DensEMANN: How to Automatically Generate an Efficient while Compact DenseNet

Anonymous Author(s)
Affiliation
Address
email

Abstract

We present a new and improved version of DensEMANN, an algorithm that grows small DenseNet architectures virtually from scratch while simultaneously training them on target data. Following a finite-state machine based on the network's accuracy and the evolution of its weight values, the algorithm adds and prunes dense layers and convolution filters during training only when this leads to significant accuracy improvement. We show that our improved version of DensEMANN can quickly and efficiently search for small and competitive DenseNet architectures for well-known image classification benchmarks. In half a GPU day or less, this method generates networks with under 500k parameters and between 93% and 95% accuracy on various benchmarks (CIFAR-10, Fashion-MNIST, SVHN). For CIFAR-10, we show that it comes very close to the state-of-the-art Pareto front between accuracy and size, finding networks with 98.84% of the accuracy and 98.08% of the size of the closest Pareto-optimal competitor, in only 0.70% of the search time it took to find that competitor. We also show that DensEMANN generates its networks with optimal weight values, and identify a simple mechanism that allows it to generate such optimal weights. All in all, we show this "in-supervised" essentially incremental approach to be promising for a fast design of competitive while compact convolution networks.

1 Introduction

The architecture of a neural network (NN) is known to have a great impact on its performance on a target task—on par with that of the training process through which the NN learns the task [1, 2, 3, 4, 5]. The main motivation behind neural architecture search (NAS) is precisely to find the most adequate architecture for a task, in the sense of achieving the highest accuracy [6, 7, 8], but also of using resources as efficiently as possible [7, 9, 8]. To this aim, NAS algorithms must compare the performance a great number of candidate architecture designs. Since naïvely training all of them from scratch would be very inefficient, the NAS community has put much effort into developping reliable and ressource-efficient performance estimation strategies [6, 7, 2, 3, 8].

In this respect, so-called "growing", "constructive" or "incremental" algorithms provide an interesting approach to NAS and performance estimation. They simultaneously build and train candidate architectures, by adding elements such as weights, neurons, layers etc. during the training process [10, 11, 12, 13]. Since new candidate networks are evaluated on basis of the weights learned by previous candidates [14, 15], the time and computation resources consumed by the entire NAS process are equivalent to those required for training a single NN [12, 14]. Furthermore, the search space is not bounded, as new elements may be added *ad infinitum* [13, 16].

This paper presents our research on DensEMANN [1], an algorithm that simultaneously grows and trains small and efficient DenseNets [17] virtually from scratch. Encouraged by previous positive

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

results found by its authors [1, 18], and by the great success of other similar methods [13, 19], we created a new version of this algorithm with the aim of approaching state-of-the-art performance for well-known benchmarks, or at least the state-of-the-art Pareto front between performance and model size. As a secondary goal, we tested the authors' claim that DensEMANN-generated networks perform equally or better than similar NN even when these are trained from scratch [1].

Section 2 provides some background on growing-based NAS and related research. Section 3 contains a presentation of DensEMANN's inner workings 3.1 and lists our modifications with regards to [1] 3.2. Section 4 presents our experiments and their results, and Section 5 includes our conclusions and suggestions for future research.

2 Rediscovery of an incremental approach

Most likely [18, 13, 20], the first growing-based NAS algorithm was Dynamic Node Creation (DNC) [21] or the cascade-correlation algorithm (CC-Alg) [22]. During the 1990's, the research field of "constructive" algorithms (as they were called) was so active that at least two contemporary surveys exist of this field [23, 24]. To our knowledge, research on growing-based NAS for convolution neural networks (CNN) only took off after the introduction of `Net2Net` operators [25] and network morphisms [26], which can instantly make a CNN wider or deeper while not changing its behaviour.

We have observed a recurring pattern of *rediscovery*, or *convergence*, between early growing techniques and more recent ones. Parallels can be made, for instance, between some network morphisms [25, 26] and early node-splitting techniques [27], or between the "backwards steps" in some modern algorithms [19, 28] and early growing-pruning hybridations [24, 23]. We believe that this convergence is helped by a direct correspondence (described in [25, 29]) between pioneering neural architectures such as multi-layer perceptrons, and more recent ones like CNN: perceptron layers correspond to convolutional layers, and perceptron neurons correspond to 3D convolution filters.

The most serious competitor to growing-based NAS algorithms are trainless or zero-cost algorithms [30, 31, 2]. These evaluate candidate NN on basis of their performance with random weights. Such methods can explore large search spaces in a matter of minutes or even seconds [31, 2]. However, extra time is still needed for training the final candidate architecture in order to use it.

3 DensEMANN: building a DenseNet one filter at a time

DensEMANN [1] is a growing algorithm that simultaneously builds and trains DenseNets [17] virtually from scratch. It is based on EMANN [16], an algorithm that grows multi-layer perceptrons with an analogous connection scheme to that of DenseNet, and on previous research on DenseNet-growing techniques by the same authors [18]. Based on an introspective "self-structuring" or "in-supervised" approach, much more in line with real neurology than purely performance-based NAS, it grows and prunes the candidate NN on basis of the evolution of its internal weight values.

3.1 General presentation of DensEMANN

By default, DensEMANN's seed architecture is a DenseNet containing a single dense block, inside which there is a single dense layer producing $k = 12$ feature maps. "Dense layers" may either be DenseNet or DenseNet-BC composite functions, with the same characteristics as in [17]. Also like in [17], the DenseNet's inputs are pre-processed by an initial convolution layer with $2 * k = 24$ filters, and its final outputs are generated through 2D batch normalization [32] and a fully connected layer.

Paralleling EMANN's "double level adaptation" [16], DensEMANN consists of two components, the macro-algorithm and the micro-algorithm [1]. Each of them builds the seed network's dense block at a different granularity level. Afterwards, the dense block may be replicated a certain user-set number of times to produce an N-block DenseNet (see Section 3.1.3).

3.1.1 The macro-algorithm

The **macro-algorithm** works at the level of dense layers, and is reminiscent of CC-Alg [22]. It iteratively stacks up dense layers in the block until there is no significant change in the accuracy (see Algorithm 1 for its pseudocode).

Algorithm 1 DensEMANN macro-algorithm

procedure MACROALGORITHM($model$)
  $accuracy_{last} \leftarrow 0$
  $model_{last} \leftarrow model$
  $model, accuracy \leftarrow$ MICROALGORITHM($model$)
  while $|accuracy - accuracy_{last}| \geq IT$ **do**
    $accuracy_{last} \leftarrow accuracy$
    $model_{last} \leftarrow model$
    $model \leftarrow$ ADDNEWLAYER($model$)
    $model, accuracy \leftarrow$ MICROALGORITHM($model$)
  end while
  return $model_{last}$
end procedure

Each new layer is created with the same initial number of 3D convolution filters, set by a *growth rate* parameter (by default $k = 12$). In the case of DenseNet-BC, the dense layer's first convolution is created with $4 * k$ filters, and its second convolution with k filters.

Before each layer addition, the macro-algorithm saves the current NN model (architecture and weights) and its accuracy. It then adds the new layer and calls the micro-algorithm to build it. Once the micro-algorithm finishes, the macro-algorithm compares the current accuracy to the one before the new layer was added. If the absolute difference between the two accuracies surpasses an *improvement threshold* (by default $IT = 0.01$), the macro-algorithm loops and creates a new layer. Otherwise, the algorithm undoes the layer's addition by loading back the last saved model, and stops there.

3.1.2 The micro-algorithm

The **micro-algorithm** works at the level of convolution filters. It operates only in the dense block's last layer (for DenseNet-BC, the second convolution in the last dense layer), and follows a finite-state machine with states for growing, pruning, and performance recovery.

While the network is trained through standard backpropagation, the micro-algorithm establishes different categories of filters on basis of their *kernel connection strength* (kCS). For filter λ, its kCS is the arithmetic mean of its absolute weight values $w_1, ..., w_n$.

$$kCS_\lambda = \sum_{i=1}^{n} |w_i| / n \tag{1}$$

A filter is declared "settled" if its kCS remains near-constant for the last 40 training epochs.[1] After at least $k/2$ filters have settled, settled filters can be declared "useful" if their average kCS over the last 10 epochs falls above a *usefulness threshold* (UFT), and "useless" if that same value falls below a *uselessness threshold* (ULT). During the micro-algorithm's improvement stage (1), the UFT and ULT are recalculated after each training epoch on basis of the maximum and minimum kCS among settled filters, and of user-settable parameters UFT_{auto} and ULT_{auto} (by default respectively 0.8 and 0.2):

$$UFT = UFT_{auto} * \left(\max_{\lambda \, is \, settled}(kCS_\lambda) - \min_{\lambda \, is \, settled}(kCS_\lambda) \right) + \min_{\lambda \, is \, settled}(kCS_\lambda) \tag{2}$$

$$ULT = ULT_{auto} * \left(\max_{\lambda \, is \, settled}(kCS_\lambda) - \min_{\lambda \, is \, settled}(kCS_\lambda) \right) + \min_{\lambda \, is \, settled}(kCS_\lambda) \tag{3}$$

The micro-algorithm uses these three filter categories—settled, useful and useless—as references for building the network's last layer. To do this, it alternates between three stages (see Figure 1):

1. **Improvement**: the network starts training with initial learning rate $LR_0 = 0.1$, and filters are progressively declared settled, then useful or useless. Meanwhile, a countdown begins with a fixed length in training epochs: the *patience parameter* (by default $PP = 40$ epochs).

[1] The actual criterion is: if the first derivative of the kCS, calculated as the difference between the current kCS and the one 10 epochs ago divided by 10-1=9, remains near-zero for the last 30 epochs.

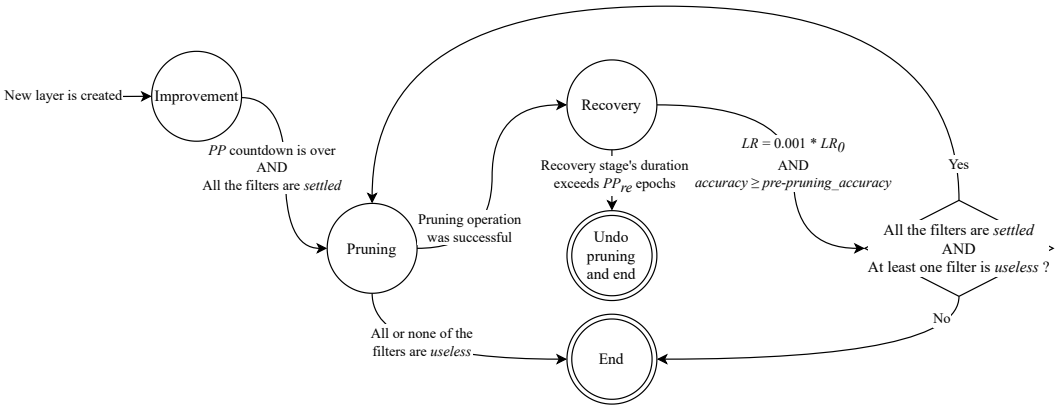

Figure 1: Flowchart of the finite-state machine for DensEMANN's micro-algorithm.

The learning rate (LR) is divided by 10 after 50% and 75% of this countdown has elapsed. If at any point during this stage the number of useful filters exceeds its last maximum value, a new filter is added and the countdown and LR are reset. The stage ends when (1) the countdown is over and (2) all filters have settled. The next stage is always 2 (pruning).

2. **Pruning**: if all or none of the layer's filters are useless, the micro-algorithm ends here. Otherwise, the micro-algorithm saves the NN model and its accuracy (like the macro-algorithm does before creating a new layer), deletes all useless filters, and moves to stage 3 (recovery). From the first pruning stage onwards, the UFT and ULT values are frozen.
3. **Recovery**: the network is trained again, with the same PP-epoch countdown and the same initial and scheduled LR values as in stage 1 (improvement), but without filter additions. There are two additional countdowns, one with the same length as the last improvement stage (PP + the number of epochs it took for all filters to settle), and another one with a length of $PP_{re} > PP$ epochs (by default $PP_{re} = 130$). Three things may happen:
   (a) If (1) the learning rate has already reached its lowest scheduled value (i.e. in practice after $0.75 * PP$ epochs) and (2) the current accuracy has reached or surpassed its pre-pruning value, the stage ends. If at this point (3) all the filters have settled and (4) there is at least one useless filter, the next stage is 2 (pruning). Otherwise, the micro-algorithm ends.
   (b) If the stage's duration exceeds PP_{re} epochs, the previous pruning operation is considered "fruitless" and undone. The pre-pruning model is loaded back, and the micro-algorithm ends.
   (c) If the stage's duration exceeds that of the previous improvement stage, the filters' kCS values are considered "frozen". In practice, this means that all filters are declared settled at most 40 epochs after this point, and that the frozen kCS values will be used as the reference for any subsequent pruning.

DensEMANN's weight initialization mechanisms are also worth commenting:

1. The weights for new layers are initialized along a truncated normal distribution, similar to that of TensorFlow v1's [33] "variance scaling initializer". For this initializer, the distribution's standard deviation (SD) is usually inversely proportional to each layer's number of input features and to its filters' dimensions. However, for layers that the micro-algorithm can act upon, the distribution's SD only depends on the filters' dimensions, resulting in bigger initial weights in these layers.
2. The weights of new filters are initialized using a special *complementarity* mechanism borrowed from EMANN [16]. The weights' absolute values are random, and follow the same truncated normal distribution as weights in new modifiable layers. Their sign configuration, however, is not random: it is the inverted sign configuration of the filter with the lowest kCS. This is done to ensure the mutual complementarity and co-adaptation of new and old filters.

3.1.3 Building more than one dense block

DensEMANN can also be set to build DenseNets with a user-set number of dense blocks N. To do this, DensEMANN first uses the macro- and micro-algorithms to build a one-block DenseNet, and then replicates the generated dense block $N-1$ times to create a N-block DenseNet.

Between blocks, transition layers are created with a similar architecture to that in [17], i.e. with a batch normalisation [32], a ReLU function [34], a 1x1 convolution and a 2x2 average pooling layer [35]. The number of filters in the 1x1 convolution depends on whether the network is a DenseNet or a DenseNet-BC: for DenseNet it is the same as the previous block's number of output features, while for DenseNet-BC it is multiplied by a reduction factor, by default $\theta = 0.5$.

The weights for the $N-1$ new blocks are initialized using the same method that DensEMANN uses for new layers.[2] After the new blocks are added, the NN is trained for 300 extra epochs. The LR recovers its initial value LR_0 at the beginning of these last 300 epochs, and is divided by 10 on epochs 150 and 255 (i.e., 50% and 75% through the extra training epochs). During these epochs, DensEMANN adopts a "best model saving" approach: the NN's weights are saved whenever its loss reaches a new minimum value, and after the 300 epochs, these "best" weights are loaded back to allow the NN to reach optimal performance.

Although it is in direct contrast with DensEMANN's incremental philosophy, this method for replicating the generated block $N-1$ times is activated by default, with $N = 3$. Its development was motivated by previous experimental results with mechanisms that copy DensEMANN-generated layers a predefined number of times, and by the good performance of cell-based NAS approaches [13, 14, 36] that first search for a small neural pattern (the cell) and then replicate it N times. In Appendix A, we give the results of an ablation study that compares, among others, DensEMANN's performance with and without this dense block replication mechanism.

3.2 Differences with the original DensEMANN

Below are the differences between our version of the algorithm and the one described in [1]:

1. Changes to the macro-algorithm:
   (a) The last layer addition is always undone, as it does not fulfil the accuracy improvement criterion. This was suggested in [1], and EMANN uses a similar mechanism [16].
   (b) The improvement threshold's default value was changed to $IT = 0.01$. Observations in [1] suggest that, with the previous default value (0.005), the last few layer additions do not have a big impact in the NN's final accuracy.
2. Changes to the micro-algorithm:
   (a) Only settled filters may be declared useful or useless. This was proposed in [1] as a means to avoid quick cascades of often superfluous filter additions.
   (b) The patience parameter's default value was changed to $PP = 40$ epochs. Observations in [1] suggest that it takes approximately 40 epochs for all the filters in a layer to settle.
   (c) If *at any point* during the improvement stage the number of useful filters exceeds its last highest value, a new filter is added and the patience countdown is reset. In v1.0, this can only happen if the countdown has not yet ended.
   (d) The pruning and recovery stages have been heavily modified to avoid long recovery stages and their effects. We indeed observed that the kCS of settled filters is not constant but actually decreases very slowly over time. If the recovery stage is too long, this causes a very harsh pruning after which the accuracy cannot be recovered.
3. We added a method that replicates the generated dense block N times.

4 Experiments

We re-implemented DensEMANN from scratch using the PyTorch (v1.8 or higher[3]) [37] and Fastai (v2.5.3) [38] Python libraries. Our code is based on the DenseNet implementation for PyTorch by

[2] Except for transition layers and the first layers in each block, whose weights are initialized with zero values.

[3] Depending on the computational environment, we used PyTorch v1.8.1 or v1.10.0+cu113 for running our experiments. See further below.

Pleiss et al. [39] and on the original DensEMANN implementation by García-Díaz [40] (both under MIT license). We initially replicated the latter as faithfully as possible, including the unusual weight initialization described in Section 3.1.2. Then, we made the modifications described in this paper.

In our experiments we use three well-known image classification benchmarks: CIFAR-10 [41], Fashion-MNIST [42] and SVHN [43]. For each training process (either when running DensEMANN or training NN from scratch), we split the target data into three parts:

- Training set: a random set of examples, different for each training process. It is used for training the NN. For CIFAR-10 and Fashion-MNIST it contains 45,000 "training" images, and for SVHN it contains 6,000 images from the "train" and "extra" sets.
- Validation set: another random set of examples, different for each training process but separate from the training set (there is no overlap between the two sets). It is used for estimating the NN's accuracy and loss during training (and in the case of DensEMANN, during growing). For CIFAR-10 and Fashion-MNIST it contains 5,000 "training" images, and for SVHN it contains 6,000 images from the "train" and "extra" sets.
- Test set: a predefined set of examples that the NN never "sees" during training. It is used for evaluating the NN's final performance (accuracy and cross-entropy loss), and to compare it against the state of the art. It is the entire set of "test" images provided by each dataset's authors: 10,000 images for CIFAR-10 and Fashion-MNIST, and 26,032 images for SVHN.

A batch size of 64 images is used for all datasets, and for all three of the above splits.

Our data pre-processing workflow is as follows:

1. Random crop with 4-pixel padding + random horizontal flip (as in [17]), only for CIFAR-10's training and validation data.
2. Normalization. For CIFAR-10 we use the dataset's channel-wise mean and SD values as in [17]. For the other two datasets we assume mean and SD values of 0.5 for all channels.
3. Cutout regularization [44], only for the training and validation data.

DensEMANN's parameters were set to their default values: $IT = 0.01$, $PP = 40$, $PP_{re} = 130$, $LR_0 = 0.1$, $k = 12$ for the first layer, $N = 3$ blocks in the final network. All other default values are the same as in [1]. We opted to generate DenseNet-BC architectures, as in past research they provided better results than standard DenseNet [1, 18, 17].

For our experiments, we used the following computation environments:

- MSi GT76 Titan DT laptop: Windows 10 Pro (64-bit) OS, Intel Core i9-10900K CPU (3.70 GHz), NVIDIA GeForce RTX 2080 Super GPU, 64.0 GB RAM (63.9 GB usable). Python is v3.9.8, PyTorch is v1.10.0+cu113.
- Internal cluster: Linux Ubuntu 20.04.4 LTS (x86-64) OS, 16 AMD EPYC-Rome Processor CPUs (2.35 GHz), NVIDIA GeForce RTX 3090 GPU, 64 GB RAM. Python is v3.8.6, PyTorch is v1.8.1.

In Table 1, GPU times in black were obtained with the MSi GT76, while GPU times in *italized purple* were obtained on the internal cluster. We consider the times obtained on the MSi GT76 to be more reliable, as on the internal cluster we have let up to four tests run at a time, whereas on the MSi GT76 we have only run one test at a time.

The total computation time for all the experiments in this paper (excluding the appendices) was 6.49 GPU days (3.61 days on the MSi GT76 and 2.88 days on the internal cluster). Below are the computation times for each experiment:

- 4.1: 4.88 GPU days (2.81 on the MSi GT76, 2.07 on the cluster).
- 4.2: 1.61 GPU days (0.80 on the MSi GT76, 0.80 on the cluster).

4.1 DensEMANN's full potential unlocked

We began by running DensEMANN 5 times for each dataset, in order to get an idea of the algorithm's performance. The results of this experiment correspond to the two first lines for each dataset in

Table 1: Using DensEMANN for growing and training DenseNet-BC on benchmark datasets

Dataset	Experiment	GPU execution time (hours)	GPU inference time (seconds)	Num. layers per block	Trainable parameters (k)	Validation set		Test set	
						Acc. (%)	Loss	Acc. (%)	Loss
CIFAR-10	Average performance	13.48 ± 2.72	3.21 ± 0.36	5.8 ± 1.6	186.36 ± 56.68	90.06 ± 1.38	0.30 ± 0.04	93.41 ± 0.90	0.23 ± 0.03
	Best network	16.55 (67.39)	3.47	7	245.42	91.34	0.26	93.91	0.21
	Best network retrained	3.86 ± 0.01	3.46 ± 0.04	7	245.42	91.90 ± 0.42	0.25 ± 0.01	94.25 ± 0.16	0.20 ± 0.01
Fashion-MNIST	Average performance	*6.55 ± 1.80*	*3.98 ± 0.35*	2.2 ± 1.3	51.84 ± 25.51	92.63 ± 0.73	0.20 ± 0.02	93.68 ± 0.68	0.20 ± 0.01
	Best network	*7.53 (32.75)*	*4.26*	3	68.64	93.62	0.18	94.43	0.19
	Best network retrained	*2.81 ± 0.02*	*3.75 ± 0.20*	3	68.64	93.70 ± 0.53	0.18 ± 0.01	94.47 ± 0.22	0.19 ± 0.01
SVHN	Average performance	*3.39 ± 0.26*	*13.36 ± 0.33*	11.0 ± 1.2	339.81 ± 63.39	93.38 ± 0.47	0.24 ± 0.02	94.43 ± 0.29	0.27 ± 0.02
	Best network	*3.23 (16.96)*	*13.28*	11	336.07	94.10	0.22	94.70	0.26
	Best network retrained	*1.04 ± 0.17*	*11.76 ± 2.69*	11	336.07	93.81 ± 0.39	0.23 ± 0.01	94.50 ± 0.16	0.26 ± 0.01

Table 1: the "average performance" lines contains the mean and SD over the 5 runs for each variable, whereas the "best network" line corresponds to the DensEMANN-generated NN that obtained the lowest (cross-entropy) loss on the validation set. In the latter case, we indicate two execution times: the execution time for the run that generated this "best" NN, and the total execution time for all 5 runs of DensEMANN (i.e. the total GPU time that we consumed to search for this optimal candidate).

All in all, DensEMANN performs very well on all three benchmarks. The generated architectures are always under 0.5 million parameters (in the case of Fashion-MNIST they are even under 0.1 million parameters), yet the average test set accuracies are all between 93% and 95%. The current state of the art test set accuracies on CIFAR-10 [45] and SVHN [46] are at 99% or higher, while that on Fashion-MNIST [47] is at just under 97%. This said, the top-performing models for these benchmarks are very large, containing several millions of parameters.

Concerning DensEMANN's execution times, they range from around 3 hours (SVHN) to just over half a day (CIFAR-10). Consequently, it always took us less than 3 days to run DensEMANN 5 times, and find our best candidate network for all benchmarks.

It is noteworthy that DensEMANN *does* seem to build minimal architectures that adapt to each dataset's peculiarities. For Fashion-MNIST, a grayscale dataset with smaller images than the other two datasets, DensEMANN generated very small and shallow architectures—an order of magnitude smaller than those for the two other datasets. Meanwhile, the biggest and deepest NN were generated for SVHN, an RGB dataset whose images contain distractors around the main data to classify.

4.2 Retraining our best networks from scratch: DensEMANN vs. "perfect" NAS

For our second experiment, we erased the weights in DensEMANN's best network for each dataset, replaced them with randomly initialized weights (using an exact copy of TensorFlow's "variance scaling initializer") and trained the network from scratch 5 times for 300 epochs. We used the same workflow as for the block replication mechanism at the end of DensEMANN: beginning with a LR value of $LR_0 = 0.1$, we divide it by 10 on epochs 150 and 255. We also use the same "best model saving" approach, where we save the weight values that produce the lowest validation set loss, and load them back at the end of the training process. The results of this experiment correspond to the "best network retrained" lines in Table 1.

This experiment is useful for two reasons. On one hand, it allows us to test the claims in [1] that DensEMANN generates its networks with optimal weights. If that were the case, the network's performance (accuracy and loss) with the original and retrained weights would not be very different. On the other hand, it can be used for comparing DensEMANN's execution times to those of a hypothetical "perfect" NAS algorithm. As explained in Section 2, even if we imagine a "perfect" zero-cost NAS algorithm that can instantly find an optimal network in DensEMANN's search space, one still needs to train the candidate network before using it. For this reason, we consider the GPU time cost of retraining DensEMANN's final architecture to be a good proxy for the GPU time cost of using such a "perfect" NAS algorithm to explore DensEMANN's search space.

Concerning the network's performance, the difference before and after being retraining is not very big, and this is true on both the validation and test set. In the case of CIFAR-10, there is an improvement for the accuracy, but the loss does not seem significantly different. One-sample T-tests confirm this observation: the only statistically significant differences are found for CIFAR-10's validation and test set accuracies (respectively $P = 0.039$ and $P = 0.010$), and for SVHN's test set accuracy ($P = 0.044$). In the former case the accuracy improved after retraining, while in the latter case it

worsened after retraining. Nevertheless, in both cases the test loss does not change significantly, which leads us to conclude that DensEMANN *does* optimally train the architectures that it grows.

Concerning the GPU times, for all datasets there is a significant difference between DensEMANN's execution times (both the average time and that of the best run) and the average time cost for retraining the best run's final candidate NN. The mean GPU time cost of DensEMANN is 3.49 times longer than that of retraining for CIFAR-10, 2.33 times longer for Fashion-MNIST, and 3.25 times longer for SVHN. Part of this difference most certainly comes from the 300 extra training epochs required by DensEMANN's block replication mechanism—exactly the same number of epochs that we use to retrain the best generated NN from scratch. Since for two similar architectures 300 training epochs will represent a similar GPU time, it is mathematically impossible for DensEMANN to outperform "perfect" NAS' time cost when using block replication. Nevertheless, the GPU time costs of DensEMANN and SGD training remain of the same order of magnitude (hours).

4.3 Comparison against the state of the art

In Table 2 and Figure 2, we take advantage of the widespread use of CIFAR-10 as a benchmark task to map the state of the art for the compromise between model size and error rate on this dataset, and see where DensEMANN fits with regards to that state of the art. Concretely, we use Figure 2 to visualize the current Pareto front for the size vs. error rate compromise. We compare DensEMANN to this Pareto front by identifying the NN models and NAS algorithms that make up this front, and by focusing on the ones that are closest to DensEMANN's performance.

The closest Pareto-optimal competitor to DensEMANN is LEMONADE S-I [19], another growing-based algorithm that is designed specifically to explore the Pareto front between model acccuracy and size. After 80 GPU days LEMONADE S-I generated a final candidate architecture with 190 thousand parameters that obtained 94.5% accuracy on CIFAR-10. DensEMANN's average performance is very close to this: with 98.08% of its size, we reach 98.84% of the LEMONADE S-I network's accuracy, in 0.70% of the GPU time that LEMONADE took to find that network.

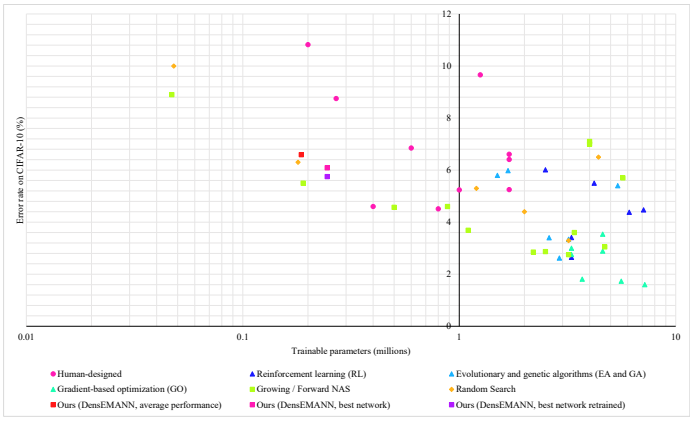

Figure 2: Scatter plot of the performance of the human-designed NN models and NAS algorithms in Table 2 (including DensEMANN): accuracy on CIFAR-10 vs. size in trainable parameters.

Concerning the execution time, a closer competitor to DensEMANN is NASH Random, by the same authors as LEMONADE [14]. On average, it takes 0.19 GPU days to produce its final candidate networks, their average size is 4.4 million parameters, and their average accuracy on CIFAR-10 is 93.50%. DensEMANN reaches a similar accuracy with 4.24% of the size, but its average execution time is 2.96 times longer. It is likely that network morphisms [25, 26], which NASH uses for preserving the NN's behaviour after growing, are in part responsible for its quick execution time.

5 Conclusions and future work

We present DensEMANN, an "in-supervised" growing-based NAS algorithm that simultaneously builds and trains DenseNet architectures for target tasks. We show that, in half a GPU day or less, DensEMANN can generate very small networks (under 500 thousand trainable parameters) for various benchmark image classification tasks, training them with optimal weights that allow them to reach around 94% accuracy on these benchmarks. For one of them (CIFAR-10), we show that

Table 2: Performance comparison of DensEMANN against human-designed NN models and state-of-the-art NAS algorithms, for architectures with less than 10 million parameters

Category	Name	Trainable parameters (M)	Error rate on CIFAR-10 (%)	GPU execution time (days)
Human-designed	ResNet 20 [48]	0.27	8.75	N/A
	ResNet 110 (as reported by He et al. [48])	1.7	6.61 ± 0.16	N/A
	ResNet 110 (as reported by Huang et al. [49])	1.7	6.41	N/A
	ResNet 110 with Stochastic Depth [49]	1.7	5.25	N/A
	WRN 40-1 (no data augmentation) [50]	0.6	6.85	N/A
	DenseNet 40 (k = 12) [17]	1	5.24	N/A
	DenseNet-BC 100 (k = 12) [17]	0.8	4.51	N/A
	Highway 1 (Fitnet 1) [51]	0.2	10.82	N/A
	Highway 4 [51]	1.25	9.66	N/A
	Petridish initial model (N=6, F=32) + cutout [13]	0.4	4.6	N/A
Reinforcement learning (RL)	NAS-RL / REINFORCE (v1 no stride or pooling) [52]	4.2	5.5	22400
	NAS-RL / REINFORCE (v2 predicting strides) [52]	2.5	6.01	22400
	NAS-RL / REINFORCE (v3 max pooling) [52]	7.1	4.47	22400
	NASNet-A (6 @ 768) [36]	3.3	3.41	2000
	NASNet-A (6 @ 768) + cutout [36]	3.3	2.65	2000
	Block-QNN-S, N=2 [53]	6.1	4.38	96
Evolutionary and genetic algorithms (EA and GA)	Large-Scale Evolution [54]	5.4	5.4	2600
	CGP-CNN (ConvNet) [55]	1.5	5.8	12
	CGP-CNN (ResNet) [55]	1.68	5.98	14.9
	AmoebaNet-A (N=6, F=32) [56]	2.6	3.4 ± 0.08	3150
	AmoebaNet-A (N=6, F=36) [56]	3.2	3.34 ± 0.06	3150
	EcoNAS + cutout [57]	2.9	2.62 ± 0.02	8
Gradient-based optimization (GO)	ENAS + micro search space [58]	4.6	3.54	0.45
	ENAS + micro search space + cutout [58]	4.6	2.89	0.45
	DARTS (1st order) + cutout [59]	3.3	3 ± 0.14	1.5
	DARTS (2nd order) + cutout [59]	3.3	2.76 ± 0.09	4
	XNAS-Small + cutout [60]	3.7	1.81	0.3
	XNAS-Medium + cutout [60]	5.6	1.73	0.3
	XNAS-Large + cutout [60]	7.2	1.6	0.3
Growing / Forward NAS	NASH ($n_steps = 5$, $n_neigh = 8$, 10 runs) [14]	5.7	5.7 ± 0.35	0.5
	LEMONADE SS-I + mixup + cutout [19]	0.047–3.4	8.9–3.6	80
	LEMONADE SS-II + mixup + cutout [19]	0.5–13.1	4.57–2.58	80
	Petridish macro (N=6, F=32) + cutout [13]	2.2	2.85 ± 0.12	5
	Petridish cell (N=6, F=32) + cutout [13]	2.5	2.87 ± 0.13	5
	Petridish cell, more filters (N=6, F=37) + cutout [13]	3.2	2.75 ± 0.21	5
	Firefly, WRN 28-1 seed + BN [10]	4	7.1 ± 0.1	N/A
	GradMax, WRN 28-1 seed + BN [10]	4	7.0 ± 0.1	N/A
Random Search	DARTS Random + cutout [59]	3.2	3.29 ± 0.15	4
	NASH Random ($n_steps = 5$, $n_neigh = 1$) [14]	4.4	6.5 ± 0.76	0.19
	LEMONADE SS-I Random + mixup + cutout [19]	0.048–2	10–4.4	80
Ours	DensEMANN (average performance) + cutout	0.056 ± 0.009	13.91 ± 1.28	0.33 ± 0.05
	DensEMANN (best network) + cutout	0.245	6.09	2.81
	DensEMANN (best network retrained) + cutout	0.245	5.75 ± 0.16	2.97 ± 0.00

DensEMANN's performance is very close to state-of-the-art Pareto-optimality for the compromise between accuracy and neural architecture size. This said, by studying DensEMANN in detail and comparing it to other algorithms in the literature, we find methodologies—such as network morphisms and best model saving—that could make this approach even quicker and more optimal.

Future research on improving DensEMANN could follow some of the following research lines:

- Developing a zero-cost NAS algorithm that quickly explores DensEMANN's search space. This should become the baseline for evaluating future DensEMANN-inspired algorithms.
- Further incorporating best model saving into DensEMANN: during the micro-algorithm's improvement and recovery stages, save the weights—and the model—that correspond to the best validation loss since the start of the stage, then reload them at the end of the stage. This could double as a method for controlling the true usefulness of filter additions.
- Comparing different ways to initialize new filters or layers, such as network morphisms [25, 26] and GradMax [10].
- Replacing the block replication method with an improved macro-algorithm, that can decide when it is more convenient to start a new block or to just add a new layer in the current one.

References

[1] Antonio García-Díaz and Hugues Bersini. DensEMANN: building a DenseNet from scratch, layer by layer and kernel by kernel. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2021. doi: 10.1109/IJCNN52387.2021.9533783. URL https://difusion.ulb.ac.be/vufind/Record/ULB-DIPOT:oai:dipot.ulb.ac.be:2013/335727/Details.

[2] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7588–7598. PMLR, 2021. URL https://proceedings.mlr.press/v139/mellor21a.html.

[3] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Comput. Surv.*, 54(4), 2021. ISSN 0360-0300. doi: 10.1145/3447582. URL https://doi.org/10.1145/3447582.

[4] Yanan Sun, Bing Xue, Mengjie Zhang, Gary G. Yen, and Jiancheng Lv. Automatically designing CNN architectures using the genetic algorithm for image classification. *IEEE Transactions on Cybernetics*, 50(9):3840–3854, 2020. doi: 10.1109/TCYB.2020.2983860. URL https://ieeexplore.ieee.org/document/9075201.

[5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL https://arxiv.org/abs/1409.1556v6.

[6] Jeon-Seong Kang, JinKyu Kang, Jung-Jun Kim, Kwang-Woo Jeon, Hyun-Joon Chung, and Byung-Hoon Park. Neural architecture search survey: A computer vision perspective. *Sensors*, 23(3), 2023. ISSN 1424-8220. doi: 10.3390/s23031713. URL https://www.mdpi.com/1424-8220/23/3/1713.

[7] Dilyara Baymurzina, Eugene Golikov, and Mikhail Burtsev. A review of neural architecture search. *Neurocomputing*, 474:82–93, 2022. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2021.12.014. URL https://www.sciencedirect.com/science/article/pii/S0925231221018439.

[8] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL http://jmlr.org/papers/v20/18-598.html.

[9] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. Hardware-aware neural architecture search: Survey and taxonomy. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4322–4329. International Joint Conferences on Artificial Intelligence Organization, 2021. doi: 10.24963/ijcai.2021/592. URL https://doi.org/10.24963/ijcai.2021/592. Survey Track.

[10] Utku Evci, Bart van Merrienboer, Thomas Unterthiner, Fabian Pedregosa, and Max Vladymyrov. GradMax: growing neural networks using gradient information. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=qjN4h_wwUO.

[11] Xin Yuan, Pedro Henrique Pamplona Savarese, and Michael Maire. Growing efficient deep networks by structured continuous sparsification. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=wb3wxCObbRT.

[12] Wei Wen, Feng Yan, Yiran Chen, and Hai Li. AutoGrow: automatic layer growing in deep convolutional networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 833–841, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3403126. URL https://doi.org/10.1145/3394486.3403126.

[13] Hanzhang Hu, John Langford, Rich Caruana, Saurajit Mukherjee, Eric Horvitz, and Debadeepta Dey. Efficient forward architecture search. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Edward A. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 10122–10131, 2019. URL http://papers.nips.cc/paper/9202-efficient-forward-architecture-search.

[14] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=H1hymrkDf.

[15] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018. ISBN 978-1-57735-800-8. URL https://dl.acm.org/doi/abs/10.5555/3504035.3504375.

[16] T. Salome and H. Bersini. An algorithm for self-structuring neural net classifiers. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, volume 3, pages 1307–1312 vol.3, 1994. doi: 10.1109/ICNN.1994.374473. URL https://ieeexplore.ieee.org/document/374473.

[17] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. URL https://openaccess.thecvf.com/content_cvpr_2017/papers/Huang_Densely_Connected_Convolutional_CVPR_2017_paper.pdf.

[18] Antonio García-Díaz and Hugues Bersini. Self-optimisation of dense neural network architectures: An incremental approach. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2020. doi: 10.1109/IJCNN48605.2020.9207416. URL https://difusion.ulb.ac.be/vufind/Record/ULB-DIPOT:oai:dipot.ulb.ac.be:2013/335725/Details.

[19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=ByME42AqK7.

[20] Rami M. Mohammad, Fadi Thabtah, and Lee McCluskey. An improved self-structuring neural network. In Huiping Cao, Jinyan Li, and Ruili Wang, editors, *Trends and Applications in Knowledge Discovery and Data Mining*, pages 35–47, Cham, 2016. Springer International Publishing. ISBN 978-3-319-42996-0. URL https://link.springer.com/chapter/10.1007/978-3-319-42996-0_4.

[21] Timur Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1 (4):365–375, 1989. doi: 10.1080/09540098908915647. URL https://doi.org/10.1080/09540098908915647.

[22] Scott Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/69adc1e107f7f7d035d7baf04342e1ca-Paper.pdf.

[23] Tin-Yau Kwok and Dit-Yan Yeung. Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE Transactions on Neural Networks*, 8(3):630–645, 1997. doi: 10.1109/72.572102. URL https://ieeexplore.ieee.org/document/572102.

[24] Sam Waugh. Dynamic learning algorithms. 1994. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bd69827d03d6f486ca0ef257f760d028ab076b5d.

[25] Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2Net: Accelerating learning via knowledge transfer. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL https://arxiv.org/abs/1511.05641v4.

[26] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 564–572, New York, New York, USA, 2016. PMLR. URL https://proceedings.mlr.press/v48/wei16.html.

[27] Mike Wynne-Jones. Node splitting: A constructive algorithm for feed-forward neural networks. In J. Moody, S. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4. Morgan-Kaufmann, 1991. URL https://proceedings.neurips.cc/paper_files/paper/1991/file/0fcbc61acd0479dc77e3cccc0f5ffca7-Paper.pdf.

[28] Shizuma Namekawa and Taro Tezuka. Evolutionary neural architecture search by mutual information analysis. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 966–972, 2021. doi: 10.1109/CEC45853.2021.9504845. URL https://ieeexplore.ieee.org/document/9504845/.

[29] Lemeng Wu, Dilin Wang, and Qiang Liu. Splitting steepest descent for growing neural architectures. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/3a01fc0853ebeba94fde4d1cc6fb842a-Paper.pdf.

[30] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas Donald Lane. Zero-cost proxies for lightweight NAS. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0cmMMy8J5q.

[31] Ekaterina Gracheva. Trainless model performance estimation based on random weights initialisations for neural architecture search. *Array*, 12:100082, 2021. ISSN 2590-0056. doi: https://doi.org/10.1016/j.array.2021.100082. URL https://www.sciencedirect.com/science/article/pii/S2590005621000308.

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015. URL https://arxiv.org/abs/1502.03167v3.

[33] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association. ISBN 9781931971331. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[34] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 2011. PMLR. URL https://proceedings.mlr.press/v15/glorot11a.html.

[35] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791. URL http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf.

[36] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of*

the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018. URL https://openaccess.thecvf.com/content_cvpr_2018/html/Zoph_Learning_Transferable_Architectures_CVPR_2018_paper.html.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: an imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[38] Jeremy Howard and Sylvain Gugger. Fastai: A layered api for deep learning. *Information*, 11 (2), 2020. ISSN 2078-2489. doi: 10.3390/info11020108. URL https://www.mdpi.com/2078-2489/11/2/108.

[39] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q. Weinberger. Memory-efficient implementation of densenets. 2017. URL https://arxiv.org/abs/1707.06990v1.

[40] Antonio García-Díaz. DensEMANN: self-constructing DenseNet with TensorFlow. URL https://github.com/AntonioGarciaDiaz/Self-constructing_DenseNet.

[41] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[42] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017. URL https://arxiv.org/abs/1708.07747v2.

[43] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. 2011. URL http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

[44] Terrance Devries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017. URL https://arxiv.org/abs/1708.04552v2.

[45] Papers with code - CIFAR-10 benchmark (image classification). URL https://paperswithcode.com/sota/image-classification-on-cifar-10.

[46] Papers with code - SVHN benchmark (image classification). URL https://paperswithcode.com/sota/image-classification-on-svhn.

[47] Papers with code - Fashion-MNIST benchmark (image classification). URL https://paperswithcode.com/sota/image-classification-on-fashion-mnist.

[48] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Los Alamitos, CA, USA, 2016. IEEE Computer Society. doi: 10.1109/CVPR.2016.90. URL https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.90.

[49] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 646–661. Springer, 2016. doi: 10.1007/978-3-319-46493-0_39. URL https://doi.org/10.1007/978-3-319-46493-0_39.

[50] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In Edwin R. Hancock Richard C. Wilson and William A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, 2016. ISBN 1-901725-59-6. doi: 10.5244/C.30.87. URL https://dx.doi.org/10.5244/C.30.87.

[51] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks, 2015. URL https://arxiv.org/abs/1505.00387v2.

[52] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=r1Ue8Hcxg.

[53] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. URL https://openaccess.thecvf.com/content_cvpr_2018/html/Zhong_Practical_Block-Wise_Neural_CVPR_2018_paper.html.

[54] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911. PMLR, 2017. URL https://proceedings.mlr.press/v70/real17a.html.

[55] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 497–504, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349208. doi: 10.1145/3071178.3071229. URL https://doi.org/10.1145/3071178.3071229.

[56] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4780–4789. AAAI Press, 2019. ISBN 978-1-57735-809-1. URL https://aaai.org/ojs/index.php/AAAI/article/view/4405.

[57] Dongzhan Zhou, Xinchi Zhou, Wenwei Zhang, Chen Change Loy, Shuai Yi, Xuesen Zhang, and Wanli Ouyang. EcoNAS: Finding proxies for economical neural architecture search. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11393–11401, 2020. doi: 10.1109/CVPR42600.2020.01141. URL https://ieeexplore.ieee.org/document/9157571.

[58] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104. PMLR, 2018. URL https://proceedings.mlr.press/v80/pham18a.html.

[59] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=S1eYHoC5FX.

[60] Niv Nayman, Asaf Noy, Tal Ridnik, Itamar Friedman, Rong Jin, and Lihi Zelnik. XNAS: Neural architecture search with expert advice. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. doi: https://proceedings.neurips.cc/paper_files/paper/2019/file/00e26af6ac3b1c1c49d7c3d79c60d000-Paper.pdf. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/00e26af6ac3b1c1c49d7c3d79c60d000-Paper.pdf.