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

Alexander Chebykin\textsuperscript{a,∗}, Arkadiy Dushatskiy\textsuperscript{a}, Tanja Alderliesten\textsuperscript{b} and Peter Bosman\textsuperscript{a, c}

\textsuperscript{a}Centrum Wiskunde & Informatica
\textsuperscript{b}Leiden University Medical Center, Department of Radiation Oncology
\textsuperscript{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.
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 $\alpha^*$ from a search space $\Omega$ that maximizes an objective function $f$ after training the weights $\theta$:

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

$\Omega$ 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 $\alpha$ 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, $\alpha$ 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\(^3\). In each iteration of the algorithm, every network is trained for $e_{\text{step}}$ epochs. Then, each of the worst $\tau\%$ of the networks is replaced by a mix of two networks from the best $\tau\%$ (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.

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).

\(^3\) 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 $e_{\text{total}}$, step size $e_{\text{step}}$, selection parameter $\tau$, probability $p$ of replacing a layer, parameters $\lambda$, $\gamma$ of shrink-perturb

1: $\text{pop} \leftarrow \{N \text{ random architectures from } \Omega\}$
2: $e \leftarrow 0$
3: while $e < e_{\text{total}}$ do
4:   for $i \leftarrow 0$ to $N - 1$ do // in parallel
5:     train $\text{pop}_i$ for $e_{\text{step}}$ epochs
6:     $\text{pop}_i.\text{fitness} \leftarrow \text{evaluate}(\text{pop}_i)$
7:   end for
8:   sort $\text{pop}$ by $\text{fitness}$
9:   $\text{best}_\text{nets} \leftarrow \text{the best } \tau\% \text{ nets}$
10:  $\text{worst}_\text{indices} \leftarrow \text{indices of the worst } \tau\% \text{ nets}$
11:  for $j$ in $\text{worst}_\text{indices}$ do
12:     $\text{pop}_j \leftarrow \text{create Architecture}(\text{best}_\text{nets}, p, M, \lambda, \gamma)$ // the result of mixing, see Algorithm 2
13: end for
14: $e \leftarrow e + e_{\text{step}}$
15: end while

---

**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 λ, γ of shrink-perturb
1: net1, net2 ← randomly sample from nets
2: if net1.fitness < net2.fitness then
3:   net1, net2 ← net2, net1 // sort by fitness
4: end if
5: netnew ← copy(net1)
6: for i = 0 to M - 1 do // iterate over architecture variables
7:   if random_uniform() < p then
8:     netnew, αi ← net2, αi // copy the value of the variable
9:   if ∃net2, W then
10:     // if the variable is a layer, copy and modify its weights
11:     Wnew ← copy(net2, W)
12:     shrink_perturb(Wnew, λ, γ)
13:     netnew, Wnew ← Wnew
14: end if
15: end if
16: end for
17: return netnew

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 \( e_{\text{step}} \) 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 \( e_{\text{step}} \) 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, \( e_{\text{step}} \) 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 PB-T 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.
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. 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.

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.

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, 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.

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.
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