ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230294

Shrink-Perturb Improves Architecture Mixing During Population Based Training for Neural Architecture Search

Alexander Chebykin[a;*], **Arkadiy Dushatskiy**[a], **Tanja Alderliesten**[b] **and Peter Bosman**[a, c]

[a]Centrum Wiskunde & Informatica
[b]Leiden University Medical Center, Department of Radiation Oncology
[c]Delft University of Technology
ORCiD ID: Alexander Chebykin https://orcid.org/0000-0002-3549-3533,
Arkadiy Dushatskiy https://orcid.org/0000-0003-0945-0262,
Tanja Alderliesten https://orcid.org/0000-0003-4261-7511, Peter Bosman https://orcid.org/0000-0002-4186-6666

Abstract.
In this work, we show that simultaneously training and mixing neural networks is a promising way to conduct Neural Architecture Search (NAS). For hyperparameter optimization, reusing the partially trained weights allows for efficient search, as was previously demonstrated by the Population Based Training (PBT) algorithm. We propose PBT-NAS, an adaptation of PBT to NAS where architectures are improved during training by replacing poorly-performing networks in a population with the result of mixing well-performing ones and inheriting the weights using the shrink-perturb technique. After PBT-NAS terminates, the created networks can be directly used without retraining. PBT-NAS is highly parallelizable and effective: on challenging tasks (image generation and reinforcement learning) PBT-NAS achieves superior performance compared to baselines (random search and mutation-based PBT).

1 Introduction

Neural Architecture Search (NAS) is the process of automatically finding a neural network architecture that performs well on a target task (such as image classification [25], natural language processing [21], image generation [12]). One of the key questions for NAS is the question of efficiency, since evaluating every promising architecture by fully training it would require an extremely large amount of computational resources.

Many approaches have been proposed for increasing the search efficiency: low-fidelity evaluation [45, 32], using weight sharing via a supernetwork [30, 5], estimating architecture quality via training-free metrics [28]. Typically, each approach has two stages: first, finding an architecture efficiently, then, training it (or its scaled-up version) from scratch. This final training usually requires a manual intervention (e.g., if an architecture of a cell is searched, determining how many of these cells should be used), which diminishes the benefit of an automatic approach (potentially, this could also be automated, but we are not aware of such studies in the literature). Ideally, an architecture itself (not its proxy version) should be searched on the target problem, with the search result being immediately usable after the search (such single-stage NAS approaches exist but are limited: e.g., they restrict potential search spaces [17] or require costly pretraining [5, 38]).

For the task of hyperparameter optimization (which is closely related to NAS), effective and efficient single-stage algorithms exist in the form of Population Based Training (PBT) [19] and its extensions [24, 9]. The key idea of PBT is to train many networks with different hyperparameters (a population) in parallel: as the training progresses, worse networks are replaced by copies of better ones (including the weights), with hyperparameter values explored via random perturbation. PBT is highly efficient due to the weight reuse, and due to its parallel nature: given a sufficient amount of computational resources, running PBT takes approximately the same wall-clock time as training just one network.

Determining the best way to adapt PBT to NAS is an open research question [9]: if a network architecture has been perturbed, the partly-trained weights cannot be reused (because, e.g., weights of a convolutional layer cannot be used in a linear one). The naive approach of initializing them randomly does not work well (see Section 5.3), and existing algorithms extending PBT to NAS [11, 37] sidestep the issue at the cost of parallelizability or performance (see Section 2.2).

We propose to adapt PBT to NAS by modifying the search to rely not on random perturbations but on mixing layers of the networks in the population. An example of this principle is combining an encoder and a decoder from two different autoencoder networks, ultimately obtaining a better-performing network. In this setting, the source of the weights for the changed layers is natural: they can be copied from the parent networks. Furthermore, we explore if additionally adapting the copied weights with the shrink-perturb technique [2] (reducing weight magnitude and adding noise) is helpful for achieving a successful transfer of a layer from one network to another.

For many standard tasks (such as image classification), single-objective NAS algorithms are matched by (or show only a small improvement over) the simple baseline of random search [23]. In order to make the potential benefit of PBT-NAS clear, experiments in this paper are conducted in two challenging settings: Generative Adversarial Network (GAN) training, and Reinforcement Learning (RL) for visual continuous control. We further advocate for harder tasks and search spaces in Section 6.

While our approach could potentially be extended to include hyperparameter optimization, this paper is focused on architecture search.

The contributions of this work are threefold:

* Corresponding Author. Email: a.chebykin@cwi.nl.

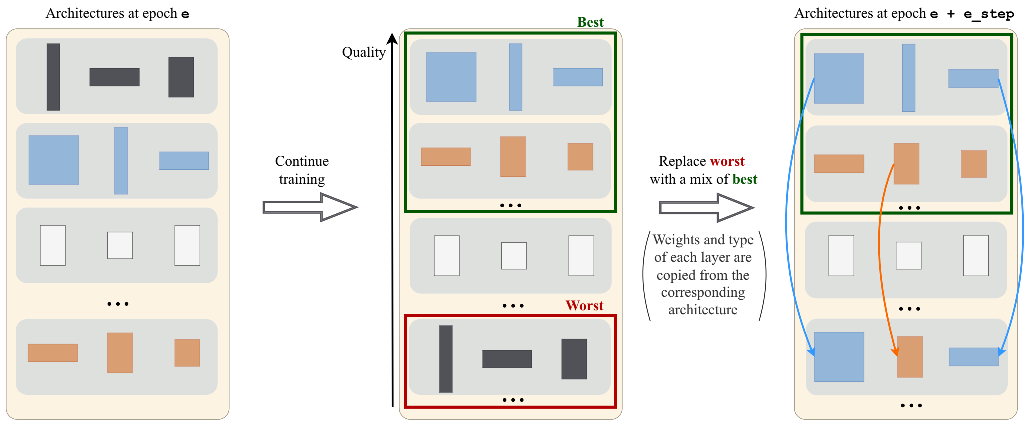

Architectures at epoch e
Continue training
Quality
Best
Worst
Replace worst with a mix of best
Weights and type of each layer are copied from the corresponding architecture
Architectures at epoch e + e_step

Figure 1. In each iteration of PBT-NAS, architectures in the population continue training for several epochs and then are sorted by performance. Every architecture from the bottom percentile is replaced with a mix of two architectures from the top percentile. During mixing, each layer is copied from one of these two architectures (weights from the architecture with worse performance are shrink-perturbed). Different shapes represent different types of layers.

1. We propose to conduct NAS by training a population of different architectures and mixing them on-the-fly to create better ones (while inheriting the weights).
2. We investigate if applying shrink-perturb [2] to the weights is a superior technique for weight inheritance compared to copying or random reinitialization.
3. Integrating these ideas, we introduce PBT-NAS, an efficient and general NAS algorithm, and evaluate it on challenging NAS search spaces and tasks.

2 Related work

2.1 Neural Architecture Search

NAS is the automatic process of finding a well-performing neural network architecture for a specific task. Already in early NAS work [45], efficiency concerns played a role: candidate architectures were trained for only a few epochs. Similar low-fidelity search methods save compute by using fewer layers [27], or only a subset of the data [32]. Another way to save compute is by utilizing a training-free metric to perform NAS without any training [28].

ENAS [30] introduced the idea of weight sharing: all candidate architectures are viewed as subsets of a supernetwork, with the weights of the common parts reused across the architectures. The final architecture is scaled up and trained from scratch. This approach greatly decreased cost of the search to just several GPU-days. DARTS [25] further increased efficiency by continuously relaxing the problem. Many approaches build upon DARTS by e.g., reducing memory usage [41] or improving performance [7]. AdversarialNAS [12] extends the approach to GAN training, outperforming previous algorithms [13].

In OnceForAll [5], a supernetwork is pretrained such that subnetworks would perform well without retraining, in AttentiveNAS [38] performance is further improved. These approaches are a good fit for multi-objective NAS (where in contrast to single-objective NAS, multiple architectures with different trade-offs between objectives such as performance and latency are searched). However, the costs for the proposed pretraining reach thousands of GPU-hours. Additionally, in any supernetwork approach, the diversity and size of the architectures are restricted by the supernetwork.

Our approach of exchanging layers and weights between different networks is distinct from the supernetwork-based weight sharing. The weights in the supernetwork are constrained to perform well in a variety of subnetworks, while in our approach, after the weights have been copied to the network with a novel architecture, they can be freely adapted to it, independently of what happens to their original version in the parent network.

The general idea of creating new architectures by modifying existing ones and reusing the weights has been explored in NAS approaches [10, 20] relying on network morphisms[6]. Network morphisms are operators that change the architecture of a neural network without influencing its functionality. Although [10, 20] successfully used morphisms, the idea was later challenged [39] with experiments demonstrating that random initialization of new layers is superior to morphisms. Morphisms are different from our work: while they create a new architecture by modifying one existing architecture, we seek to mix two distinct architectures and reuse their weights.

2.2 Population Based Training

In hyperparameter optimization, hyperparameters of neural network training, such as learning rate or weight decay, are optimized. Bayesian optimization algorithms [18] are commonly used for sequentially evaluating promising hyperparameter configurations. Other approaches include Evolutionary Algorithms [26, 24], and random search [3], a simple but reasonably good baseline.

In contrast to the approaches that train weights for each hyperparameter configuration from scratch, PBT [19] reuses partly-trained weights when exploring hyperparameters (see Section 1 for short description and [19] for details).

To the best of our knowledge, two algorithms were proposed for including architecture search into PBT: SEARL [11] and BG-PBT [37]. In SEARL, the architecture is modified by mutation, which can add a linear layer, add neurons to an existing layer, change an activation function, or add noise to the weights. We use a SEARL-like mutation as a baseline. In BG-PBT, there are multiple generations; in each generation, network architectures are sampled, initialized with random weights, and their training is sped up via distillation from the best network of the previous generation. This approach adds complexity in the form of multiple generations (the number of which must be manually determined) and using distillation (that would require adaptation to each setting, e.g., GAN training). In addition, sequential generations decrease parallelizability. Both SEARL and BG-PBT were proposed exclusively for RL tasks, while we construct PBT-NAS to be a general NAS algorithm.

2.3 *Combining several neural networks into one*

Neural networks can be combined in various ways. In evolutionary NAS [27] where weights are trained from scratch for each considered architecture, crossover is performed between encodings of architectures. Alternatively, there exist methods combining only the weights of networks that have the same architecture [36, 1]. Naively averaging the weights leads to a large loss in performance [1], which motivated these approaches to align neurons so that they would represent similar features. Averaging weights without alignment is possible if the weights of the networks are closely related. The idea of model soups [40] is to start with a pretrained model, fine-tune it with different sets of hyperparameters, and greedily search which of the fine-tuned models to average.

In our approach, we mix different architectures *together with the weights* during training, in contrast to evolutionary NAS algorithms combining only architecture encodings, and training the weights from scratch. We also avoid the additional complexity of aligning neurons, instead we continue to train the created network, and allow the gradient descent procedure to adapt the neurons to each other (which is facilitated by shrink-perturb [2], see Section 3.3).

3 Method

3.1 *Problem setting*

The goal of single-objective NAS is to find a network architecture α^* from a search space Ω that maximizes an objective function f after training the weights θ:

$$\alpha^* = \arg\max_{\alpha \in \Omega} f(\alpha; \theta) \quad (1)$$

Ω typically includes network architecture properties such as the number of layers, types of each layer, and its hyperparameters (e.g., convolution size). We will describe an architecture α by M categorical variables $\{x_i\}_{i=0..M-1}$, each taking l_i possible values. Note that in the case where more than one architecture is searched for (e.g., generator and discriminator of a GAN), we consider, for simplicity, α to include architecture parameters of all architectures.

3.2 *Algorithm overview*

In our algorithm, PBT-NAS, we follow the general structure of PBT, where N networks are trained in parallel[1]. In each iteration of the algorithm, every network is trained for e_step epochs. Then, each of the worst τ% of the networks is replaced by a mix of two networks from the best τ% (according to the objective function f). Over time, better architectures are created. Mixing architectures during training is the key component of PBT-NAS. In Section 3.3, we motivate the choice to do NAS by mixing networks. Further details of how we mix architectures are given in Section 3.4.

A visual representation of one iteration of PBT-NAS is shown in Figure 1, and the pseudocode is listed in Algorithm 1.

3.3 *Key question when modifying architecture during training: where to get the weights from?*

PBT relies on random perturbations of hyperparameters for exploring the search space while the network weights are being continuously trained. This works well when searching for hyperparameters that can be replaced independently of the weights: e.g., after changing the learning rate, the training can continue with the same weights.

[1] Note that since each network has a different architecture, it has a different training speed, so to avoid biasing the search towards models that require less training time, we use the synchronous variant of PBT.

Algorithm 1 PBT-NAS

```
Input: search space $\Omega$, number of variables $M$, population size $N$,
number of epochs $e\_total$, step size $e\_step$, selection parameter $\tau$,
probability $p$ of replacing a layer, parameters $\lambda$, $\gamma$ of shrink-perturb
1:  $pop \leftarrow \{N$ random architectures from $\Omega\}$
2:  $e \leftarrow 0$
3:  while $e < e\_total$ do
4:      for $i \leftarrow 0$ to $N-1$ do // in parallel
5:          train $pop_i$ for $e\_step$ epochs
6:          $pop_i.fitness \leftarrow$ evaluate($pop_i$)
7:      end for
8:      sort $pop$ by $fitness$
9:      $best\_nets \leftarrow$ the best $\tau$% nets
10:     $worst\_indices \leftarrow$ indices of the worst $\tau$% nets
11:     for $j$ in $worst\_indices$ do
12:         $pop_j \leftarrow$ create_architecture($best\_nets, p, M, \lambda, \gamma$)
            // the result of mixing, see Algorithm 2
13:     end for
14:     $e \leftarrow e + e\_step$
15: end while
```

However, searching for an architecture means introducing changes that impact the weights, e.g., changing the type of a layer from linear to convolutional. After such a change, the training process is disturbed: the weights of one type of layer cannot be used in another one.

To follow the paradigm of PBT and continue training the network after an architectural change, the source of the weights needs to be determined. We consider three potential approaches (Figure 2).

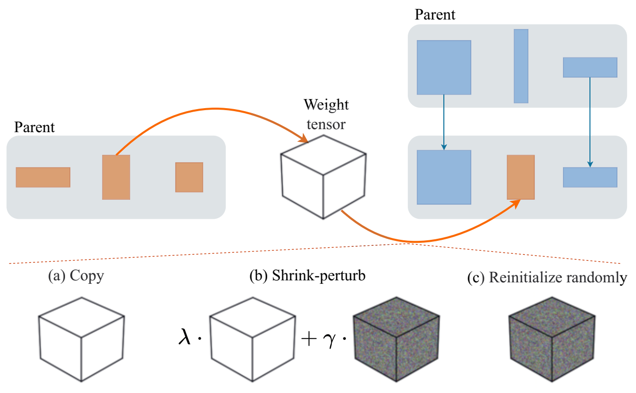


Figure 2. Three potential operations to perform on the weight tensor when copying the corresponding layer from the parent.

One approach is initializing the new weights randomly. Intuitively, this could be problematic, as replacing weights of a whole layer with random ones can substantially disrupt the learned connections between neurons across the whole network.

Instead of being initialized randomly, the weights of the modified part can come from another network in the population. If the new value of the type of the layer is not generated randomly but copied from another solution, the corresponding layer weights can be copied from it too. Straightforward weight copying may be better than random initialization but it faces the following issue: even though the layers at the same depth of different networks should perform similar transformations (when trained on the same task), the actual data

Algorithm 2 `create_architecture`

```
Input: set of networks to potentially mix $nets$, probability $p$ of replacing a layer, number of variables $M$, parameters $\lambda$, $\gamma$ of shrink-perturb
1: $net_1, net_2 \leftarrow$ randomly sample from $nets$
2: if $net_1.fitness < net_2.fitness$ then
3:     $net_1, net_2 \leftarrow net_2, net_1$ // sort by fitness
4: end if
5: $net_{new} \leftarrow$ copy($net_1$)
6: for $i = 0$ to $M - 1$ do // iterate over architecture variables
7:     if random_uniform() $< p$ then
8:         $net_{new}.\alpha_i \leftarrow net_2.\alpha_i$ // copy the value of the variable
9:         if $\exists net_2.W^i$ then
10:            // if the variable is a layer, copy and modify its weights
11:            $W_{new} \leftarrow$ copy($net_2.W^i$)
12:            shrink_perturb($W_{new}, \lambda, \gamma$)
13:            $net_{new}.W^i \leftarrow W_{new}$
14:        end if
15:    end if
16: end for
17: return $net_{new}$
```

representations in each network are likely to be different. The copied weights would need to be adapted to a different representation space, but it might be difficult for gradient descent to adapt them quickly.

Shrink-perturb [2] is potentially helpful in this scenario. It was motivated by the observation that in online learning, continuing training from already trained weights when new data comes in can be worse than retraining from scratch using all the available data. Shrink-perturb consists of modifying the weights of a neural network by *shrinking* (multiplying by a constant λ) and *perturbing* them (adding noise multiplied by a constant γ; a new initialization of the network architecture is used as the source of noise).

Applying shrink-perturb to the copied weights is the middle ground between copying the weights as-is, and initializing them randomly. This preserves some useful information in the weights, while also potentially making their adaptation to the new architecture easier.

3.4 Mixing networks

Algorithm 2 shows our procedure for creating a new network. Firstly, two parent networks are randomly sampled from the top τ percentile of the population. An offspring solution is created by copying the better parent, and replacing with probability p each layer with the layer from the worse parent (including the weights, which are shrink-perturbed). Our mixing is a version of uniform crossover [34] where only one offspring solution is produced. Note that our mixing requires that layers in the same position can be substituted for each other (i.e., the output can be used as the input of the next layer), with the architecture remaining valid after a layer is replaced. We further discuss this limitation in Section 6.

Unlike existing approaches to combining neural networks (see Section 2.3), we do not expect (or need) the new network to perform well right away. Instead, it will be trained for several epochs in the next iteration of PBT-NAS, the same as the other networks in the population.

4 Experiment setup

4.1 General

We evaluate PBT-NAS on two tasks known to require careful tuning of network architecture and hyperparameters: GAN training and RL for visual control. In these settings, architecture can strongly influence performance [14, 33]. We consider non-trivial architecture search spaces, see Sections 4.2 and 4.3. We would like to emphasize that achieving a state-of-the-art result on the chosen tasks is not our goal, instead we aim to demonstrate the feasibility of architecture search via simultaneous training and architecture mixing on tasks where performance strongly depends on architecture.

Hyperparameters of PBT-NAS are population size N, step size e_step, selection parameter τ (we use the default value from PBT, 25%, in all experiments), probability p of replacing a layer (which is also set to 25%). We aim to avoid unnecessary hyperparameter tuning to see if our approach is robust enough to perform well without it and to save computational resources.

The experiments were run in a distributed way, the details on used hardware and on GPU-hour costs of experiments are given in [46] Appendix E. The algorithms used the amount of compute equivalent to training N networks. Every experiment was run three times, we report the mean and standard deviation of the performance of the best solution from each run. We use the Wilcoxon signed-rank test with Bonferroni correction for statistical testing (target p-value 0.05, 4 tests, corrected p 0.0125, mentions of statistical significance in the text imply smaller p, all p-values are reported in [46] Appendix B). Our code is available at https://github.com/AwesomeLemon/PBT-NAS, it includes configuration files for all experiments.

4.2 GANs

In AdversarialNAS [12], the authors describe searching for a GAN architecture (for unconditional generation) in a search space where random search achieved poor results — this motivated us to adopt this search space, which we refer to as `Gan`. In AdversarialNAS, both generator and discriminator architectures are searched for but we noticed that in the official implementation, the searched discriminator is discarded, and an architecture from the literature [13] is used instead. This prompted us to create an extended version of the search space (which we call `GanHard`) that includes discriminator architectures resembling the one manually selected by the AdversarialNAS authors. AdversarialNAS cannot be used to search in `GanHard` because some of the options cannot be searched for via continuous relaxation (one example is searching whether a layer should downsample: since output tensors with and without downsampling have different dimensions, a weighted combination cannot be created).

Specifics of search spaces are not critical for our research, so we give condensed descriptions here, see [46] Appendix D and our code for more details.

In `Gan`, operations for three DARTS-like [25] cells are searched (each cell is a Directed Acyclic Graph (DAG) with operations on the edges; in contrast to DARTS, each cell may have a different architecture). Additionally, inspecting the code of AdversarialNAS showed that the output of some cells is pointwise summed with a projection of a part of a latent vector. Each such projection is a single linear layer mapping a part of a latent vector to a tensor of the same dimensionality as the cell output: $(\#channels, width, height)$. These projections contain many parameters and are therefore an important part of the architecture. In the code of AdversarialNAS, these projections are adjusted for each dataset. As to the discriminator, the architecture from [13] is used.

Next, we describe `GanHard`. In `GanHard`, the parameters of the projections in the generator can be searched for. We additionally treat the layer mapping latent vector to the input of the generator as a projection, since it is conceptually similar. The projections (one per cell) can be enabled or disabled, except for the first one (connected to generator input) which is always enabled. A projection can take as input either the whole latent vector or the corresponding one-third of it (the first third of the vector for the first projection, etc.). There are three options for the spatial dimensions of the output of a projection: *target* (equal to the output dimensions of the corresponding cell), *smallest* (equal to the input dimensions of the first cell), and *previous* (equal to the output dimensions of the previous cell). Since the projection output is summed with the cell output pointwise, the dimensions need to match, which is not the case for the last two options. To upsample the tensor to the target dimensions, either $bilinear$ or $nearest\_neighbour$ interpolation is used, which is also a part of the search space. Finally, given that a projection is a large linear layer with potentially millions of parameters (which makes overfitting plausible), we introduce an option for a dropout layer in the projection, with possible parameters $0.0, 0.1, 0.2$.

The discriminator search space in `GanHard` is based on the one in AdversarialNAS, the discriminator has 4 cells (each being a DAG with two branches), each cell has a downsampling operation in the end. However, we noticed that the fixed architecture from [13] that is used for final training of AdversarialNAS downsamples only in the first two cells. Additionally, in the first cell, the input is downsampled rather than the output. We amend the discriminator search space to contain a similar architecture. Firstly, we search whether each cell should downsample or not. Secondly, we add options for downsampling operations that are performed at the start of each branch rather than at the end of them. To enrich the search space further, we add two more nodes to each cell.

The number of variables in `Gan` is 21 and the size of the search space is $\approx 3.4 \cdot 10^{19}$. In `GanHard`, there are 72 variables (32 for the generator, 40 for the discriminator), and the size of the search space is $\approx 2.9 \cdot 10^{53}$.

Following AdversarialNAS, we run the experiments on CIFAR-10 [22] and STL-10 [8], using both `Gan` and `GanHard`. In AdversarialNAS, the networks were trained for 600 epochs. We reduce that number to 300 epochs to save computation time (preliminary experiments showed diminishing returns to longer training), for the other hyperparameters, the same values as in AdversarialNAS are used.

The Frechet Inception Distance (FID) [16] is a commonly used metric for measuring GAN quality. We use its negation as the objective function, computing it on 5,000 images during the search. For reporting the final result, the FID for the best network is computed on 50,000 images. We additionally report the Inception Score (IS) [31], another common metric of GAN quality. The idea behind both FID and IS is to compare representations of real and generated images.

Based on preliminary experiments, the population size N is set to 24, and e_step is set to 10.

4.3 RL

We build upon DrQ-v2 [43], a model-free RL algorithm for visual continuous control. DrQ-v2 achieves great results on the Deep Mind Control benchmark [35], solving many tasks. Searching for architectures for solved tasks is not necessary, therefore for our experiments we chose tasks where DrQ-v2 did not achieve the maximum possible performance: Quadruped Run, Walker Run, Humanoid Run.

DrQ-v2 is an actor-critic algorithm with three components: *1)* an encoder that creates a representation of the pixel-based environment observation, *2)* an actor that, given the representation, outputs probabilities of actions, and *3)* a critic that, given the representation, estimates the Q-value of the state-action pair (the critic contains two networks because double Q-learning is used).

We design the search space to include the architectures of all the components of DrQ-v2. Each network has three searchable layers. For the encoder, the options are Identity, Convolution {3x3, 5x5, 7x7}, ResNet [15] block {3x3, 5x5, 7x7}, Separable convolution {3x3, 5x5, 7x7}. For the actor and both networks of the critic, the available layers are Identity, Linear, and Residual [4] with multiplier 0.5 or 2.0. Additionally, we search whether to use Spectral Normalization [29] (for each network separately) and which activation function to use in each layer (options: Identity, Tanh, ReLU, Swish). We also search the dimensionality of representation: in DrQ-v2 it was set to either 50 or 100 depending on the task, we have $25, 50, 100$, and 150 as options.

There are 36 variables in total, the search space size is $\approx 4.6 \cdot 10^{21}$.

The hyperparameters of DrQ-v2 are used without additional tuning. For the Walker and Humanoid tasks, DrQ-v2 uses a replay buffer of size 10^6. Our servers do not have enough RAM to allow for such a buffer size when many agents are training in parallel, therefore for these tasks, we use a shared replay buffer (proposed in [11]): different agents can learn from the experiences of each other. To run in a distributed scenario with no shared storage, the buffers are only shared by the networks on the same machine. For fairness, all the baselines also use a shared buffer per machine.

Based on preliminary experiments, the population size N is set to 12. For the Quadruped and the Walker tasks, the DrQ-v2 agent used $3 \cdot 10^6$ frames. We use the same number of frames per agent, which means that N times more total frames are used. For the Humanoid task, $3 \cdot 10^7$ frames were used in DrQ-v2, we use only $1.5 \cdot 10^7$ per agent to save computation time. For uniformness of notation with GANs, we also use "epoch" in the context of RL, one epoch is defined as 10^4 frames. This means that 300 epochs are used for the Quadruped and Walker tasks, the same as for GANs, and e_step is also set to 10. Similar to BG-PBT [37], our preliminary experiments showed that having longer periods without selection at the start of the training is beneficial, therefore during the first half of the training, e_step is doubled from 10 to 20 epochs for the Quadruped and Walker tasks. Since for Humanoid only half the training is performed (in terms of frames per agent), the step size is fixed at 100 epochs (scaled up from 10 proportionally to the increase in the number of frames).

4.4 Baselines

We consider two general baselines that parallelize well and that can search in the proposed challenging search spaces.

1. **Random search.** N architectures are randomly sampled and trained.
2. **SEARL-like mutation**. In order to fairly evaluate the performance of a mutation-based architecture search approach like SEARL [11], we replace the mixing operator of PBT-NAS with the mutation operator from SEARL, adapting it to be applicable to both GAN and RL settings: with equal probability, either *(a)* one variable in the architecture encoding is resampled, *(b)* weights are mutated using the procedure from SEARL, or *(c)* no change is performed.

AdversarialNAS is a specialized baseline only capable of searching in `Gan`. In [12], the performance for only one seed was reported. We run the official implementation with 5 seeds and report the mean and standard deviation of performance.

Table 1. Results for GAN training (mean ± st. dev.). The best value in each column is in bold.

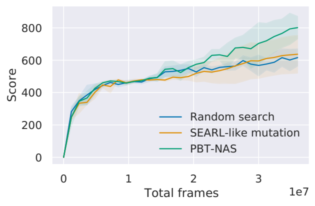
Algorithm	CIFAR-10				STL-10	
	`Gan`		`GanHard`		`GanHard`	
	FID ↓	IS ↑	FID ↓	IS ↑	FID ↓	IS ↑
AdversarialNAS [12]	$12.29_{\pm 0.80}$	$8.47_{\pm 0.14}$	—	—	—	—
Random search	$13.39_{\pm 0.28}$	$8.22_{\pm 0.15}$	$16.79_{\pm 0.97}$	$7.80_{\pm 0.14}$	$28.58_{\pm 1.77}$	$9.33_{\pm 0.18}$
SEARL-like mutation [11]	$13.78_{\pm 1.02}$	$8.38_{\pm 0.05}$	$15.72_{\pm 2.22}$	$\mathbf{8.26}_{\pm 0.21}$	$26.94_{\pm 0.93}$	$9.66_{\pm 0.27}$
PBT-NAS	$\mathbf{12.21}_{\pm 0.16}$	$\mathbf{8.63}_{\pm 0.17}$	$\mathbf{13.25}_{\pm 1.64}$	$8.25_{\pm 0.27}$	$\mathbf{25.11}_{\pm 0.94}$	$\mathbf{9.71}_{\pm 0.09}$

5 Results

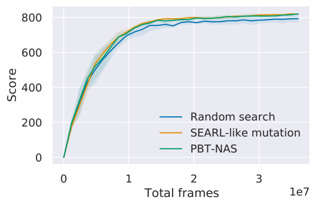
5.1 PBT-NAS vs. the baselines

As can be seen in Table 1, PBT-NAS achieves the best performance among all tested approaches in all GAN settings[2]. The improvements in FID over both random search and SEARL-based mutation are statistically significant. Despite the claim of [12] that random search performs poorly in `Gan`, we find that the gap between it and AdversarialNAS [12] on CIFAR-10 is small, and the difference between all algorithms is overall not large. The decreased performance of random search in `GanHard` shows that `GanHard` is indeed a more challenging search space. The results of searching in this space for CIFAR-10 and STL-10 show a clear improvement of PBT-NAS over the baselines in terms of FID. IS is better in the majority of settings.

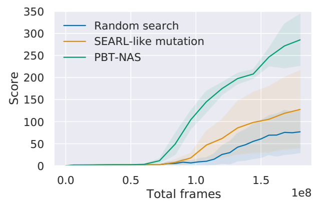
PBT-NAS is also the best among alternatives on RL tasks, achieving better anytime performance, as shown in Figure 3 (the improvements in score over both random search and SEARL-based mutation are statistically significant). For Walker Run, there is no meaningful difference between algorithms, as the task is solved by all tested approaches, demonstrating that for differences between performance of the algorithms to be clear, both the RL task and the search space need to be of significant complexity.



(a) Quardruped Run



(b) Walker Run



(c) Humanoid Run

Figure 3. Results for RL tasks, mean ± st. dev. (shaded area).

5.2 Mixing networks is better than cloning good networks

In order to show that creating new architectures makes a difference, we run a "No mixing" ablation: every component of PBT-NAS is kept the same, except that a new model is created by mixing a well-performing model with itself (rather than with another well-performing model). This way, no new architecture is produced, but the other benefits of PBT-NAS remain (e.g., replacing poorly-performing networks with well-performing ones). As seen in Table 2, this degrades the performance, clearly showing the impact that creating a better architecture can have.

[2] When searching in `Gan` for STL-10, we faced reproducibility issues (despite using the official implementation), see [46] Appendix C for results and discussion.

Table 2. Results of ablation studies (mean ± st. dev.). The best value in each column is in bold.

Algorithm	FID ↓ (CIFAR-10, `GanHard`)	Score ↑ (Quadruped Run)
PBT-NAS (default)	$\mathbf{13.25}_{\pm 1.64}$	$\mathbf{801}_{\pm 70}$
No mixing	$14.90_{\pm 1.29}$	$672_{\pm 53}$
Shrink-perturb coefficients:		
[1, 0] — copy exactly	$14.94_{\pm 0.56}$	$699_{\pm 4}$
[0, 1] — reinitialize randomly	$15.06_{\pm 2.59}$	$532_{\pm 62}$

5.3 Shrink-perturb is the superior way of weight inheritance

Table 2 shows that copying weights from the donor without change (shrink-perturb parameters $[1, 0]$), or replacing them with random weights (shrink perturb $[0, 1]$) leads to worse results in comparison to the usage of shrink-perturb. Thus, in our settings, using shrink-perturb is the best method to inherit the weights. The default parameters of shrink-perturb from [44] ($[0.4, 0.1]$) worked well in PBT-NAS without any tuning.

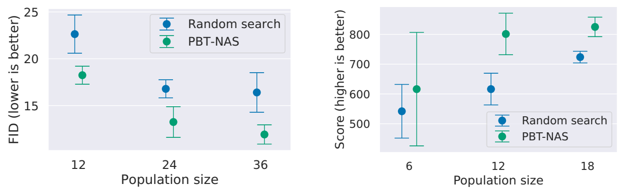
In [44], shrink-perturb was found to benefit performance, thus raising the question if using it gives PBT-NAS an unfair advantage that is not related to NAS. In order to test this, we added shrink-perturb to random search. As shown in Table 3, performance deteriorates, indicating that using shrink-perturb with default parameters in our setting is not helpful outside the context of NAS.

Table 3. The effect of using shrink-perturb in random search

Use shrink-perturb in random search	FID ↓ (CIFAR-10, `GanHard`)	Score ↑ (Quadruped Run)
No (default)	$16.79_{\pm 0.97}$	$616_{\pm 53}$
Yes	$22.39_{\pm 2.64}$	$498_{\pm 30}$

5.4 Increasing population size improves performance

We design our algorithm to be highly parallel and scalable. Figure 4 demonstrates that as the population size increases, the performance strictly improves (although diminishing returns can be observed). Given enough GPUs, the increased population size will not meaningfully increase wall-clock time, since every population member can be evaluated in parallel.



(a) CIFAR-10, GanHard (FID ↓) (b) Quadruped Run (Score ↑)

Figure 4. Impact of scaling population size, mean ± st. dev.

5.5 *Model soups*

As mentioned in Section 2.3, the idea of a model soup [40] is to improve performance by averaging weights of closely-related neural networks. As such, it seems like an especially good fit for the PBT setting: although the networks in the population start from different weights (and different architectures in the case of PBT-NAS), as worse networks are replaced by offspring of better networks, the population gradually converges. Since creating a model soup is done after training and requires a negligible amount of computation (evaluating at most N models), its inclusion into PBT-like algorithms could give an almost free performance improvement. Therefore, we create model soups following the greedy algorithm from [40].

Table 4 shows that soups improve GAN FID by approximately 0.4 points. For RL, however, there is no improvement when both the encoder and the actor are averaged (Table 5). We hypothesize that different actors may have dissimilar internal representations implementing different behaviour logic, unlike the encoders that only convert pixel inputs into representations. Therefore, we tried to separately average encoders, or actors. The results with averaged encoders are the best overall but they still do not lead to improved performance. For Walker Run, the task where performance is saturated, there is no difference between settings.

Table 4. The difference in metrics between a model soup and the best individual model (GAN), mean ± st. dev.

Dataset	GanHard	
	Δ FID ↓	Δ IS ↑
CIFAR-10	$-0.48_{\pm 0.34}$	$0.07_{\pm 0.05}$
STL-10	$-0.35_{\pm 0.26}$	$0.04_{\pm 0.25}$

Table 5. The difference in score between a model soup and the best individual model (RL), mean ± st. dev.

What to average	Δ Score ↑		
	Quadruped	Walker	Humanoid
Encoder	$-6_{\pm 10}$	$-1_{\pm 4}$	$-23_{\pm 19}$
Actor	$-196_{\pm 275}$	$-4_{\pm 7}$	$-244_{\pm 32}$
Both	$-142_{\pm 193}$	$0_{\pm 6}$	$-223_{\pm 20}$

Previously, soups were only demonstrated for classification tasks, so it is interesting to see that they could also be beneficial in GANs. While no improvement was seen for RL, the fact that only the vision-related network, the encoder, could be averaged without large performance degradation hints at the limitations of the technique.

6 Discussion

We have introduced PBT-NAS, a NAS algorithm that creates new architectures by simultaneously training and mixing a population of neural networks. PBT-NAS brings the efficiency of PBT (designed for hyperparameter optimization) to NAS, providing a novel way to search for architectures. As computation power grows, especially in the form of multiple affordable GPUs, having parallelizable and scalable algorithms such as PBT-NAS becomes more important. At the same time, this computation power is not limitless, and reusing the partly-trained weights during architecture search is important from the perspective of search efficiency.

Currently, a large amount of effort in single-objective NAS research is directed at searching classifier architectures in cell-based search spaces, which are quite restrictive, and where random search achieves competitive results [23, 42]. We think that pivoting to more challenging search spaces and tasks could lead to NAS having a larger impact (e.g., in constructing state-of-the-art architectures, which is still mostly done by hand), and to comparisons between NAS algorithms leading to clearer differences. In Section 5.1, we showed that PBT-NAS could search in the challenging GanHard space, where an existing efficient algorithm, AdversarialNAS, could not be applied.

One limitation of exchanging layers during training is the requirement that different layer options (in the same position) need to be interoperable: the activation tensors they produce should be possible for the next layer to take as input (so that after replacing a layer, the architecture remains valid). This means that the number of neurons can be searched only when it does not influence the output shape. This could be addressed by e.g., duplicating neurons if there are too few of them and removing excessive ones if there are too many. Another limitation arises due to the greedy nature of PBT-NAS: architectures are selected based on their intermediate performance, and, therefore, suboptimal architectures can be selected when early performance of an architecture is not representative of the final one.

Achieving good performance in different tasks with minimal hyperparameter tuning is a desirable property for a NAS algorithm. We used hyperparameters from the literature without tuning both in GAN training and in RL, as well as relying on default selection strategy from PBT. PBT-NAS outperformed baselines despite using these default values, tuning them could potentially further improve the results.

7 Conclusion

In this paper we designed and evaluated PBT-NAS, a novel way to search for an architecture by mixing different architectures while they are being trained. We find that adapting the weights with the shrink-perturb technique during mixing is advantageous compared to copying or randomly reinitializing them.

PBT-NAS is shown to be effective on challenging tasks (GAN training, RL), where it outperformed considered baselines. At the same time, it is efficient, requiring training of only tens of networks to explore large search spaces. The algorithm is straightforward, parallelizes and scales well, and has few hyperparameters.

While in this work only NAS was considered, in the future, PBT-NAS could be adapted to simultaneously search for hyperparameters of neural network training, and of the algorithm itself, both of which would be necessary in order to fully automate the process of neural network training.

Acknowledgements

This work is part of the research projects DAEDALUS (funded via the Open Technology Programme of the Dutch Research Council (NWO), project number 18373; part of the funding is provided by Elekta and ORTEC LogiqCare) and OPTIMAL (funded by NWO, project OCENW.GROOT.2019.015).

References

[1] Samuel K Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa, 'Git re-basin: Merging models modulo permutation symmetries', *arXiv preprint arXiv:2209.04836*, (2022).

[2] Jordan Ash and Ryan P Adams, 'On warm-starting neural network training', *Advances in Neural Information Processing Systems*, (2020).

[3] James Bergstra and Yoshua Bengio, 'Random search for hyper-parameter optimization.', *JMLR*, **13**(2), (2012).

[4] Nils Bjorck, Carla P Gomes, and Kilian Q Weinberger, 'Towards deeper deep reinforcement learning with spectral normalization', *Advances in Neural Information Processing Systems*, **34**, 8242–8255, (2021).

[5] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han, 'Once for all: Train one network and specialize it for efficient deployment', in *International Conference on Learning Representations*, (2020).

[6] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens, 'Net2net: Accelerating learning via knowledge transfer', *arXiv:1511.05641*, (2015).

[7] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian, 'Progressive DARTS: Bridging the optimization gap for nas in the wild', *International Journal of Computer Vision*, **129**, 638–655, (2021).

[8] Adam Coates, Andrew Ng, and Honglak Lee, 'An analysis of single-layer networks in unsupervised feature learning', in *Proceedings of AISTATS*. JMLR Workshop and Conference Proceedings, (2011).

[9] Valentin Dalibard and Max Jaderberg, 'Faster improvement rate population based training', *arXiv preprint arXiv:2109.13800*, (2021).

[10] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter, 'Efficient multi-objective neural architecture search via lamarckian evolution', *arXiv preprint arXiv:1804.09081*, (2018).

[11] Jörg KH Franke, Gregor Köhler, André Biedenkapp, and Frank Hutter, 'Sample-efficient automated deep reinforcement learning', *arXiv preprint arXiv:2009.01555*, (2020).

[12] Chen Gao, Yunpeng Chen, Si Liu, Zhenxiong Tan, and Shuicheng Yan, 'AdversarialNAS: Adversarial neural architecture search for GANs', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5680–5689, (2020).

[13] Xinyu Gong, Shiyu Chang, Yifan Jiang, and Zhangyang Wang, 'AutoGAN: Neural architecture search for generative adversarial networks', in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3224–3234, (2019).

[14] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye, 'A review on generative adversarial networks: Algorithms, theory, and applications', *IEEE Transactions on Knowledge and Data Engineering*, (2021).

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, 'Deep residual learning for image recognition', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, (2016).

[16] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter, 'GANs trained by a two time-scale update rule converge to a local nash equilibrium', *Advances in Neural Information Processing Systems*, **30**, (2017).

[17] Shoukang Hu, Sirui Xie, Hehui Zheng, Chunxiao Liu, Jianping Shi, Xunying Liu, and Dahua Lin, 'DSNAS: Direct neural architecture search without parameter retraining', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, (2020).

[18] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown, 'Sequential model-based optimization for general algorithm configuration', in *Learning and Intelligent Optimization: 5th International Conference*, pp. 507–523. Springer, (2011).

[19] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al., 'Population based training of neural networks', *arXiv preprint arXiv:1711.09846*, (2017).

[20] Haifeng Jin, Qingquan Song, and Xia Hu, 'Auto-Keras: An efficient neural architecture search system', in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, (2019).

[21] Nikita Klyuchnikov, Ilya Trofimov, Ekaterina Artemova, Mikhail Salnikov, Maxim Fedorov, Alexander Filippov, and Evgeny Burnaev, 'NAS-Bench-NLP: neural architecture search benchmark for natural language processing', *IEEE Access*, **10**, 45736–45747, (2022).

[22] Alex Krizhevsky and Geoffrey Hinton, 'Learning multiple layers of features from tiny images', *Master's thesis, University of Toronto*, (2009).

[23] Liam Li and Ameet Talwalkar, 'Random search and reproducibility for neural architecture search', in *Uncertainty in Artificial Intelligence*, pp. 367–377. PMLR, (2020).

[24] Jason Liang, Santiago Gonzalez, Hormoz Shahrzad, and Risto Miikkulainen, 'Regularized evolutionary population-based training', in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 323–331, (2021).

[25] Hanxiao Liu, Karen Simonyan, and Yiming Yang, 'DARTS: Differentiable architecture search', *arXiv preprint arXiv:1806.09055*, (2018).

[26] Ilya Loshchilov and Frank Hutter, 'CMA-ES for hyperparameter optimization of deep neural networks', *arXiv:1604.07269*, (2016).

[27] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf, 'NSGA-Net: neural architecture search using multi-objective genetic algorithm', in *Proceedings of the Genetic and Evolutionary Computation Conference*, (2019).

[28] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley, 'Neural architecture search without training', in *International Conference on Machine Learning*, pp. 7588–7598. PMLR, (2021).

[29] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida, 'Spectral normalization for generative adversarial networks', *arXiv preprint arXiv:1802.05957*, (2018).

[30] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean, 'Efficient neural architecture search via parameters sharing', in *International Conference on Machine Learning*, pp. 4095–4104. PMLR, (2018).

[31] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen, 'Improved techniques for training GANs', *Advances in neural information processing systems*, **29**, (2016).

[32] Jae-hun Shim, Kyeongbo Kong, and Suk-Ju Kang, 'Core-set sampling for efficient neural architecture search', *arXiv preprint arXiv:2107.06869*, (2021).

[33] Samarth Sinha, Homanga Bharadhwaj, Aravind Srinivas, and Animesh Garg, 'D2rl: Deep dense architectures in reinforcement learning', *arXiv preprint arXiv:2010.09163*, (2020).

[34] Gilbert Syswerda et al., 'Uniform crossover in genetic algorithms.', in *ICGA*, volume 3, pp. 2–9, (1989).

[35] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al., 'Deepmind control suite', *arXiv preprint arXiv:1801.00690*, (2018).

[36] Thomas Uriot and Dario Izzo, 'Safe crossover of neural networks through neuron alignment', in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 435–443, (2020).

[37] Xingchen Wan, Cong Lu, Jack Parker-Holder, Philip J Ball, Vu Nguyen, Binxin Ru, and Michael Osborne, 'Bayesian generational population-based training', in *International Conference on Automated Machine Learning*, pp. 14–1. PMLR, (2022).

[38] Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra, 'Attentivenas: Improving neural architecture search via attentive sampling', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6418–6427, (2021).

[39] 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*, pp. 833–841, (2020).

[40] Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, et al., 'Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time', in *International Conference on Machine Learning*, pp. 23965–23998. PMLR, (2022).

[41] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong, 'PC-DARTS: Partial channel connections for memory-efficient architecture search', in *International Conference on Learning Representations*, (2020).

[42] Antoine Yang, Pedro M Esperança, and Fabio M Carlucci, 'NAS evaluation is frustratingly hard', in *International Conference on Learning Representations*, (2020).

[43] Denis Yarats, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto, 'Mastering visual continuous control: Improved data-augmented reinforcement learning', in *International Conference on Learning Representations*, (2022).

[44] Sheheryar Zaidi, Tudor Berariu, Hyunjik Kim, Jörg Bornschein, Claudia Clopath, Yee Whye Teh, and Razvan Pascanu, 'When does re-initialization work?', (arXiv:2206.10011), (Jun 2022).

[45] Barret Zoph and Quoc V Le, 'Neural architecture search with reinforcement learning', *arXiv preprint arXiv:1611.01578*, (2016).

[46] Alexander Chebykin, Arkadiy Dushatskiy, Tanja Alderliesten and Peter A. N. Bosman, 'Shrink-Perturb Improves Architecture Mixing during Population Based Training for Neural Aschitecture Search', arXiv preprint arXiv:2307.15621, (2023).