Evolutionary Neural AutoML for Deep Learning

Jason Liang, Elliot Meyerson, Babak Hodjat, Dan Fink, Karl Mutch, and Risto Miikkulainen[*]

Cognizant Technology Solutions
The University of Texas at Austin

ABSTRACT

Deep neural networks (DNNs) have produced state-of-the-art results in many benchmarks and problem domains. However, the success of DNNs depends on the proper configuration of its architecture and hyperparameters. Such a configuration is difficult and as a result, DNNs are often not used to their full potential. In addition, DNNs in commercial applications often need to satisfy real-world design constraints such as size or number of parameters. To make configuration easier, automatic machine learning (AutoML) systems for deep learning have been developed, focusing mostly on optimization of hyperparameters.

This paper takes AutoML a step further. It introduces an evolutionary AutoML framework called LEAF that not only optimizes hyperparameters but also network architectures and the size of the network. LEAF makes use of both state-of-the-art evolutionary algorithms (EAs) and distributed computing frameworks. Experimental results on medical image classification and natural language analysis show that the framework can be used to achieve state-of-the-art performance. In particular, LEAF demonstrates that architecture optimization provides a significant boost over hyperparameter optimization, and that networks can be minimized at the same time with little drop in performance. LEAF therefore forms a foundation for democratizing and improving AI, as well as making AI practical in future applications.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; *Distributed artificial intelligence*; Computer vision;

KEYWORDS

Neural Networks/Deep Learning; Artificial Intelligence; AutoML

1 INTRODUCTION

Applications of machine learning and artificial intelligence have increased significantly recently, driven by both improvements in computing power and quality of data. In particular, deep neural networks (DNN) [29] learn rich representations of high-dimensional data, exceeding the state-of-the-art in an variety of benchmarks in computer vision, natural language processing, reinforcement learning, and speech recognition [16, 21, 23]. Such state-of-the-art DNNs are very large, consisting of hundreds of millions of parameters, requiring large computational resources to train and run. They are also highly complex, and their performance depends on their architecture and choice of hyperparameters [13, 23, 39].

Much of the recent research in deep learning indeed focuses on discovering specialized architectures that excel in specific tasks. There is much variation between DNN architectures (even for single-task domains) and so far, there are no guiding principles for deciding between them. Finding the right architecture and hyperparameters is essentially reduced to a black-box optimization process. However, manual testing and evaluation is a tedious and time consuming process that requires experience and expertise. The architecture and hyperparameters are often chosen based on history and convenience rather than theoretical or empirical principles, and as a result, the network has does not perform as well as it could. Therefore, automated configuration of DNNs is a compelling approach for three reasons: (1) to find innovative configurations of DNNs that also perform well, (2) to find configurations that are small enough to be practical, and (3) to make it possible to find them without domain expertise.

Currently, the most common approach to satisfy the first goal is through partial optimization. The authors might tune a few hyperparameters or switch between several fixed architectures, but rarely optimize both the architecture and hyperparameters simultaneously [24, 49]. This approach is understandable since the search space is massive and existing methods do not scale as the number of hyperparameters and architecture complexity increases. The standard and most widely used methods for hyperparameter optimization is grid search, where hyperparameters are discretized into a fixed number of intervals and all combinations are searched exhaustively. Each combination is tested by training a DNN with those hyperparameters and evaluating its performance with respect to a metric on a benchmark dataset. While this method is simple and can be parallelized easily, its computational complexity grows combinatorially with the number of hyperparameters, and becomes intractable once the number of hyperparameters exceeds four or five [27]. Grid search also does not address the question of what the optimal architecture of the DNN should be, which may be just as important as the choice of hyperparameters. A method that can optimize both structure and parameters is needed.

Recently, commercial applications of deep learning have become increasingly important and many of them run on smartphones. Unfortunately, the hundreds of millions of weights of modern DNNs cannot fit to the few gigabytes of RAM in most smartphones. Therefore, an important second goal of DNN optimization is to minimize the complexity or size of a network, while simultaneously maximizing its performance [25]. Thus, a method for optimizing multiple objectives is needed to meet the second goal.

[*]firstname.lastname@cognizant.com

In order to achieve the third goal, i.e. democratizing AI, systems for automating DNN configuration have been developed, such as Google AutoML [1] and Yelp's Metric Optimization Engine (MOE [5], also commercialized as a product called SigOpt [8]). However, existing systems are often limited in both the scope of the problems they solve and how much feedback they provide to the user. For example, Google AutoML system is a black-box that hides the network architecture and training from the user; it only provides an API by which the user can use to query on new inputs. MOE is more transparent on the other hand, but since it uses a Bayesian optimization algorithm underneath, it only tunes hyperparameters of a DNN. Neither systems minimizes the size or complexity of the networks.

The main contribution of this paper is a novel AutoML system called LEAF (Learning Evolutionary AI Framework) that addresses these three goals. LEAF leverages and extends an existing state-of-the-art evolutionary algorithm for architecture search called CoDeepNEAT [35], which evolves both hyperparameters and network structure. While its hyperparameter optimization ability already matches those of other AutoML systems, LEAF's additional ability to optimize DNN architectures further makes it possible to achieve state-of-the-art results. The speciation and complexification heuristics inside CoDeepNEAT also allows it to be easily adapted to multiobjective optimization to find minimal architectures. The effectiveness of LEAF will be demonstrated in this paper on two domains, one in language: Wikipedia comment toxicity classification (also referred to as Wikidetox), and another in vision: Chest X-rays multitask image classification. LEAF therefore forms a foundation for democratizing, simplifying, and improving AI.

2 BACKGROUND AND RELATED WORK

This section will review background and related work in hyperparameter optimization and neural architecture search.

2.1 Hyperparameter Tuning for DNNs

As mentioned in Section 1, the simplest form of hyperparameter optimization is exhaustive grid search, where points in hyperparameter space are sampled uniformly at regular intervals [49]. A straightforward extension is random search, where the points are sampled uniformly at random from the search space [11]. These methods can optimize simple DNNs, but are ineffective when all hyperparameters are crucial to performance and must be tuned to very particular values. For networks with such characteristics, Bayesian optimization using Gaussian processes [44] is a feasible alternative. Bayesian optimization requires relatively few function evaluations and works well on multimodal, non-separable, and noisy functions where there are several local optima. It first creates a probability distribution of functions (also known as Gaussian process) that best fits the objective function and then uses that distribution to determine where to sample next. The main weakness of Bayesian optimization is that it is computational expensive and scales cubically with the number of evaluated points. DNGO [45] tried to address this issue by replacing Gaussian processes with linearly scaling Bayesian neural networks. Another downside of Bayesian optimization it performs poorly when the number of hyperparameters is moderately high, i.e. more than 10-15 [32].

EAs are another class of algorithms widely used for black-box optimization of complex, multimodal functions. They rely on biological inspired mechanisms to improve iteratively upon a population of candidate solutions to the objective function. One particular EA that has been successfully applied to DNN hyperparameter tuning is CMA-ES [32]. In CMA-ES, a Gaussian distribution for the best individuals in the population is estimated and used to generate/sample the population for the next generation. Furthermore, it has mechanisms for controlling the step-size and the direction that the population will move. CMA-ES has been shown to perform well in many real-world high-dimensional optimization problems and in particular, CMA-ES has been shown to outperform Bayesian optimization on tuning the parameters of a convolutional neural network [32]. It is however limited to continuous optimization and there does not extend naturally to architecture search.

2.2 Architecture Search for DNNs

One recent approach is to use reinforcement learning (RL) to search for better architectures. A recurrent neural network (LSTM) controller generates a sequence of layers that begin from the input and end at the output of a DNN [55]. The LSTM is trained through a gradient-based policy search algorithm called REINFORCE [51]. The architecture search space explored by this approach is sufficiently large to improve upon hand-design. On popular image classification benchmarks such as CIFAR-10 and ImageNet, such an approach achieved performance within 1-2 percentage points of the state-of-the-art, and on a language modeling benchmark, it achieved state-of-the-art performance at the time [55].

However, the architecture of the optimized network still must have either a linear or tree-like core structure; arbitrary graph topologies are outside the search space. Thus, it is still up to the user to define an appropriate search space beforehand for the algorithm to use as a starting point. The number of hyperparameters that can be optimized for each layer are also limited. Furthermore, the computations are extremely heavy; to generate the final best network, many thousands of candidate architectures have to be evaluated and trained, which requires hundreds of thousands of GPU hours.

An alternative direction for architecture search is evolutionary algorithms (EAs). They are well suited for this problem because they are black-box optimization algorithms that can optimize arbitrary structure. Some of these approaches use a modified version of NEAT [43], an EA for neuron-level neuroevolution [46], for searching network topologies. Others rely on genetic programming [47] or hierarchical evolution [31]. There is some very recent work on multiobjective evolutionary architecture search [18, 33], where the goal is to optimize both the performance and training time/complexity of the network.

The main advantage of EAs over RL methods is that they can optimize over much larger search spaces. For instance, approaches based on NEAT [43] can evolve arbitrary graph topologies for the network architecture. Most importantly, hierarchical evolutionary methods [31], can search over very large spaces efficiently and evolve complex architectures quickly from a minimal starting point. As a result, the performance of evolutionary approaches match or exceed that of reinforcement learning methods. For example, the current state-of-the-art results on CIFAR-10 and ImageNet were achieved by an evolutionary approach [42]. In this paper, LEAF uses CoDeepNEAT, a powerful EA based on NEAT that is capable of hierarchically evolving networks with arbitrary topology.
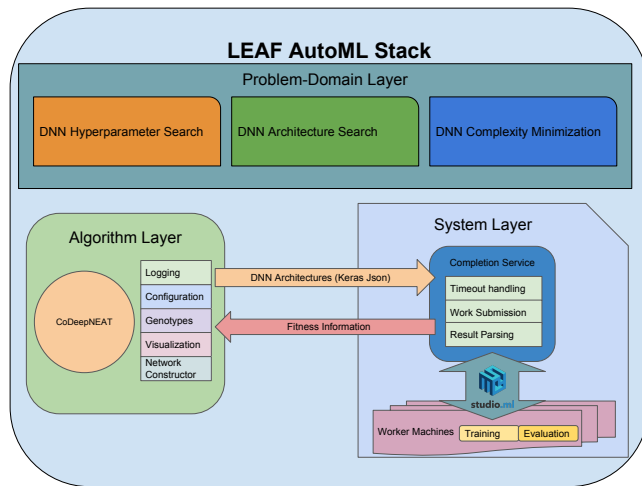
Figure 1: A visualization of LEAF and its internal subsystems. The three main components are: (1) the algorithm layer which uses CoDeepNEAT to evolve hyperparameters or neural networks, (2) the system layer which helps train and evaluate the networks evolved by the algorithm layer, and (3) the problem-domain layer, which utilizes the two previous layers to optimize DNNs. The decoupling of the algorithm and system layers allows LEAF to be easily applied to varying problem types, e.g., via options for multiobjective optimization and different types of neural network layers.

3 LEAF OVERVIEW

LEAF is an AutoML system composed of three main components: algorithm layer, system layer, and problem-domain layer. The algorithm layer allows the LEAF to evolve DNN hyperparameters and architectures. The system layer parallelizes training of DNNs on cloud compute infrastructure such as Amazon AWS [2], Microsoft Azure [6], or Google Cloud [3], which is required to evaluate the fitnesses of the networks evolved in the algorithm layer. The algorithm layer sends the network architectures in Keras JSON format [14] to the system layer and receives fitness information back. These two layers work in tandem to support the problem-domain layer, where LEAF solves problems such as hyperparameter tuning, architecture search, and complexity minimization. An overview of LEAF AutomML's structure is shown in Figure 1.

3.1 Algorithm Layer

The core of the algorithm layer is composed of CoDeepNEAT, an cooperative coevolutionary algorithm based on NEAT for evolving DNN architectures and hyperparameters [35]. Cooperative coevolution is a commonly used technique in evolutionary computation to discover complex behavior during evaluation by combining simpler components together. It has been used with success in many domains, including function optimization [40], predator-prey dynamics [53], and subroutine optimization [52]. The specific coevolutionary mechanism in CoDeepNEAT is inspired by Hierarchical SANE [38] but is also influenced by component-evolution approaches of ESP [20] and CoSyNE [19]. These methods differ from conventional neuroevolution in that they do not evolve entire networks. Instead, both approaches evolve components that are then assembled into complete networks for fitness evaluation.
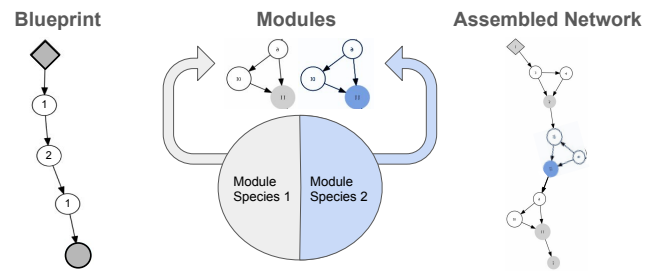

Figure 2: A visualization of how CoDeepNEAT assembles networks for fitness evaluation. Modules and blueprints are assembled together into a network through replacement of blueprint nodes with corresponding modules. This approach allows evolving repetitive and deep structures seen in many hand-designed DNNs.

Algorithm 1 CoDeepNEAT

---

1. Given population of modules/blueprints
2. For each blueprint B_i during every generation:
    3. For each node N_j in B_i
        4. Choose randomly from module species that N_j points to
        5. Replace N_j with randomly chosen module M_j
    6. When all nodes in B_i are replaced, convert B_i to assembled network N_i
7. Evaluate fitnesses of the assembled networks N
8. For each network N_i
    9. Attribute fitness of N_i to its component blueprint B_i and modules M_j
10. Evolve blueprint and module population with NEAT

CoDeepNEAT follows the same fundamental process as NEAT: First, a population of chromosomes of minimal complexity is created. Each chromosome is represented as a graph and is also referred to as an individual. Over generations, structure (i.e. nodes and edges) is added to the graph incrementally through mutation. As in NEAT, mutation involves randomly adding a node or a connection between two nodes. During crossover, historical markings are used to determine how genes of two chromosomes can be lined up and how nodes can be randomly crossed over. The population is divided into species (i.e. subpopulations) based on a similarity metric. Each species grows proportionally to its fitness and evolution occurs separately in each species.

CoDeepNEAT differs from NEAT in that each node in the chromosome no longer represents a neuron, but instead a layer in a DNN. Each node contains a table of real and binary valued hyperparameters that are mutated through uniform Gaussian distribution and random bit-flipping, respectively. These hyperparameters determine the type of layer (such as convolutional, fully connected, or recurrent) and the properties of that layer (such as number of neurons, kernel size, and activation function). The edges in the chromosome are no longer marked with weights; instead they simply indicate how the nodes (layers) are connected. The chromosome also contains a set of global hyperparameters applicable to the entire network (such as learning rate, training algorithm, and data preprocessing).

As summarized in Algorithm 1, two populations of modules and blueprints are evolved separately using mutation and crossover operators of NEAT. The blueprint chromosome (also known as an

individual) is a graph where each node contains a pointer to a particular module species. In turn, each module chromosome is a graph that represents a small DNN. During fitness evaluation, the modules and blueprints are combined to create a large assembled network. For each blueprint chromosome, each node in the blueprint's graph is replaced with a module chosen randomly from the species to which that node points. If multiple blueprint nodes point to the same module species, then the same module is used in all of them. After the nodes in the blueprint have been replaced, the individual is converted into a DNN. This entire process for assembling the network is visualized in Figure 2.

The assembled networks are evaluated by first letting the networks learn on a training dataset for the task and then measuring their performance with an unseen validation set. The fitnesses, i.e. validation performance, of the assembled networks are attributed back to blueprints and modules as the average fitness of all the assembled networks containing that blueprint or module. This scheme reduces evaluation noise and allows blueprints or modules to be preserved into the next generation even if they be occasionally included in a poorly performing network. After CoDeepNEAT finishes running, the best evolved network is trained until convergence and evaluated on another holdout testing set.

3.2 System Layer

One of the main challenges in using CoDeepNEAT to evolve the architecture and hyperparameters of DNNs is the computational power required to evaluate the networks. However, because evolution is a parallel search method, the evaluation of individuals in the population every generation can be distributed over hundreds of worker machines, each equipped with a dedicated GPU. For most of the experiments described in this paper, the workers are GPU equipped machines running on Microsoft Azure, a popular platform for cloud computing [6].

To this end, the system layer of LEAF uses the API called the completion service that is part of an open-source package called StudioML [9]. First, the algorithm layer sends networks ready for fitness evaluation in the form of Keras JSON to the system layer server node. Next, the server node submits the networks to the completion service. They are pushed onto a queue (buffer) and each available worker node pulls a single network from the queue to train. After training is finished, fitness is calculated for the network and the information is immediately returned to the server. The results are returned one at a time and without any order guarantee through a separate return queue. By using the completion service to parallelize evaluations, thousands of candidate networks are trained in a matter of days, thus making architecture search tractable.

3.3 Problem-Domain Layer

The problem-domain layer solves the three tasks mentioned earlier, i.e. optimization of hyperparameters, architecture, and network complexity, using CoDeepNEAT is a starting point.

Hyperparameter Optimization. By default, LEAF optimizes both architecture and hyperparameters. To demonstrate the value of architecture search, it is possible to configure CoDeepNEAT in the algorithm layer to optimize hyperparameters only. In this case, mutation and crossover of network structure and node-specific hyperparameters are disabled. Only the global set of hyperparameters contained in each chromosome are optimized, as in the case in other hyperparameter optimization methods. Hyperparameter-only

---

Algorithm 2 Multiobj CoDeepNEAT Module/Blueprint Ranking

1. Given population of modules/blueprints, evaluated primary and secondary objectives (X and Y)
2. For each species S_i during every generation:
    3. Create new empty species $\hat{S}_i$
    4. While S_i is not empty
        5. Determine Pareto front of S_i by passing X_i and Y_i to Alg. 3
        6. Remove individuals in Pareto front of S_i and add to $\hat{S}_i$
    7. Replace S_i with $\hat{S}_i$
    8. Truncate $\hat{S}_i$ by removing the last fraction F_l
    8. Generate new individuals using mutation/crossover
    9. Add new individuals to $\hat{S}_i$, proceed as normal

Algorithm 3 Multiobj CoDeepNEAT Pareto Front Calculation

1. Given list of individuals I, and corresponding objective fitnesses X and Y
2. Sort I in descending order by first objective fitnesses X
3. Create new Pareto front PF with first individual I_0
4. For each individual I_i, $i > 0$
    5. If Y_i is greater than the Y_j, where I_j is last individual in PF
        6. Append I_i to PF
7. Sort PF in descending order by second objective Y (Optional)

CoDeepNEAT is very similar to a conventional genetic algorithm in that there is elitist selection and the hyperparameters undergo uniform mutation and crossover. However, it still has NEAT's speciation mechanism, which protects new and innovative hyperparameters by grouping them into subpopulations.

Architecture Search. LEAF directly utilizes standard CoDeepNEAT to perform architecture search in simpler domains such as single-task classification. However, LEAF can also be used to search for DNN architectures for multitask learning (MTL). The foundation is the soft-ordering multitask architecture [34] where each level of a fixed linear DNN consists of a fixed number of modules. These modules are then used to a different degree for the different tasks. LEAF extends this MTL architecture by coevolving both the module architectures and the blueprint (routing between the modules) of the DNN [30].

DNN Complexity Minimization with Multiobjective Search. In addition to adapting CoDeepNEAT to multiple tasks, LEAF also extends it to multiple objectives. In a single-objective evolutionary algorithm, elitism is applied to both the blueprint and the module populations. The top fraction F_l of the individuals within each species is passed on to the next generation as in single-objective optimization. This fraction is based simply on ranking by fitness. In the multiobjective version of CoDeepNEAT, the ranking is computed from successive Pareto fronts [17, 54] generated from the primary and secondary objectives.

Algorithm 2 details this calculation for the blueprints and modules; assembled networks are also ranked similarly. Algorithm 3 shows how the Pareto front, which is necessary for ranking, is calculated given a group of individuals that have been evaluated for each objective. There is also an optional configuration parameter for multiobjective CoDeepNEAT that allows the individuals within each Pareto front to be sorted and ranked with respect to the secondary objective instead of the primary one. Although the primary objective, i.e performance, is usually more important, this parameter can be used to emphasize the secondary objective more, if necessary for a particular domain.

Thus, multiobjective CoDeepNEAT can be used to maximize the performance and minimize the complexity of the evolved networks simultaneously. While performance is usually measured as the loss on the unseen set of samples, there are many ways to characterize how complex a DNN is. They include the number of parameters, the number of floating point operations (FLOPS), and the training/inference time of the network. The most commonly used metric is number of parameters because other metrics can change depending on the deep learning library implementation and performance of the hardware. In addition, this metric is becoming increasingly important in mobile applications as mobile devices are highly constrained in terms of memory and require networks with as high performance per parameter ratio as possible [25]. Thus, number of parameters is used as the secondary objective for multiobjective CoDeepNEAT in the experiments in the following section. Although the number of parameters can vary widely across architectures, such variance does not pose a problem for multiobjective CoDeepNEAT since it only cares about the relative rankings between different objective values and no scaling of the secondary objective is required.

4 EXPERIMENTAL RESULTS

LEAF's ability to democratize AI, improve the state-of-the-art, and minimize solutions is verified experimentally on two difficult real-world domains: (1) Wikipedia comment toxicity classification and (2) Chest X-rays multitask image classification. The performance and efficiency of LEAF are compared against existing AutoML systems.

4.1 Wikipedia Comment Toxicity Classification Domain

Wikipedia is one of the largest encyclopedias that is publicly available online, with over 5 million written articles for the English language alone. Unlike traditional encyclopedias, Wikipedia can be edited by any user who registers an account. As a result, in the discussion section for some articles, there are often vitriolic or hateful comments that are directed at other users. These comments are commonly referred to as "toxic" and it has become increasingly important to detect toxic comments and remove them. The Wikipedia Detox dataset (Wikidetox) is a collection of 160K example comments that are divided into 93K training, 31K validation, and 31K testing examples [7]. The labels for the comments are generated by humans using crowd-sourcing methods and contain four different categories for toxic comments. However, following previous work [15], all toxic comment categories are combined, thus creating a binary classification problem. The dataset is also unbalanced with only about 9.6% of the comments actually being labeled as toxic.

LEAF was configured to use standard CoDeepNEAT to search for well performing architectures in this domain. The search space for these architectures was defined using recurrent (LSTM) layers as the basic building block. Since comments are essentially an ordered list of words, recurrent layers (having been shown to be effective at processing sequential data [36]) were a natural choice. In order for the words to be given as input into a recurrent network, they must be converted into an appropriate vector representation first. Before given as input to the network, the comments were preprocessed using FastText, a recently introduced method for generating word embeddings [12] that improves upon the more commonly used Word2Vec [37]. Each evolved DNN was trained for three epochs and the classification accuracy on the testing set was returned

as the fitness. Preliminary experiments showed that three epochs of training was enough for the network performance to converge. Thus, unlike vision domains such as Chest X-rays, there was no need for an extra step after evolution where the best evolved network was trained extensively from scratch. The training and evaluation of networks at every generation were distributed over 100 worker machines. For more information regarding evolution configuration and search space explored, refer to Tables 2 and 3, respectively.

The Wikidetox domain was part of a Kaggle challenge, and as a result, there already exists several hand-designed networks for the domain [4]. Furthermore, due to the relative speed at which networks can be trained on this dataset, it was practical to evaluate hyperparameter optimization methods from companies such as Microsoft, Google, and MOE on this dataset. Figure 3 shows a comparison of LEAF architecture search against several other approaches. The first one is the baseline network from the Kaggle competition, illustrating performance that a naive user can expect by applying a standard architecture to a new problem. After spending just 35 CPU hours, LEAF found architectures that exceed that performance. The next three comparisons illustrate the power of LEAF against other AutoML systems. After about 180 hrs, it exceeded the performance of Google AutoML's text classifier optimization [1] with default compute resources. After about 1000 hrs, LEAF surpassed Microsoft's TLC library for model pipeline optimization [10], and after 2000 hrs, it exceeded MOE, a Bayesian optimization library [5] (both libraries used enough compute resources for the performance to plateau). The LEAF hyperparameter-only version performed slightly better than MOE, demonstrating the power of evolution against other optimization approaches. Finally, if the user is willing to spend about 9000 hrs of CPU time on this problem, the result is state-of-the-art performance. At that point, LEAF discovered architectures that exceed the performance of Kaggle competition winner, i.e. improve upon the best known hand-design. The performance gap between this result and the hyperparameter-only version of LEAF is also important: it shows the value added by optimizing network architectures, demonstrating that it is an essential ingredient in improving the state-of-the-art.

What is interesting about LEAF is that there are clear trade-offs between the amount of training time/money used and the quality of the results. Depending on the budget available, the user running LEAF can stop earlier to get results competitive with existing approaches (such as TLC or Google AutoML) or run it to convergence to get the best possible results. If the user is willing to spend more compute, increasingly more powerful architectures are obtained. This kind of flexibility demonstrates that LEAF is not only a tool for improving AI, but also for democratizing AI.

4.2 Chest X-rays Multitask Image Classification

Chest X-rays classification is a recently introduced MTL benchmark [41, 50]. The dataset is composed of 112,120 high resolution frontal chest X-ray images, and the images are labeled with one or more of 14 different diseases, or no diseases at all. The multi-label nature of the dataset naturally lends to an MTL setup where each disease is an individual binary classification task. Past approaches generally apply the classical MTL DNN architecture [50] and the current state-of-the-art approach uses a slightly modified version of Densenet [41], a widely used, hand-designed architecture that is competitive with the state-of-the-art on the Imagenet domain [26]. The images are divided into 70% training, 10% validation, and
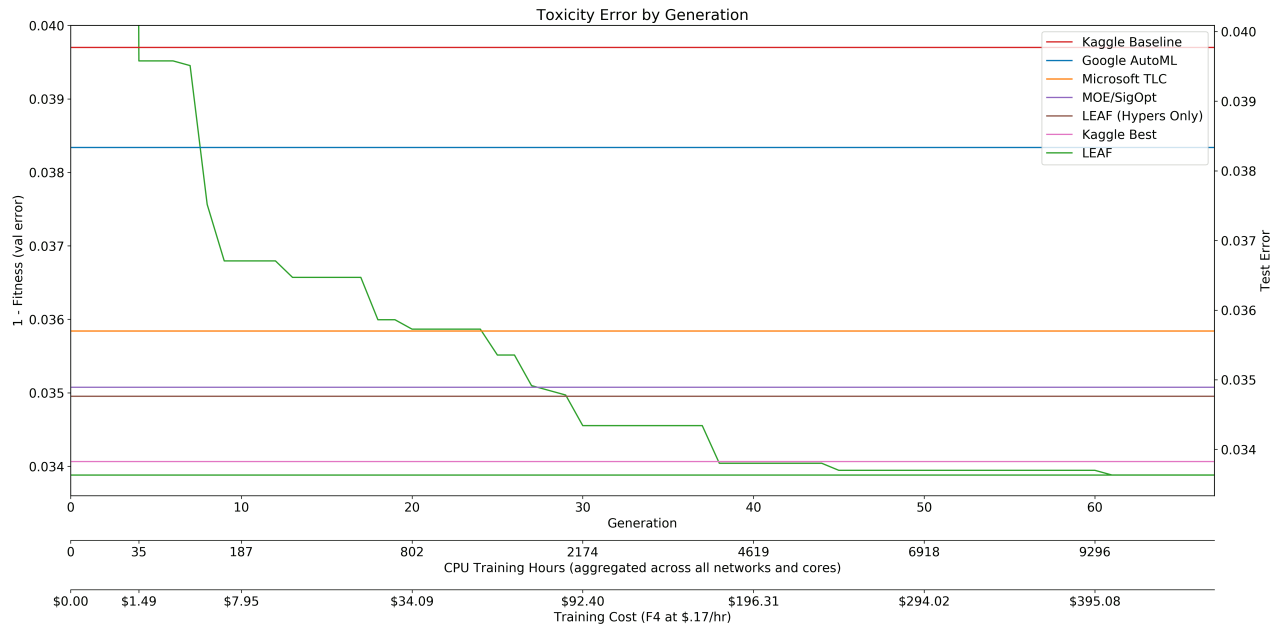
Figure 3: A comparison of LEAF in the Wikidetox domain against the networks discovered via several commercially available methods, including Keras, Google AutoML, MSFT TLC, and MOE, as well as the top human-designed network in the Kaggle comparison. The y-axis shows the best best fitness (i.e. accuracy) achieved so far, while the x-axes shows the generations, total training time, and total amount of money spent on cloud compute. As the plot shows, LEAF is gradually able to discover better networks, eventually finding one in the 40th generation that outperforms all other approaches.

20% testing. The metric used to evaluate the performance of the network is the average area under the ROC curve for all the tasks (AUROC). Although the actual images are larger, all approaches (including LEAF) preprocessed the images to be 224×224 pixels, the same input size used by many Imagenet DNN architectures.

Since Chest X-rays is a multitask dataset, LEAF was configured to use the MTL variant of CoDeepNEAT to evolve network architectures. The search space for these architectures was designed around 2D convolutional layers and includes skip connections seen in networks such as ResNet [22]. For fitness evaluations, all networks were trained using Adam [28] for eight epochs. After training was completed, AUROC was computed over all images in the validation set and returned as the fitness. No data augmentation was performed during training and evaluation in evolution, but the images were normalized using the mean and variance statistics from the Imagenet dataset. The average time for training was usually around 3-4 hours depending on the network size, although for some larger networks the training time exceeded 12 hours. Like in the Wikidetox domain, the training and evaluation were parallelized over 100 worker machines. For more information regarding evolution configuration and search space explored, refer to Tables 2 and 3, respectively.

After evolution converged, the best evolved network was trained for an increased number of epochs using the ADAM optimizer [28]. As with other approaches to neural architecture search [42, 55], the model augmentation method was used, where the number of filters of each convolutional layer was increased. Data augmentation was also applied to the images during every epoch of training, including random horizontal flips, translations, and rotations. The learning rate was dynamically adjusted downward based on the validation

Algorithm	Test AUROC (%)
1. Wang et al. (2017) [50]	73.8
2. CheXNet (2017) [41]	84.4
3. Google AutoML (2018) [1]	79.7
4. LEAF	84.3

Table 1: Performance on Chest X-rays testing set for hand-designed architectures and for networks that were evolved using Google AutoML and LEAF. LEAF improves significantly over Google AutoML and achieves performance virtually identically to the best hand-designed DNN, demonstrating state-of-the-art results in a task that requires very large networks.

AUROC every epoch and sometimes reset back to its original value if the validation performance plateaued. After training was complete, the testing set images were evaluated 20 times with data augmentation enabled and the network outputs were averaged to form the final prediction result.

Table 1 compares the performance of the best evolved networks with existing approaches that use hand-designed network architectures on a holdout testing set of images. These include results from the authors who originally introduced the Chest X-rays dataset [50] and also CheXNet [41], which is the currently published state-of-the-art in this task. For comparison with other AutoML systems, results from Google AutoML [1] are also listed. Google AutoML was set to optimize a vision task using a preset time of 24 hours (the higher of the two time limits available to the user), with an unknown amount of compute and number of worker machines. Due
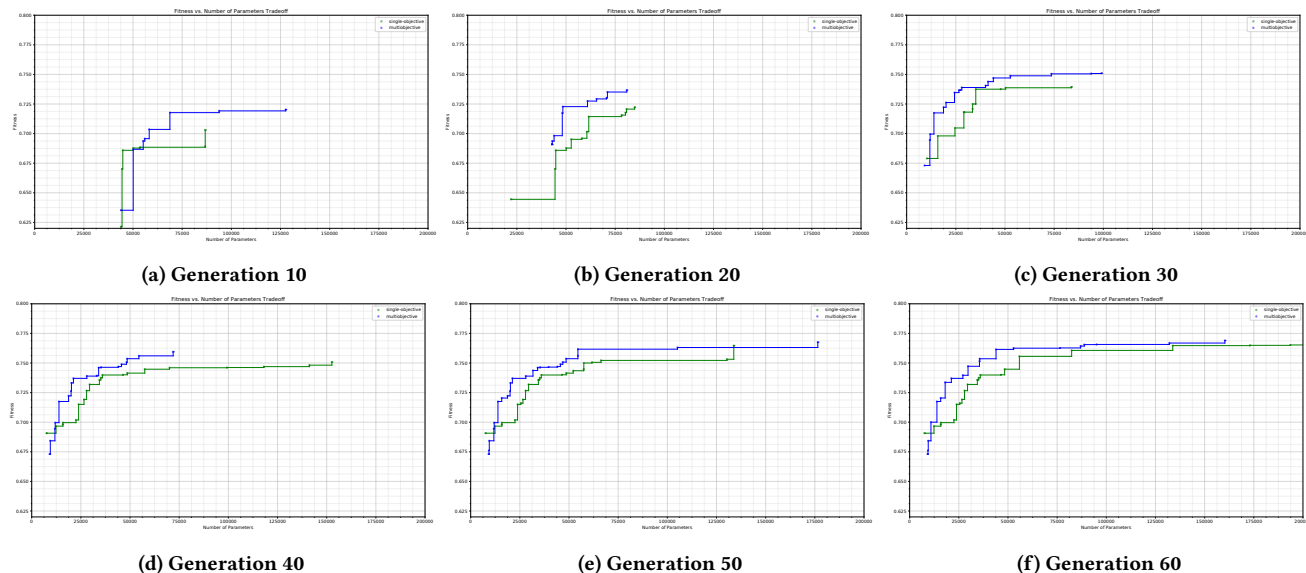
(a) Generation 10

(b) Generation 20

(c) Generation 30

(d) Generation 40

(e) Generation 50

(f) Generation 60

Figure 4: A comparison of Pareto fronts generated by LEAF using single-objective (green) and multiobjective (blue) CoDeep-NEAT at various generations. The x-axis shows number of parameters (secondary objective) and the y-axis shows AUROC fitness (primary objective). The Pareto front for multiobjective LEAF dominates over the single objective Pareto front. In other words, multiobjective LEAF discovered trade-offs between complexity and performance that are always better than those found by standard, single-objective LEAF. Multiobjective LEAF not only found architectures with state-of-the-art performance, but also networks that are smaller and therefore more practical.

to the size of the domain, it was not practical to evaluate Chest X-rays with other AutoML methods. The performance of best network discovered by LEAF matches that of the human designed CheXNet. LEAF is also able to exceed the performance of Google AutoML by a large margin of nearly 4 AUROC points. These results demonstrate that state-of-the-art results are possible to achieve even in domains that require large, sophisticated networks.

An interesting question then is: can LEAF also minimize the size of these networks without sacrificing much in performance? Interestingly, when LEAF used the multiobjective extension of CoDeep-NEAT (multiobjective LEAF) to maximize fitness and minimize network size, LEAF actually converged faster during evolution to the same final fitness. As expected, multiobjective LEAF was also able to discover networks with fewer parameters. As shown in Figure 4, the Pareto front generated during evolution by multiobjective LEAF (blue) dominated that of single-objective LEAF (green) when compared at the same generations during evolution. Although single-objective LEAF used standard CoDeepNEAT, it was possible to generate a Pareto front by giving the primary and secondary objective values of all the networks discovered in past generations to Algorithm 3. The Pareto front for multiobjective LEAF was also created the same way.

Interestingly, multiobjective LEAF discovered good networks in multiple phases. In generation 10, networks found by these two approaches have similar average complexity but those evolved by multiobjective LEAF have much higher average fitness. This situation changes later in evolution and by generation 40, the average complexity of the networks discovered by multiobjective LEAF is noticeably lower than that of single-objective LEAF, but the gap in average fitness between them has also narrowed. Multiobjective LEAF first tried to optimize for the first objective (fitness) and only when fitness was starting to converge, did it try to improve the

second objective (complexity). In other words, multiobjective LEAF favored progress in the metric that was easiest to improve upon at the moment and did not get stuck; it would try to optimize another objective if no progress was made on the current one.

Visualizations of selected networks evolved by multiobjective LEAF are shown in Figure 5. LEAF was able to discover a very powerful network (Figure 5b) that achieved 77% AUROC after only eight epochs of training. This network has 125K parameters and is already much smaller than networks with similar fitness discovered with single-objective LEAF. Furthermore, multiobjective LEAF was able to discover an even smaller network (Figure 5a) with only 56K parameters and a fitness of 74% AUROC after eight epochs of training. The main difference between the smaller and larger network is that the smaller one uses a particularly complex module architecture (Figure 5c) only two times within its blueprint while the larger network uses the same module four times. This result shows how a secondary objective can be used to bias the search towards smaller architectures without sacrificing much of the performance.

5 DISCUSSION

The results for LEAF in the Wikidetox domain show that an evolutionary approach to optimizing deep neural networks is feasible. It is possible to improve over a naive starting point with very little effort and to beat the existing state-of-the-art of both AutoML systems and hand-design with more effort. Although a significant amount of compute is needed to train thousands of neural networks, it is promising that the results can be improved simply by running LEAF longer and by spending more on compute. This feature is very useful in a commercial setting since it gives the user flexibility to find optimal trade-offs between resources spent and the quality of the results. With computational power becoming cheaper and more

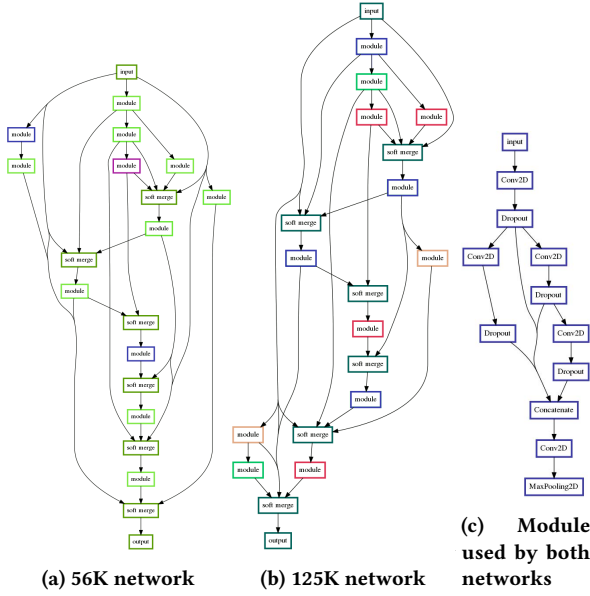(a) 56K network   (b) 125K network   (c) Module used by both networks

Figure 5: Visualizations of networks with different complexity discovered by multiobjective LEAF. The performance of the smaller 56K network (Figure 5a) is nearly as good as that of the larger 125K network (Figure 5b). The smaller network uses only two instances of the module architecture shown in Figure 5c while the larger network uses four instances of the same module. These two networks show that multiobjective LEAF is able to find good trade-offs between two conflicting objectives by cleverly using modules.

available in the future, significantly better results are expected to be obtained. Not all the approaches can put such power to good use, but evolutionary AutoML can.

The experiments with LEAF show that multiobjective optimization is effective in discovering networks that trade-off multiple metrics. As seen in the Pareto fronts of Figure 4, the networks discovered by multiobjective LEAF dominate those evolved by single-objective LEAF with respect to the complexity and fitness metrics in almost every generation. More surprisingly, multiobjective LEAF also maintains a higher average fitness with each generation. This finding suggests that minimizing network complexity produces a regularization effect that also improves the generalization of the network. This effect may be due to the fact that networks evolved by multiobjective LEAF reuse modules more often when compared to single-objective LEAF; extensive module reuse has been shown to improve performance in many hand-designed architectures [22, 48].

In addition to the three goals of evolutionary AutoML demonstrated in this paper, a fourth one is to take advantage of multiple related datasets. As shown in prior work [30], even when there is little data to train a DNN in a particular task, other tasks in a multitask setting can help achieve good performance. Evolutionary AutoML thus forms a framework for utilizing DNNs in domains that otherwise would be impractical due to lack of data.

6 CONCLUSION

This paper showed that LEAF can outperform existing state-of-the-art AutoML systems and the best hand designed solutions. The hyperparameters, components, and topology of the architecture can all be optimized simultaneously to fit the requirements of the

task, resulting in superior performance. LEAF achieves such results even if the user has little domain knowledge and provides a naive starting point, thus democratizing AI. With LEAF, it is also possible to optimize other aspects of the architecture at the same time, such as size, making it more likely that the solutions discovered are useful in practice.

The biggest impact of automated design frameworks such as LEAF is that it makes new and unexpected applications of deep learning possible in vision, speech, language, robotics, and other areas. In the long term, hand-design of algorithms and DNNs may be fully replaced by more sophisticated, general-purpose automated systems to aid scientists in their research or to aid engineers in designing AI-enabled products.

APPENDIX

Evolution parameter	Value
Wikidetox domain	
Module population size	56
Add module node prob	0.05
Add module connection prob	0.05
Module species size	4
Blueprint population size	22
Add blueprint node prob	0.05
Add blueprint connection prob	0.05
Blueprint species size	1
Assembled population size	100
Chest X-rays domain	
Module population size	56
Add module node prob	0.08
Add module connection prob	0.08
Module species size	4
Blueprint population size	22
Add blueprint node prob	0.16
Add blueprint connection prob	0.12
Blueprint species size	1
Assembled population size	100

Table 2: Evolution configuration for the Wikidetox and Chest X-rays domains.

Search space parameter	Range
Wikidetox domain	
Layer types	Conv1D, LSTM, GRU, Dropout
Layer width	[64, 192]
Kernel Size	1, 3, 5, 7
Activation Function	ReLU, Linear, ELU, SeLU
Weight Initializer	Glorot, He
Dropout Rate	[0, 0.5]
Weight Decay	[1e-9, 1e-3]
Minimum pooling layers	5
Weight sharing between layers	False
Chest X-rays domain	
Layer types	Conv2D, Dropout
Layer width	[16, 64]
Kernel Size	1, 3
Activation Function	ReLU, Linear, ELU, SeLU
Weight Initializer	Glorot, He
Dropout Rate	[0, 0.7]
Weight Decay	[1e-9, 1e-3]
Minimum pooling layers	4
Weight sharing between layers	True

Table 3: Search space explored for the Wikidetox and Chest X-rays domains.

ACKNOWLEDGEMENTS

REFERENCES

[1] 2017. AutoML for large scale image classification and object detection. https://ai.googleblog.com/2017/11/automl-for-large-scale-image.html. (Nov 2017).
[2] 2019. Amazon Web Services (AWS) - Cloud Computing Services. aws.amazon.com. (2019).
[3] 2019. Google Cloud. cloud.google.com. (2019).
[4] 2019. Jigsaw Toxic Comment Classification Challenge. kaggle.com/jigsaw-toxic-comment-classification-challenge. (2019).
[5] 2019. Metric Optimization Engine. https://github.com/Yelp/MOE. (2019).
[6] 2019. Microsoft Azure Cloud Computing Platform and Services. azure.microsoft.com. (2019).
[7] 2019. Research:Detox/Data Release. meta.wikimedia.org/wiki/Research:Detox. (2019).
[8] 2019. SigOpt. sigopt.com/product. (2019).
[9] 2019. StudioML. https://www.studio.ml/. (2019).
[10] 2019. Using the Microsoft TLC Machine Learning Tool. jamesmccaffrey.wordpress.com/2017/01/13/using-the-microsoft-tlc-machine-learning-tool. (2019).
[11] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.
[12] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *arXiv preprint arXiv:1607.04606* (2016).
[13] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. 2016. Recurrent Neural Networks for Multivariate Time Series with Missing Values. *CoRR* abs/1606.01865 (2016). http://arxiv.org/abs/1606.01865
[14] F. Chollet et al. 2015. Keras. (2015).
[15] Theodora Chu, Kylie Jue, and Max Wang. 2016. Comment Abuse Classification with Deep Learning. *Von https://web. stanford. edu/class/cs224n/reports/2762092. pdf abgerufen* (2016).
[16] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*. ACM, 160–167.
[17] Kalyanmoy Deb. 2015. Multi-objective evolutionary algorithms. In *Springer Handbook of Computational Intelligence*. Springer, 995–1015.
[18] Thomas Elsken, J Hendrik Metzen, and Frank Hutter. 1804. Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution. *ArXiv e-prints* (1804).
[19] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. 2008. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research* 9, May (2008), 937–965.
[20] Faustino J Gomez and Risto Miikkulainen. 1999. Solving non-Markovian control tasks with neuroevolution. In *IJCAI*, Vol. 99. 1356–1361.
[21] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 6645–6649.
[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. *CoRR* abs/1603.05027 (2016). http://arxiv.org/abs/1603.05027
[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *European conference on computer vision*. Springer, 630–645.
[25] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[26] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks.. In *CVPR*, Vol. 1. 3.
[27] Eamonn Keogh and Abdullah Mueen. 2011. Curse of dimensionality. In *Encyclopedia of machine learning*. Springer, 257–258.
[28] D. P. Kingma and J. Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).
[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[30] Jason Liang, Elliot Meyerson, and Risto Miikkulainen. 2018. Evolutionary Architecture Search for Deep Multitask Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18)*. ACM, New York, NY, USA,

466–473. https://doi.org/10.1145/3205455.3205489
[31] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).
[32] Ilya Loshchilov and Frank Hutter. 2016. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. *arXiv preprint arXiv:1604.07269* (2016).
[33] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. 2018. NSGA-NET: A Multi-Objective Genetic Algorithm for Neural Architecture Search. *arXiv preprint arXiv:1810.03522* (2018).
[34] E. Meyerson and R. Miikkulainen. 2018. Beyond Shared Hierarchies: Deep Multitask Learning through Soft Layer Ordering. *ICLR* (2018).
[35] R. Miikkulainen, J. Liang, E. Meyerson, et al. 2017. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548* (2017).
[36] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.
[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
[38] David E Moriarty and Risto Miikkulainen. 1998. Hierarchical evolution of neural networks. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*. IEEE, 428–433.
[39] Joe Yue-Hei Ng, Matthew J. Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. 2015. Beyond Short Snippets: Deep Networks for Video Classification. *CoRR* abs/1503.08909 (2015). http://arxiv.org/abs/1503.08909
[40] Mitchell A Potter and Kenneth A De Jong. 1994. A cooperative coevolutionary approach to function optimization. In *International Conference on Parallel Problem Solving from Nature*. Springer, 249–257.
[41] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. 2017. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225* (2017).
[42] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2018. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548* (2018).
[43] E. Real, S. Moore, A. Selle, et al. 2017. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041* (2017).
[44] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.
[45] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Mr Prabhat, and Ryan P Adams. 2015. Scalable Bayesian Optimization Using Deep Neural Networks.. In *ICML*. 2171–2180.
[46] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10 (2002), 99–127. stanley:ec02
[47] M. Suganuma, S. Shirakawa, and T. Nagao. 2017. A genetic programming approach to designing convolutional neural network architectures. In *Proc. of GECCO*. ACM, 497–504.
[48] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.
[49] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. 2010. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research* 11, Dec (2010), 3371–3408.
[50] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. 2017. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. IEEE, 3462–3471.
[51] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.
[52] Kohsuke Yanai and Hitoshi Iba. 2001. Multi-agent robot learning by means of genetic programming: Solving an escape problem. In *International Conference on Evolvable Systems*. Springer, 192–203.
[53] Chern Han Yong and Risto Miikkulainen. 2001. Cooperative coevolution of multi-agent systems. *University of Texas at Austin, Austin, TX* (2001).
[54] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. 2011. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation* 1, 1 (2011), 32–49.
[55] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016). http://arxiv.org/abs/1611.01578