While all of the DnC results presented in this paper used the PyTorch framework, the DnC codebase has been updated to support TensorFlow for the image classification (CNN) and MLP applications. This feature was added by USC MSEE students Ziping Chen, Zixuan He, and Tianyi Zhang as a class project in the spring 2019 USC class EE599: Special Topics: Deep Learning Systems.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. AWSLabs: AutoGluon: AutoML toolkit for deep learning. https://autogluon.mxnet.io/# (2020). Accessed 1 Jul 2020.
2. Baker B, Gupta O, Naik N, Raskar R. Designing neural network architectures using reinforcement learning. In: Proc. ICLR, 2017.
3. Baker B, Gupta O, Raskar R, Naik N. Accelerating neural architecture search using performance prediction. In: Proc. ICLR, 2017.
4. Baldi P, Sadowski P, Whiteson D. Searching for exotic particles in high-energy physics with deep learning. Nat Commun. 2014;5:4308.
5. Bergstra J, Yamins D, Cox DD. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proc. ICML, 2013; pp. I-115–I-123.
6. Brochu E, Cora VM, de Freitas N. 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.
7. Cai H, Dey S. Private communication with ProxylessNAS authors. 2020. We (the authors of this paper) corresponded with the authors of ProxylessNAS via email in March 2020.
8. Cai H, Dey S. Private communication with ProxylessNAS authors. 2020. We (the authors of this paper) corresponded with the authors of ProxylessNAS via email in March 2020.
9. Cai H, Zhu L, Han S. ProxylessNAS: direct neural architecture search on target task and hardware. In: Proc. ICLR, 2019.
10. Dey S, Huang KW, Beerel PA, Chugg KM. Pre-defined sparse neural networks with hardware acceleration. IEEE JETCAS. 2019;9(2):332–45.
11. Dey S, Kanala SC, Chugg KM, Beerel PA. Deep-n-Cheap: an automated search framework for low complexity deep learning. In: Proc. ACML, 2020; pp. 273–288.
12. Dey S, Shao Y, Chugg K, Beerel P. Accelerating training of deep neural networks via sparse edge processing. In: Proc. ICANN, 2017; pp. 273–280.
13. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385. 2015.
14. He K, Zhang X, Ren S, Sun J. Identity mappings in deep residual networks. CoRR abs/1603.05027. 2016.
15. He Y, Lin J, et al. AMC: AutoML for model compression and acceleration on mobile devices. In: Proc. ECCV, 2018; pp. 784–800.
16. Huang G, Sun Y, et al. Deep networks with stochastic depth. In: Proc. ECCV, 2016; pp. 646–661.
17. Hutter F, Osborne MA. A kernel for hierarchical parameter spaces. arXiv preprint arXiv:1310.5738. 2013.
18. Jin H. Comment on 'not able to load best automodel after saving' issue. https://github.com/keras-team/autokeras/issues/966#issuecomment-594590617 (2019). Accessed 1 Apr 2020.
19. Jin H, Song Q, Hu X. Auto-keras: an efficient neural architecture search system. In: Proc. KDD, 2019; pp. 1946–1956.
20. Johnson R, Zhang T. Effective use of word order for text categorization with convolutional neural networks. In: Proc. NAACL, 2015; pp. 103–112.
21. Johnson R, Zhang T. Deep pyramid convolutional neural networks for text categorization. In: Proc. ACL, 2017; pp. 562–570.
22. Kandasamy K, Neiswanger W, et al. Neural architecture search with Bayesian optimisation and optimal transport. In: Proc. NeurIPS, 2018; pp. 2020–2029.
23. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Proc. NeurIPS, 2012; pp. 1097–1105.
24. Liu C, Zoph B, et al. Progressive neural architecture search. In: Proc. ECCV, 2018; pp. 19–35.
25. Liu H, Simonyan K, Yang Y. DARTS: differentiable architecture search. In: Proc. ICLR, 2019.
26. Mayo RC, Kent D, et al. Reduction of false-positive markings on mammograms: a retrospective comparison study using an artificial intelligence-based CAD. J Dig Imaging. 2019;32:618–24.
27. Mendoza H, Klein A, et al. Towards automatically-tuned deep neural networks. In: AutoML: methods, systems, challenges, chap. 7, 2018; pp. 141–156.
28. Miikkulainen R, Liang J, et al. Evolving deep neural networks. In: Artificial intelligence in the age of neural networks and brain computing, chap. 15, pp. 293 – 312. Academic Press; 2019.
29. Page D. How to train your ResNet. https://myrtle.ai/how-to-train-your-resnet/ (2019). Accessed 1 Apr 2020.
30. Pham H, Guan M, et al. Efficient neural architecture search via parameter sharing. In: Proc. ICML, 2018; pp. 4095–4104.
31. Real E, Aggarwal A, Huang Y, Le QV. Regularized evolution for image classifier architecture search. In: Proc. AAAI, 2019; pp. 4780–4789.
32. Santana E, Hotz G. Learning a driving simulator. arXiv preprint arXiv:1608.01230. 2016.
33. Snoek J, Larochelle H, Adams RP. Practical Bayesian optimization of machine learning algorithms. In: Proc. NeurIPS, 2012; pp. 2951–2959.
34. Swersky K, Duvenaud D 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.
35. Tan M, Le QV. Efficientnet: rethinking model scaling for convolutional neural networks. arXiv preprint arXiv:1905.11946. 2019.
36. Thornton C, Hutter F, Hoos HH, Leyton-Brown K. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Proc. KDD, 2013; pp. 847—-855.
37. Xie L, Yuille A. Genetic CNN. In: Proc. ICCV, 2017; pp. 1388–1397.
38. Xie S, Zheng H, Liu C, Lin L. SNAS: stochastic neural architecture search. In: Proc. ICLR, 2019.
39. Yang Z, Dai Z, Yang Y, Carbonell JG, Salakhutdinov R, Le QV. XLNet: generalized autoregressive pretraining for language understanding. In: NIPS, 2019; pp. 5754–5764.
40. Zagoruyko S, Komodakis N. Wide residual networks. In: Proc. BMVC, 2016; pp. 87.1–87.12.
41. Zhang X, Zhao JJ, LeCun, Y. Character-level convolutional networks for text classification. arXiv preprint arXiv:1509.01626. 2015.
42. Zoph B, Vasudevan V, Shlens J, Le QV. Learning transferable architectures for scalable image recognition. In: Proc. CVPR, 2018; pp. 8697–8710.
43. Kundu S, Nazemi M, Pedram M, Chugg KM, Beerel PA. Pre-Defined Sparsity for Low-Complexity Convolutional Neural Networks. IEEE Transactions on Computers. 2020;69(7):1045–58.