Supplementary Material

Table of Contents

A Access to the benchmark

The source code and dataset are hosted under this URL : `https://github.com/SamsungLabs/blox`.

The source code and dataset are licensed by the CC BY-NC (`https://creativecommons.org/licenses/by-nc/4.0/`) license. On the above url, this license is described as follows:

1. Copy and redistribute the material in any medium or format.
2. Remix, transform, and build upon the material for non-commercial purposes.
3. The licensor cannot revoke these freedoms as long as you follow the license terms.

B Dataset generation

Details to produce the dataset and experimental results are described below. The instructions, source code and dataset can be found in the official repository.

Table 3: Information included in each part of the Blox dataset. Each section contains exactly 91,125 entries.

Section	Information
Basic *(one per tr. seed)*	Top1 validation accuracy Top1 test accuracy Training time
Extended *(one per tr. seed)*	Top1 training accuracy Top5 training accuracy Top5 validation accuracy Top5 test accuracy Training loss Validation loss Test loss
Static	FLOPs Number of parameters Architecture vector
Benchmark *(one per device)*	Latency
Environment *(one per tr. seed)*	Python version GPU model Driver version PyTorch version Codebase version

Table 4: Training hyperparameters used throughout this work.

	Normal	Distillation	Fine-tuning
Optimizer	SGD	Adam	Adam
Loss	CE	MSE	KD (α = 0.9)
T	N/A	N/A	4
Epochs	200	[1,10,50]	[10,50,200]
Momentum	0.9	N/A	N/A
Betas	N/A	(0.9, 0.999)	(0.9, 0.999)
LR schedule	cosine	cosine	cosine
Initial LR	0.01	0.005	5e-5
Final LR	0	0	0
Weight decay	0.0005	0.0005	0.0005
Batch size	256	256	256
H. flip	0.5	✗	0.5
Cutout	1/16/1.0	✗	1/16/1.0
RandAug	14/2	✗	14/2
Pad & crop	4	✗	4

B.1 Benchmark metrics and training environment

Table 3 summarizes the metrics and environment included in the Blox benchmark. All the metrics can be queried via the API provided with the code. In the dataset, we include the information obtained by *Normal* training – training and validation metrics (loss and accuracy) are logged at each epoch. Test metrics and training time are logged at the end of training for the best model. We also save the static information consisting of FLOPs, the number of parameters and an example architecture vector related with the entry (note that more than one architectural vector might map to the same architecture, therefore all entries across all parts of Blox are stored using hashes of graphs representing each model, similar to [13]). Benchmarking information for each model are included as separate sections – one file per benchmarking setting (device, input size, etc.). In addition, information about the training environment are included for reproducibility. That includes random seed, GPU used, versions of the OS, codebase, driver, etc.). The dataset is fully modular, meaning that different sections can be used alone or combined with others freely.

B.2 Training details

Table 4 summarizes the hyperparameters used for each of the three training settings considered in our paper – all hyperparameters were decided by performing a grid search using 5 random models from our search space (in case of distillation and fine-tuning the same model was used as both teacher and student). All training has been done using PyTorch-based code on a single GPU (one of NVIDIA 1080Ti, 2080Ti, V100, P40, P100, and A40). We train each architecture with the same strategy. Specifically, we train each architecture via SGD [34] or Adam [35], using the cross-entropy (CE) loss, mean squared error (MSE) or knowledge distillation (KD) [36] for 200 epochs in total. We set the weight decay as 0.0005 and decay the learning rate from 0.01/0.005/5e-5 to 0 with cosine annealing [37]. For data augmentation, we use random horizontal flip with the probability of 0.5, random crop of 32×32 patch with 4 pixels padding on each border, cutout with 1.0 probability to cut one 16x16 patch out of each image. We also use the RandAug scheme implemented by [38], 2 augmentation transformations are applied sequentially with the magnitude of 14.

C Implementation details

C.1 Blox search space

Operations. In the main paper (Section 2), we presented several convolutional operations with residuals that are based on those in common networks [26, 25, 3, 5]. Here we gives more details on each of them.

- Standard convolution (Conv) block - 3x3 convolution operation is repeated for 4 times.
- Bottleneck convolution (BConv) block - This block consists of 6 repeated operations in the form of convolution -> bottleneck -> convolution. The first convolution is pointwise (1x1 convolution) with N_i input channels and N_i/b_i output channels, where $\{b_0, b_1, b_2, b_3, b_4, b_5\}$ = $\{2, 1, 0.5, 2, 1, 0.5\}$. Then a bottleneck operation (5x5 depthwise-separable convolution) is performed. Lastly, another pointwise convolution brings the representation back to N_i output channels. Residual connection is applied to add input to the output.
- Inverted residual convolution (MBConv) - This block consists of 2 repeated operations that follow a convolution -> squeeze-and-excitation -> convolution structure. N_i input channels are first widen with an expansion ratio r via 3x3 convolution, followed by a squeeze-and-excitation operation [27]. Then a pointwise convolution reduces the number of output channels back to N_i. Finally residual connection is applied to add input to the output.

We control the repetition factor of each block to roughly balance FLOPs and parameters across the different blocks. This is in order to avoid a situation when a certain operations is naturally a better choice not because of its structure and/or ability to efficiently use its parameters, but because it's significantly larger/smaller than other choices. For example, compared to the standard convolution block, an inverted residual convolution contains significantly more parameters due to its expansion ratio – to compensate for that, we repeat it fewer time. Analogously, the bottleneck convolution block was originally proposed as a lightweight alternative to standard convolutions, therefore it naturally has significantly fewer parameters and FLOPS and we can afford to have more of them stacked together. Although we tweak repetitions and other parameters (e.g., kernel size, expansion ratio, etc.) to minimize the differences in high-level metrics, please note that in general the differences are still there, just less significant than they could have been without this balancing.

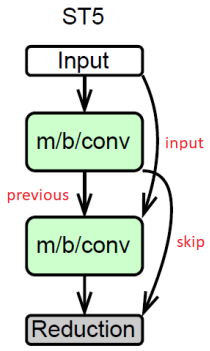

Figure 14: ST5 block architecture is an example with all *previous*, *input* and *skip* connections enabled.

Architectures. There are 6 types of block architectures as shown in the main paper (Figure 3). Each block has two cells (except ST1 which has only one cell), each of which consists of 3 possible operations. If we identify isomorphic cells in ST6 as well, there are 45 unique blocks, which account to 45^3 = 91,125 total unique models in the search space.

In Blox, we encode each model architecture by a 15-dimensional architecture vector. An architecture vector is formatted as [S_0, S_1, S_2], which describes the 3 stages of searchable blocks. Stage S_i is described as [[O_0], [O_1, *previous*, *input*, *skip*]], where O_0 and O_1 refer to the operations of the first and second cell, respectively. *previous* and *input* are binary values that indicate if the cell has connections with the previous cell and the input of the block. *skip* is another binary value that indicates if there is any skip connection in the block. Since O_0 only has one fixed connection to the input, no connection needs to be specified. Figure 14 is an example of block architecture which has all connections enabled.

C.2 NAS algorithms

Below we give the details of the conventional NAS algorithms used in the paper – all algorithms operate on discreet architecture encodings as defined in Section C.1. We did not perform any hyperparameter tuning for any of the algorithms - all our choices come from either the original papers that proposed relevant methods, or follow common practice that can be found in many NAS works.

All algorithms are optimizing for the best validation accuracy achieved by a model throughout its training – if an algorithm alters the number of training epochs, this is the best validation accuracy for the reduced training. When reporting test accuracy we assume each architecture has been trained fully. Since our encoding allows for constructing invalid models (input and output of a single cell might be disconnected if $previous = input = skip = False$), we reward such models with a constant value of -1, these models do not contribute to the cost of running a search.

Random search. We samples architectures from a uniform distribution over all possible architecture encodings.

Q-Learning. We train a 4-layer MLP with 128 neurons in each layer to predict the total return of a two-step alternation process to a model's structure. Given an initial architecture encoding (state), the agent can take an action that changes any single component of the encoding to a different value. After two consecutive actions, the accuracy of the resulting model is considered to be the final reward and the resulting trajectory of 3 models is added to the training set of the agent. We keep up to 32 last trajectories to enable history replay [39], out of which we sample 8 to construct a single training batch for the agent. Additionally, when deciding on an action to take, we always reject choices that would result in the model's structure reverting to any of the previous configurations within the same trajectory – in other words, we reject any trajectories that would contain a cycle, e.g., $A \rightarrow B \rightarrow A$. We use ϵ-greedy strategy with $\epsilon = 0.1$, and discounting with $\gamma = 0.1$. Adam [35] optimizer is used to train the MLP, with learning rate set to 3.5×10^{-4}, weight decay set to 0, betas set to $(0.9, 0.999)$ and epsilon set to 1.0×10^{-8}. The reward for each valid model (validation accuracy) is normalized linearly to the range $[0, 1]$ using 0 and 100 as min and max, respectively.

REINFORCE. We use a single cell LSTM controller trained with REINFORCE, following the setup in [32]. Specifically, the LSTM has 100 hidden and input features and is used auto-regressively to produce categorical distributions for each dimension of the architectural encoding. The distributions is constructed using tanh activation with temperature $T = 5$ and a constant scaling factor of 2.5. The initial input to the LSTM cell is all zeros. The produced distributions are then sampled to decide on a model to train and the controller is then updated by using REINFORCE [40] algorithm with Adam [35] optimizer. The optimizer's setting follows those described in the paragraph about our Q-Learning algorithm, we also use entropy-based regularization with weight 0.0001. Similar to Q-Learning, rewards for valid models are scaled linearly to the range $[0, 1]$ using 0 and 100 as min and max, respectively.

Hyperband. We use our implementation of Hyperband which follows description from the original paper [30]. We use $R = 100$ and $\eta = 3$. The first value represents training budget in percentages, where 100 corresponds to full training (200 epochs) and lower values scale the number of training epochs accordingly. The choice of η follows the recommendation in [30]. The hyperparameters exposed to the algorithm are the values constituting architecture encoding explained in C.1; we use uniform sampling to get random points.

Regularized evolution. We run regularized evolution [7] with pool size 64 and sample size 16.

BRP-NAS. We follow the implementation of BRP-NAS [29] and use a 4-layer binary GCN predictor with 600 hidden units in each layer. We use the official code available at `https://github.com/SamsungLabs/eagle` to perform training. Specifically, we use the following hyperparameters: AdamW optimizer with LR set to 3.5×10^{-4} and weight decay set to 5.0×10^{-4}, dropout rate of 0.2, LR patience of 10, early stopping patience of 35, 250 training epochs, batch size of 64, α of 0.5. We keep training the predictor every 20 models up to 200. The input to the predictor is a graph representation of a model analogous to the ones shown in Figure 22 and 23, after operations are encoded as one-hot vectors and a global node is added, as explained in [29]. The same operations at different stages of a network are assigned different labels.

C.3 Latency measurements

We run each model in Blox on the follow devices (more devices will be added in the future). (i) Desktop platform - NVIDIA GTX 1080 Ti, (ii) Mobile platform - Qualcomm Snapdragon 888 with Hexagon 780 DSP.

We run each model 1000 times on each aforementioned device using a patch size of 32 x 32 and a batch size of 1 for mobile devices and 256 for desktop GPU. For mobile devices, each model is

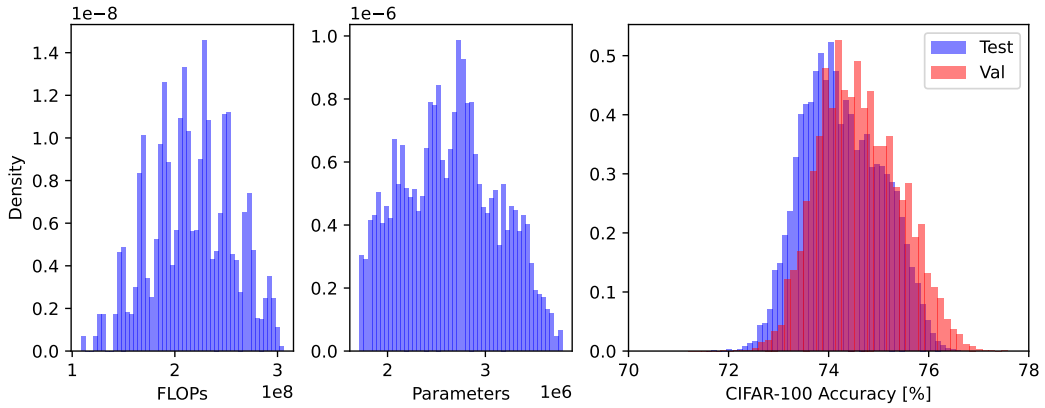

Figure 15: Distribution of FLOPs, number of parameters and accuracy in Blox.

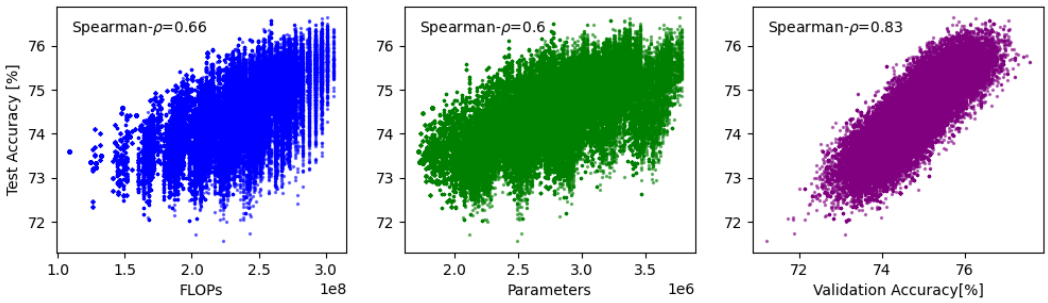

Figure 16: (Left) Correlation between top1 test accuracy and FLOPS; (Middle) Correlation between top1 test accuracy and number of parameters; (Right) Correlation between top1 validation and test accuracy.

quantized to 8 bits and run 10 time with the same settings using tools provided by Snapdragon Neural Processing Engine. For desktop GPU, we use PyTorch and run each model after it is optimized and compiled using Torchscript. In order to lessen the impact of any startup or cool-down effects such as the creation and loading of inputs into buffer, we discard latencies that fall outside the lower and higher quartile values before taking the average of every 10 runs. These averages are discarded again with the aforementioned thresholds before a final average is taken.

D Additional experimental results

D.1 Statistics of the Blox benchmark

FLOPs, number of parameters and accuracy. Figure 15 shows the distribution of FLOPs, number of paramters, as well as top1 validation and test accuracy in Blox. Regarding the relationship among these metrics, Figure 16 (left, middle) suggests that neither FLOPs nor the number of parameters are strong factors to determine the accuracy of models. Furthermore, the result in Figure 16 (right) shows a strong correlation between top1 validation accuracy and top1 test accuracy for all models.

Block architectures. Figure 22 and Figure 23 (at the end of this Appendix) show the models that achieve the best top1 test accuracy and the worst top1 test accuracy, respectively. We observe that the architectures of the best models often consist of MBConv blocks whereas those of the worst models consist of BConv blocks.

D.2 More questions about blockwise NAS

In the main paper, we asked questions related to blockwise NAS and performed analysis using the Blox benchmark. Here we ask a couple more questions to further understand the impact of predictor and distillation strategy.

Q7: Does a poor predictor performance imply poor NAS results? Correlation with accuracy is often used in NAS work as a representative proxy for the algorithm's final performance, and not without a good reason. However, in general it is possible that a NAS algorithm might still produce good results despite being poorly correlated, as highlighted in prior work [29]. This is why we measure the performance of DONNA and LANA before drawing conclusions about their efficacy. Figure 17 compares blockwise methods in the Blox search space and the models are fine-tuned by a good teacher for 10, 50, or 200 epochs. We use *FT*α to denote fine-tuning for α number of epochs. One unit of training cost is equal to the time to train for 200 epochs. The search curves starts at $x = 9$ because of the cost of block distillation (60 blocks $\times$ 3 stages $\times$ 10 epochs / 200). Notably, the number of models trained by different approaches at a common x depends on the number of fine-tuning epochs – e.q., 90, 360 and 1,800 models are evaluated for the FT200, FT50 and FT10 settings, respectively, when the cost is 100. Figure 17 shows that, with enough fine-tuning, good accuracies can be achieved with blockwise methods on the Blox dataset. In the main paper, Section 3.2 already suggested that **the type of signature used when guiding a search has a secondary role and, in general, NAS outcome is not strongly affected by it.** Specifically, we can see that **the number of fine-tuning epochs has much more profound effect than differences in searching algorithms.**

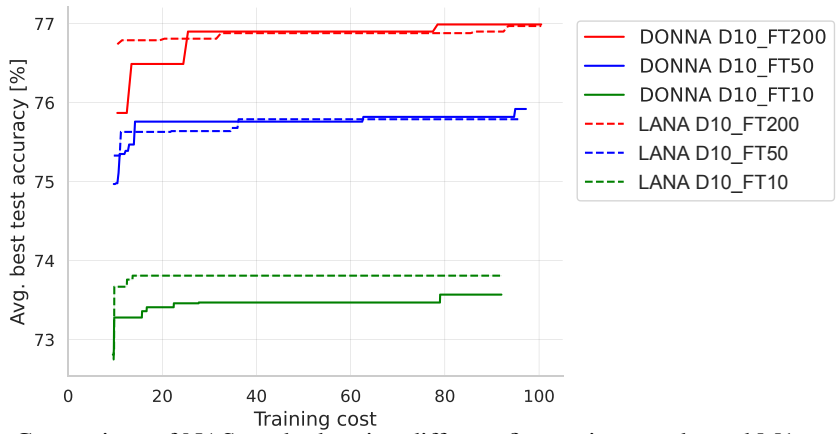

Figure 17: Comparison of NAS methods using different fine tuning epochs and M1 as the teacher. Dα indicates distilling for α epochs. Similarly, FTα indicates fine-tuning the model for α epochs.

Q8: Do we have to distill for 10 epochs? Figure 18 compares blockwise methods in which the blocks are distilled from a good teacher for 1 or 10 epochs and fine-tuned for 200 epochs. Our results show that distillation for only 1 epoch leads to worse predictors, as indicated by the bad starting point in DONNA and slow progress in LANA. On the other hand, distilling for 10 epochs leads to more efficient search. This align with the results in Figure 10 which shows that the predictor performance improves with the number of block distillation epochs. Note that in the original DONNA paper, only 1 epoch was successfully used in blockwise distillation. We anticipate that this works as the original paper uses ImageNet where an epoch is significantly larger than CIFAR-100. This suggests that **the number of distillation epochs should be tuned based on the dataset.**

D.3 Iterative fine-tuning

As we can see from the main paper (the blue curve in Figure 13), the iterative approach has significantly improved the model accuracy without knowing a good teacher in advance. Table 5 further shows how the selected models improve over iterations. After the search completes (**step 1** in Question 6 of the main paper), the top-5 models are selected, as indicated by m1 to m5, and the accuracies are shown in the first row of the table. Then the models are trained from scratch using the normal settings (**step 2**), and the results are shown in the second row. The best model (m1 in this example) is used as the teacher to fine-tune model m1 to m5 (**step 3**), with the results in the third row.

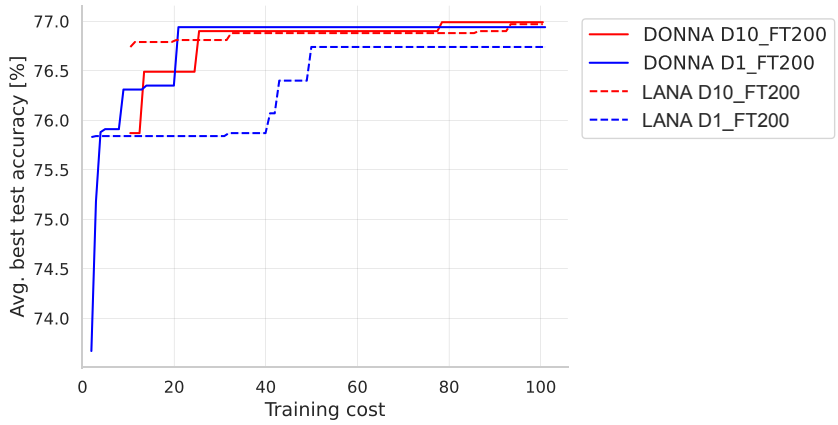

Figure 18: Comparison of NAS methods using different distillation epochs. M1 is used as the teacher.

Lastly, m4 becomes the teacher to fine-tune the models (**step 4**), the results in the forth row show that m4 achieves the best accuracy among m1 to m5.

Table 5: Results of iterative fine-tuning.

	m1	m2	m3	m4	m5
Searched model	70.58	70.50	70.52	70.61	**70.67**
Train from scratch	**74.89**	74.58	73.28	73.89	73.50
Fine tuning iter. 1	76.35	76.27	75.63	**76.36**	75.91
Fine tuning iter. 2	75.61	75.44	75.22	**76.67**	75.82

D.4 Comparison of different NAS methods

With fine-tuning. Figure 19 compares conventional and blockwise methods in the Blox search space. We also add BRP-NAS, regularized evolution, random search and DONNA-GCN (which replaces the linear regression model by a GCN) with the distillation and fine-tuning training methodology. We can see that all NAS methods converge to a high accuracy region when fine-tuned using a good teacher, though DONNA is still marginally the best one. Crucially, Figure 19 highlights that fine-tuning plays a bigger role in the final searched model accuracy than does the student model architecture itself. This begs the question of whether we should focus on searching for a good teacher model architecture then use it, through distillation, to train different student models that are tailored for different target platforms.

Without fine-tuning. In order to investigate the effect of fine-tuning on blockwise NAS, we run DONNA and LANA without applying any fine-tuning methodologies. Specifically, during search, the models are trained in normal setting that blocks are not initialized with pre-trained blocks and no knowledge is distilled from a teacher. Only block signatures (obtained by 10 epochs of distillation) are used to in the predictors to guide the search algorithms. Figure 20 shows that DONNA without fine-tuning still exhibit comparable results with conventional NAS approaches, whereas LANA struggles to show promising progress. The results align with our previous finding that the block signature of LANA is relatively bad in predicting model performance.

D.5 Comparison with other NAS benchmarks

Figure 21 plots Blox (macro search space, 91125 models) with NAS-Bench-201 (cell-based search space, 15625 models) as both have been trained on CIFAR-100. We also include NATS-Bench-SSS (32768 models) which is based on NAS-Bench-201 but scales the architecture size rather than topology. This figure compares Blox to the other search spaces with the same order of magnitude in terms of the number of architectures, parameters and FLOPs.

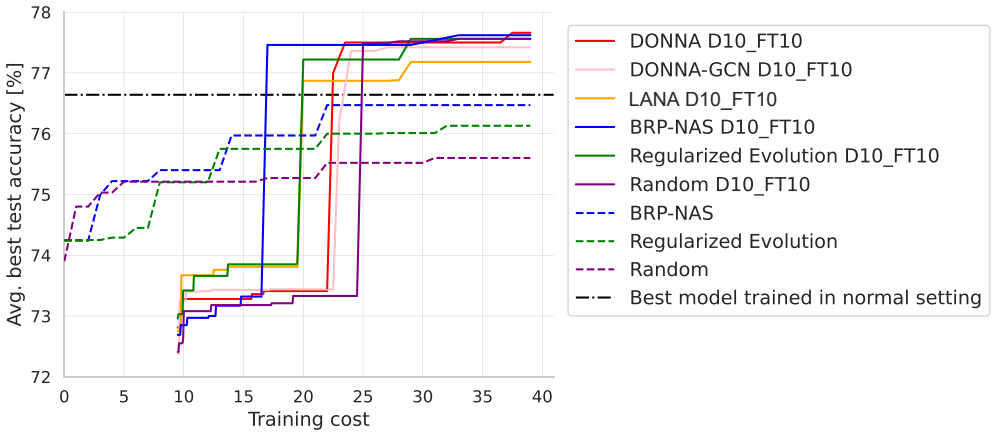

Figure 19: Comparison of conventional and blockwise methods in the Blox search space. For the blockwise methods, whenever the accuracy plateaus for 6 cost units, we switch to full retraining (ranking the models searched and fine-tuning them for 200 epochs using M1 as teacher) and hence the accuracy is boosted.

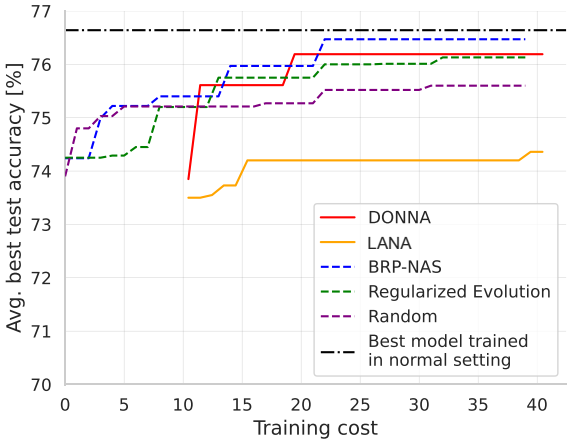

Figure 20: Comparison of blockwise methods in the Blox search space.

E Discussion and limitations

E.1 Choice of hyperparameters.

When training our models, we decided on hyperparameters by taking 5 random models and performing a grid search over different augmentations, learning rates, optimizers, and LR schedulers. We then picked the configuration that yielded the best results and kept using it throughout our work to provide comparable setting. Although in our sample of 5 models we did not observe significant changes in the the relative ranking of the models when different hyperparameters were used (the only changes were happening if the models were already very close in performance), in general it is known that different models might prefer different hyperparameters, thus making NAS results dependant on the training scheme used. Ideally, each model would be trained with its own optimal hyperparameters. However, this would be computationally infeasible and hence we resort to the setting that achieves the best average performance on the small sample of models, as described above.

In the case of distillation, we used the same principle by selecting a configuration giving the best average performance when the same 5 models are used in a self-distillation-like manner (i.e., the same architecture is used as a teacher and a student). In the case of blockwise distillation, the metric that was used to score each training scheme was the average loss of distilled blocks (3 for each model).

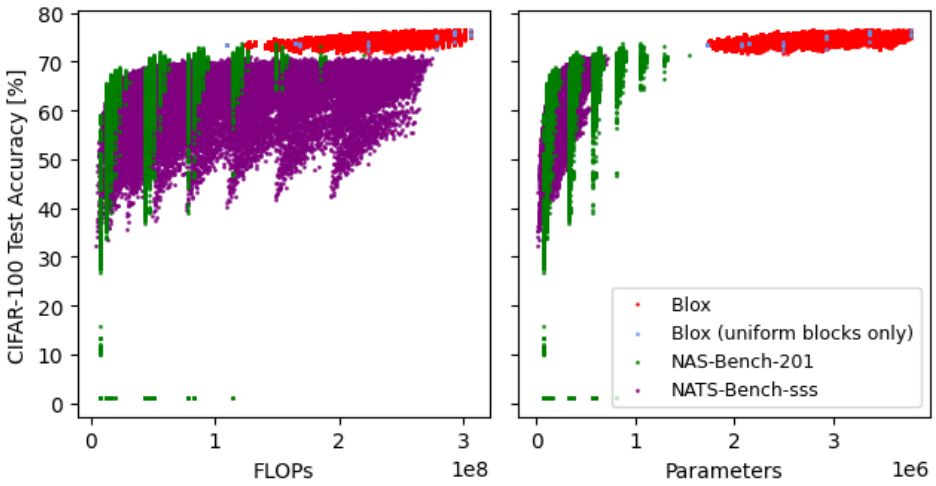

Figure 21: Comparison of Blox models with other NAS benchmarks.

In any other case, validation accuracy was used. The resulting hyperparameters are summarized in Table 4.

E.2 Multi-objective NAS.

Both DONNA and LANA are originally designed to perform multi-objective NAS where accuracy should be balanced out with latency of different models. However, we focus our analysis on the simpler case where maximizing accuracy is the sole objective. Because of that, we have to make necessary changes in both algorithms – for DONNA, the sub-component responsible for running NSGA-II is configured to behave like a standard single-objective evolutionary algorithm. For LANA, budget and cost of each block ware both set to values that guarantee that the constraint in the ILP problem can never be violated. We consider both those changes to be sensible in the context of our work.

E.3 Potential societal impact

We propose a new benchmark for NAS, aiming to make NAS faster and more accessible for researchers to run generalizable and reproducible NAS experiments. The use of tabular NAS benchmarks allow researchers to vastly reduce the carbon footprint of traditionally compute-expensive NAS methods. Our work can facilitates NAS research that may be have positive societal impacts (e.g., avoid training excessive amount of models) or negative societal impacts (e.g., lead to models that have fairness and security concerns).

E.4 Discussion on previous reviews

Definition of macro search space. Previous reviewers have asked about the differences between the macro search space and cell-based search space. We clarify the fundamental differences here –

- Macro search space – each stage of a model is allowed to have different block architecture. In Blox, there are 45 unique blocks per stage, so the size of the search space is $45^3 = 91,125$.
- Previous cell-based work such as NAS-Bench-101 [13], NAS-Bench-201 [14] and DARTS [8] – the same cell/block is repeated to form a model. If we followed such cell-based setting, the size of the search space would be 45 only.

Motivations. Regrading the motivation of creating this benchmark dataset, we emphasize that previous literature has mostly focused on cell-based designs. NAS algorithm can only search for operations and connections of a cell that is repeatedly stacked within a predefined skeleton. As shown in Figure 4 of the main paper, a macro search space enables layer diversity, and contains higher performing models than a cell-based search space. Although macro search space is promising,

the macro search space size grows exponentially with the number of blocks, posing a challenge to existing search algorithms. Blockwise search algorithms are emerging (particulary DONNA and DNA that are studied in our paper), however, different methods are not comparable to each other due to different training procedures and search spaces. As a result, we present the first large-scale benchmark on macro search space, which enables efficient ways to study NAS in this challenging setting.

Findings. To address comments about the findings of the benchmark, we asked a series of questions in order to isolate relevant behaviour of the studied algorithms – 1) *fine-tuning* (improvement brought by fine-tuning, the impact of teacher, correlation between fine-tuned models and conventionally trained models); 2) *predictor* (How do block signature and end-to-end predictor affect the performance of blockwise NAS methods?); 3) *search efficiency* (Can we fine-tune end-to-end model and distill blocks with few epochs? Can we fine-tune with bad teacher?).

The answers lead to a consistent conclusion – performance is improved on blockwise methods over conventional algorithms. In summary, 1) when a good teacher is used, blockwise NAS achieves better results (i.e. more accurate model and lower search cost) than conventional NAS ; 2) when a bad teacher is used, we proposed a simple iterative strategy which allows us to again dominate conventional NAS. 3) efficiency of blockwise NAS is improved by utilizing reduced fine-tuning proxy followed by full fine-tuning, which is our contribution stemming from Section 3.

F Dataset documentation

Here we answer the questions outlined in the datasheets for datasets paper [41].

F.1 Motivation

For what purpose was the dataset created? Standardized NAS benchmarks have been created to facilitate a fair comparison of NAS algorithms. Macro NAS, which enables the individual search for each block in a DNN, has been a promising alternative to conventional cell-based NAS. However, macro NAS is exorbitantly expensive because the search space size grows exponentially with the number of blocks. We present Blox as a large-scale benchmark to enable the empirical analysis of NAS algorithms on macro search space and to shed some light on how to perform efficient NAS in this challenging setting.

Who created the dataset (e.g., which team, research group) and on behalf of which entity (e.g., company, institution, organization)? The Automated Machine Learning Group in Samsung AI Center Cambridge created the dataset.

Who funded the creation of the dataset? Samsung AI Center Cambridge funded the creation of the dataset.

F.2 Composition

What do the instances that comprise the dataset represent (e.g., documents, photos, people, countries)? Each instance in the dataset represents a network in the Blox search space, and the corresponding accuracies, training time and environment, number of FLOPS and parameters, and inference latencies on CIFAR-100 dataset.

How many instances are there in total (of each type, if appropriate)? The dataset consists of 91,125 instances.

Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set? The dataset contains all possible instances defined in the Blox search space.

What data does each instance consist of? Each instance consists of the accuracies, training time and environment, number of FLOPS and parameters, and inference latencies on CIFAR-100 dataset. Specifically, an instance in the base dataset contains top1 validation accuracy for all epochs and the final test top1 accuracy, as well as the training time for each model. An instance in the extended dataset includes loss (training, validation, test), top1 training accuracy for all epochs, and top5

training accuracy for all epochs. An instance also includes FLOPs, number of parameters, latency and architecture vector, training environment of the model.

Is there a label or target associated with each instance? Each instance is associated with accuracies, training time, number of FLOPS and parameters, and inference latencies.

Is any information missing from individual instances? No.

Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)? Not applicable. Each instance stands on its own.

Are there recommended data splits (e.g., training, development/validation, testing)? No. It depends on the use case, for instance specific settings of the NAS alogrithms or performance predictors.

Are there any errors, sources of noise, or redundancies in the dataset? No.

Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)? The dataset is self-contained and does not rely on other datasets. The dataset is trained on CIFAR-100, however, user can re-train the models in the Blox search space on different datasets.

Does the dataset contain data that might be considered confidential (e.g., data that is protected by legal privilege or by doctor-patient confidentiality, data that includes the content of individuals' non-public communications)? No.

Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety? No.

F.3 Collection process

How was the data associated with each instance acquired? We train all instances in the Blox search space on CIFAR-100 dataset. Training and validation metrics (loss and accuracy) are logged at each epoch. Test metrics and training time are logged at the end of training for the best model. We also save static information consisting of FLOPs, the number of parameters, latency and architecture vector. In addition, information about training environment are included for reproducibility. That includes random seed, GPU used, versions of the OS, codebase, driver, etc.

What mechanisms or procedures were used to collect the data (e.g., hardware apparatus or sensor, manual human curation, software program, software API)? The data are collected by training all the models in the Blox search space using the source code provided in the repository.

If the dataset is a sample from a larger set, what was the sampling strategy (e.g., deterministic, probabilistic with specific sampling probabilities)? Not applicable.

Who was involved in the data collection process (e.g., students, crowdworkers, contractors) and how were they compensated (e.g., how much were crowdworkers paid)? The data are collected automatically by the script provided in the repository.

Over what timeframe was the data collected? The initial version of data was collected between October 2021 and June 2022.

Were any ethical review processes conducted (e.g., by an institutional review board)? No.

F.4 Preprocessing

Was any preprocessing/cleaning/labeling of the data done (e.g., discretization or bucketing, tokenization, part-of-speech tagging, SIFT feature extraction, removal of instances, processing of missing values)? The weights of the trained models are not included due to limitation of data storage.

Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data (e.g., to support unanticipated future uses)? No.

Is the software used to preprocess/clean/label the instances available? Yes, the software is available in the repository.

F.5 Uses

Has the dataset been used for any tasks already? The dataset has been used in this paper to evaluate the performance of different NAS algorithms.

Is there a repository that links to any or all papers or systems that use the dataset? Yes, it is listed in the repository.

What (other) tasks could the dataset be used for? We believe that this dataset will allow researchers to evaluate the performance of different NAS algorithms, particularly blockwise methods. In addition, the latency measurements in the dataset will enable NAS targeting different hardware devices.

Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses? No.

Are there tasks for which the dataset should not be used? No.

F.6 Distribution

Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created? No.

How will the dataset be distributed (e.g., tarball on website, API, GitHub)? The source code is available in the repository (`https://github.com/SamsungLabs/blox`). The repository also includes links to allow the users to download the dataset.

When will the dataset be distributed? The dataset is distributed at the same time as the published paper.

Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)? The dataset is distributed under CC BY-NC license.

Have any third parties imposed IP-based or other restrictions on the data associated with the instances? No.

Do any export controls or other regulatory restrictions apply to the dataset or to individual instances? No.

F.7 Maintenance

Who is supporting/hosting/maintaining the dataset? The dataset is supported and maintained by Samsung AI Center Cambridge. We host the dataset on public repository (`https://github.com/SamsungLabs/blox`).

How can the owner/curator/manager of the dataset be contacted (e.g., email address)? The contacts of the owners are included in the paper and the documentation of the public repository.

Is there an erratum? Yes, it will be included in the repository.

Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)? Yes. Updates will be communicated via the repository, and the dataset will be versioned.

If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances (e.g., were individuals in question told that their data would be retained for a fixed period of time and then deleted)? Not applicable.

Will older versions of the dataset continue to be supported/hosted/maintained? Yes.

If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so? Yes. The source code is available to allow others extend/build on/contribute to the dataset.

F.8 Author statement

The authors confirm all responsibility in case of violation of rights and confirm the licence associated with the dataset.

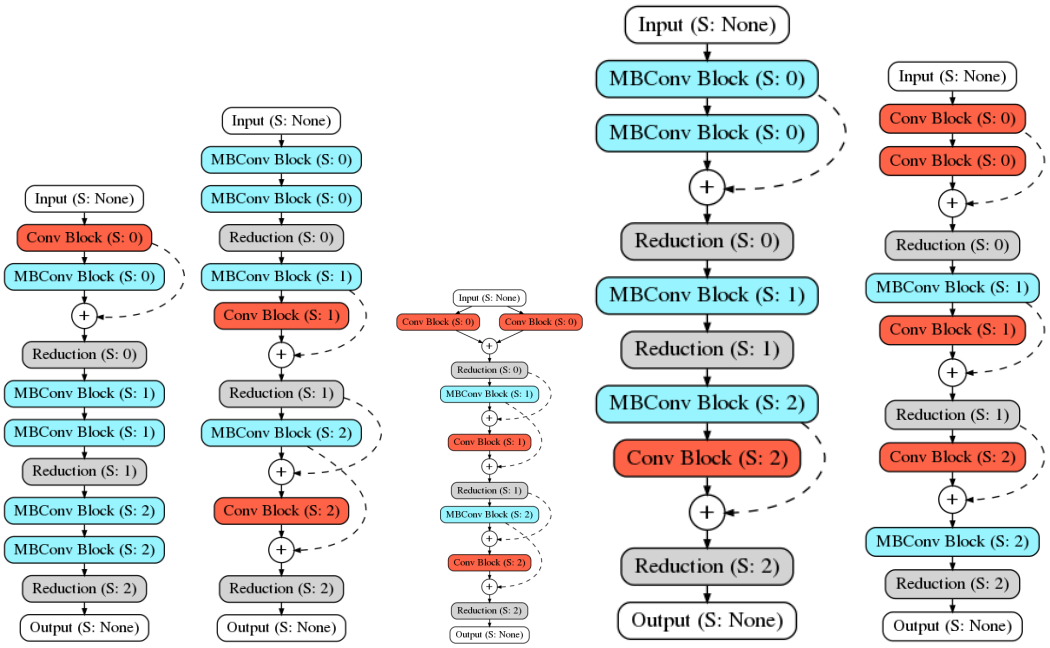

Figure 22: Best 5 models in the Blox search space.

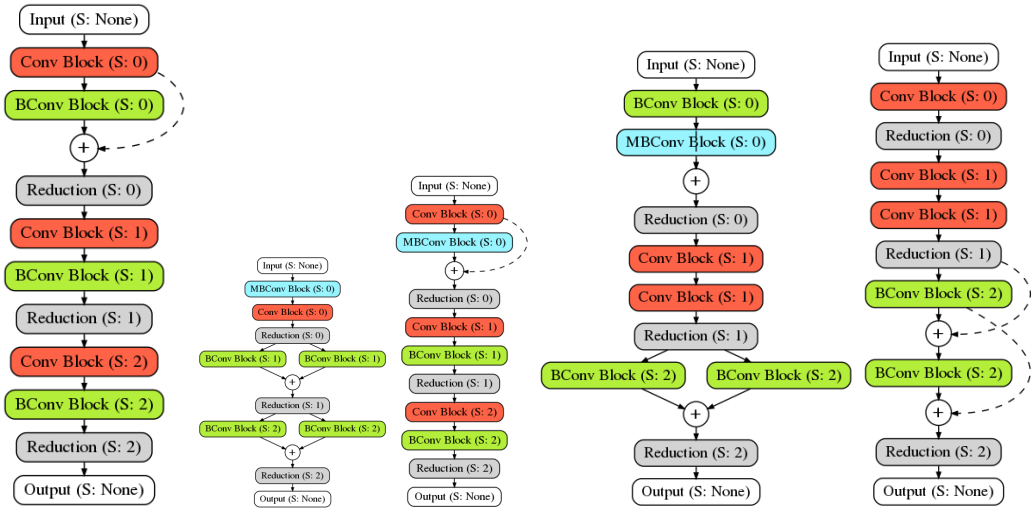

Figure 23: Worst 5 models in the Blox search space.