Cross-Model Operator Batching for Neural Network Architecture Search

Lingling Ye[1], Chi Zhang[1], Mingxia Li[1], Zhenhua Han[2], and Haisheng Tan[1(✉)]

[1] School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
{yeling0,gzhnciha,sa20011036}@mail.ustc.edu.cn, hstan@ustc.edu.cn
[2] Microsoft Research Asia (MSRA), Shanghai, China

Abstract. Recently, automated machine learning (AutoML) and neural architecture search (NAS), regarded as promising techniques to design deep learning (DL) models automatically, have received increasing attention from both industry and academia. NAS will generate a large number of candidate models, which typically consist of numerous common substructures, providing a vast opportunity for cross-model optimization (e.g., operator batching) to improve training efficiency. However, most of the existing AutoML frameworks do not make use of operator batching and we also lack an efficient batching strategy. In this work, we propose a heuristic scheme named `DPBat` to guide the operator batching among multiple models in NAS. For most models, the operator batching of `DPBat` can be finished in just a few seconds, which is negligible compared to the subsequent training. We adopt Microsoft's open source AutoML framework NNI to implement `DPBat` to real NAS scenarios. Extensive experiments show that `DPBat` is highly effective in improving training efficiency and reducing the overhead of operator batching, with a throughput 3.7× higher than the standard practice of running each job without batching.

Keywords: AutoML · NAS · Operator batching

1 Introduction

In recent years, deep learning has achieved great success in various domains, including image classification [7,8], natural language translation [13], and object detection [9]. However, this success has been accompanied by a growing demand for architectural engineering. Most of the complex neural architectures are manually designed (e.g., VGG-16 [10], BERT [4] and GPT-3 [1]), which is time-consuming and requires lots of expertise experience. Therefore, automated machine learning (AutoML) and neural architectures search (NAS) have received more and more attention from both industry and academia. Some institutions have launched their framework that implements the search for neural network architectures, such as Microsoft's NNI, Huawei's Vega, and Amazon's AutoGluon.

L. Wang et al. (Eds.): WASA 2022, LNCS 13472, pp. 231–242, 2022.
https://doi.org/10.1007/978-3-031-19214-2_19

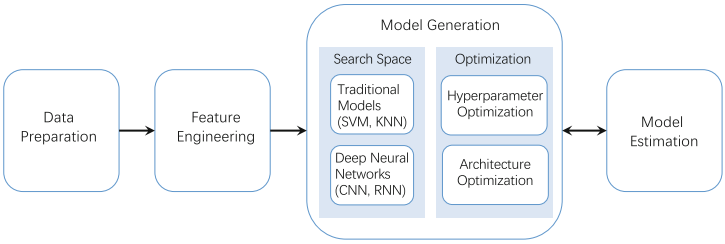


Fig. 1. An overview of AutoML pipeline

A typical AutoML pipeline contains four parts as Fig. 1 shows: data preparation, feature engineering, model generation, and model evaluation. As a key component of AutoML, the search space defines which neural architectures the NAS method can discover in principle. The number of models covered by the search space is enormous, and searching for an optimal model could take up to hundreds of hours [14]. We first investigate how NAS generates models to reduce the training cost and optimize hardware resource usage. The optimization in model generation can be divided into hyperparameter optimization (HPO) and architecture optimization (AO). Models for hyperparameter tuning often have the same types of operators with the same shape. Analogously, models in architecture optimization scenarios tend to have significant similarities as they share a common skeleton [14]. Operators with the same type and parameters can potentially be *batched* together and computed in a single operator kernel, which enables more fine-grained GPU sharing by using less GPU memory to increase SIMD utilization. Therefore, there are huge opportunities for AutoML frameworks to optimize the training of multiple similar models and improve hardware utilization.

Figure 2 illustrates an example that two models share multiple common operators, where Model 1 and Model 2 both have conv 3×3 and ReLU. After batching, the input of the common operators (conv 3×3, ReLu) are fused along the batch dimension, and the outputs are split when operators (BatchNorm2d) vary.

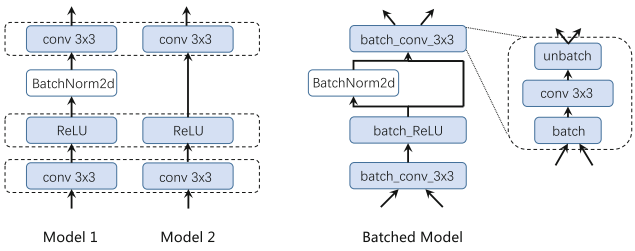


Fig. 2. Operator batching: an example

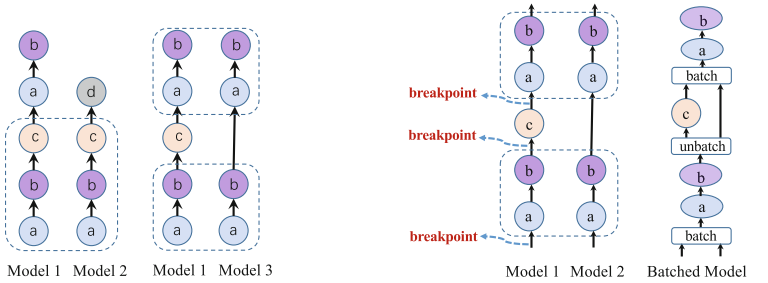
In the literature, there are apparent gaps between the requirement to support this kind of cross-model optimization and the existing operator batching method. [12] came up with the idea of inter-model horizontal fusion, which only deals with HPO scenarios. However, the submitted training tasks generated in NAS are disorder, which [12] would fail to deal with. Besides, the implementation of operators batching in [14] is limited. First, the types of operator batching it supports are limited (Conv2d only). Second, the operator batching strategy in [14] is rudimentary. Specifically, this algorithm uses the idea of breadth-first search (BFS) to compare the operators of each layer between the models until they are different, which means that when the first few layers of the models are different, the strategy's performance will degrade.

To narrow the gaps mentioned above, we propose a scheme to improve the batching efficiency in NAS scenarios. Our objective is to make full use of the similarities among the models and improve training throughput, which is a key performance indicator of training efficiency. Our contributions can be summarized as follows:

- We formulate the DL job clustering and batching problem in NAS scenarios described in Sect. 3. The objective is to maximize the throughput of model training per unit time and help accelerate the process of model generation.
- We propose a novel **D**ynamical **P**rogramming based **Bat**ching strategy, named `DPBat`. `DPBat` includes an efficient cluster algorithm that takes advantage of the similarity among the models generated in NAS. Based on the clustering result, `DPBat` determines an operator batching strategy by comprehensively investigating the performance improvement and overhead.
- We conduct extensive experiments by using Microsoft's open source AutoML framework NNI to evaluate the performance of our algorithm in real NAS scenarios. The experimental results indicate that `DPBat` can significantly improve training efficiency and reduce the overhead of operator batching, achieving up to 3.7× higher throughput than the standard practice of running each job on a separate accelerator.

2 Motivation

Lack of Indicators to Measure Which Models Should be Batched Together. The DNN models generated from the same search space tend to have similarities, and those with the highest similarity should be put together for operator batching. This is not taken into account by the existing batching algorithm due to the lack of indicators that can accurately describe the similarity of models. For example, the maximum common subgraph is not a good indicator. Although a DNN model architecture can be depicted as a data flow graph (DFG), the model similarity is not equivalent to the size of the largest common subgraph. As Fig. 3(a) shows, each model is abstracted into a DFG, where each node represents an operator or a subgraph. Obviously, model 1 and model 2 have the largest common subgraph. But model 1 and model 3 can batch



(a) Maximum common subgraph (b) Overhead in operator batching

Fig. 3. Similarity and overhead in operator batching

more nodes with smaller common subgraphs. So a wiser solution should be to batch model 1 and 3 together instead of model 1 and 2, even though they share a larger common subgraph.

Limitation of the Current Batching Schemes. The idea of inter-model horizontal fusion in [12] only applies to the situation where the models are all the same except for hyperparameters. It can not handle the situation when the architectures of submitted training models get different. Besides, there are limitations in the implementation of operators batching in [14]. The types of operator batching it supports are limited, and there is no efficient algorithm to achieve operator batching between multiple models.

Lack of Consideration for Batching Cost. [12,14] take no consideration of batching cost. Different from models in [12] which have the same architecture, each layer of operators in the model can be batched without breakpoints (the position that generates the batch/unbatch cost). In more general scenarios, operators that can be batched are not continuous. As depicted in Fig. 3(b), the input of common operators needs to be concated along the channel dimension while the output is split at the breakpoint. The operations of concating and splitting bring extra overhead in time and memory. At the same time, batching of different operators brings different performance improvements. The factors mentioned above will affect the choice of operators to be batched.

3 Problem Formulation

3.1 System Model

We consider a system with $\mathcal{D} = \{d_1, ..., d_{|\mathcal{D}|}\}$ computing devices (e.g., GPUs) and a set of training jobs $\mathcal{J} = \{j_1, ..., j_{|\mathcal{J}|}\}$ generated by NAS approaches. Each device d_i has a limited memory d_i^{mem}. The architecture of each job can be depicted as a data-flow graph (DFG) $\mathcal{G}_i(\mathcal{N}_i, \mathcal{E}_i)$. Here, $\mathcal{N}_i$ is the set of nodes belonging to graph $\mathcal{G}_i$, $\mathcal{E}_i$ is the set of directed edges defining the dependence among nodes. A single node in graph $\mathcal{G}_i$ represents an operator (or a sub-graph)

with one or multiple input and output tensors. Each node has its own runtime and memory footprint, denote as n_{ij}^t the execution time of node n_{ij} and n_{ij}^{mem} the memory occupied by n_{ij}. The training time of job j_i in one iteration is $j_i^t = \sum_{n_{ij} \in \mathcal{N}_i} n_{ij}^t$. And the memory occupied in the training process of j_i is $j_i^{\text{mem}} = \sum_{n_{ij} \in \mathcal{N}_i} n_{ij}^{\text{mem}}$.

3.2 Batching

If $\mathcal{J}_K = \{j_1, ..., j_k\}$ is a job set selected for operator batching, assuming that all nodes can be divided into b categories according to their attributes. The nodes in the same category can be batched together. We denote as $N_i = \{n_i^1, ..., n_i^{|N_i|}\}$ the nodes in the ith category. After batching, N_i will be replaced by a new BatchNode B_i. The execution time B_i^t is usually smaller than N_i^t. We denote as $p_i^t = N_i^t - B_i^t$ the benefit after batching N_i. The input of B_i need to be concated along the channel dimension and the output are split when the successor node of B_i is not BatchNode, which brings extra overhead in time. Denote as $\check{\mathcal{J}}_K$ the production of batching $\mathcal{J}_K$. All benefits and costs are $P_K^t = \sum_{i=1}^b p_i^t$ and B_K^t, respectively.

3.3 Problem Definition

Based on the above system model, given the set of computing devices $\mathcal{D}$ and jobs $\mathcal{J}$, the execution time j_i^t and occupied memory j_i^{mem} of each job j_i, as well as the possible overhead in operator batching process, our problem is to select the jobs with the most similar model architecture for operator batching without exceeding the device memory limit. Our goal is to maximize the utilization of devices by maximizing the average throughput of training models. Divide the task set $\mathcal{J}$ into several subsets $\mathcal{S} = \{s_1, ..., s_{|\mathcal{S}|} | \forall i \neq j, s_i \cap s_j = \varnothing, s_1 \cup ... \cup s_{|\mathcal{S}|} = \mathcal{J}\}$ based on their similarity. The training jobs in s_i are $\mathcal{J}_i$. For each set s_i, we need to find a batching strategy that maximize $P_i^t - B_i^t$, making $\check{\mathcal{J}}_i^t$ as small as possible. A smaller $\check{\mathcal{J}}_i^t$ means the training process has higher throughput.

4 Algorithm Design

There are several parts to deal with operator batching between multiple models. The first part is to calculate the similarity of two models (Algorithm 1) and then cluster multiple models based on similarity (Algorithm 2). The next part is the design of batching strategy of clustered models (Algorithm 3).

4.1 Clustering Based on Model's Similarity

Since each model can be represented by a DFG, the similarity between models correlates with the similarity between graphs. The methods of measuring graph similarity include maximum common subgraph [2], graph edit distance [6], graph

Algorithm 1: similarity

1 **Input** *job* j_i, j_k
2 **Output** *similarity of* j_i *and* j_k
3 *Let* H_i *and* H_k *be the hash value lists of the topologically sorted nodes from graph* g_i *and* g_k *,respectively*;
4 $l_{ik} \leftarrow$ length of the longest common subsequence of H_i and H_k;
5 $n_i, n_k \leftarrow$ the number of nodes of j_i and j_k;
6 **return** $\frac{2 \times l_{ik}}{n_i + n_k}$

isomorphism [5], etc. They cannot usually be solved in polynomial time. We made some modifications based on the longest common subsequence (LCS) and calculated the similarity between models by simplifying the graph's structure. We describe the details in Algorithm 1.

Algorithm 1 topologically sorts the nodes of the model's graph and sets the hash value of each node according to its parameters and attributes. Nodes with the same hash value mean they can be batched together. Therefore, we use an ordered list of hash values H_i to approximate the architecture of the original model's graph g_i (Line 3). And refer to the idea of LCS (Line 4), the final result l_{ik} can be used for measuring jobs' similarity (Line 5 to Line 6).

By approximately calculating the similarity of the models by Algorithm 1, we can cluster the job set $\mathcal{J}$. Divide $\mathcal{J}$ into several subsets based on similarity among models. The number of models in each subset depends on the sum of the model memory, which cannot exceed the device memory limit. We describe the details of how to cluster job set $\mathcal{J}$ in Algorithm 2. At the beginning of each round of clustering, select a job j_i that has not been clustered from the job set and remove it from $\mathcal{J}$ (Line 6). Assign j_i to set s (Line 7). When the model memory in s does not exceed the limit, select a job from the unclustered job set $\mathcal{J}$ and the clustered job set s respectively, and their similarity is the highest among all the current jobs (Line 8 to Line 9). Add candidate model to s without exceeding memory constraints and remove it from $\mathcal{J}$ (Line 10 to Line 12). Otherwise, restart the next round of clustering (Line 13 to Line 15).

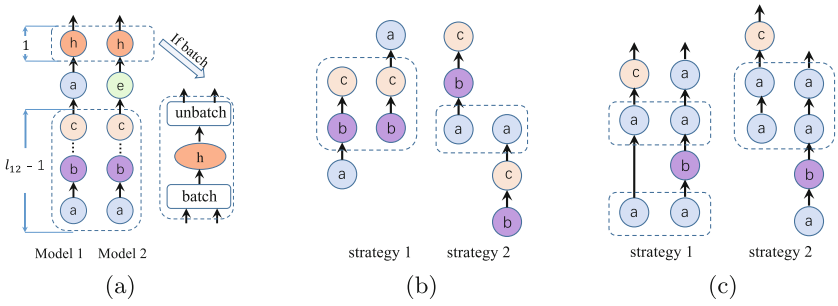


Fig. 4. Possible situations of operator batching

Algorithm 2: clustering

Input *training jobs* $\mathcal{J}$
Output *clustered job set* $\mathcal{S} = \{s_1, s_2, ...\}$
$\mathcal{S} \leftarrow \varnothing$;
while $\mathcal{J} \neq \varnothing$ **do**
  $s \leftarrow \varnothing$;
  randomly select a model j_i *from* $\mathcal{J}$ *and remove it from* $\mathcal{J}$;
  $s \leftarrow s \cup j_i$;
  while $\mathcal{J} \neq \varnothing$ *and* s *does not exceed device memory* **do**
    $j_{\text{cand}} \leftarrow \arg\max_{j_m \in \mathcal{J}} \max_{j_s \in s}$ `similarity`(j_s, j_m);
    if $s \cup j_{cand}$ *won't exceed device memory* **then**
      $s \leftarrow s \cup j_{\text{cand}}$;
      Remove j_{cand} *from* $\mathcal{J}$;
    else
      $\mathcal{S} \leftarrow \mathcal{S} \cup s$;
      break;

4.2 Design of Batching Strategy

We first consider batching strategy design for two models and then extend it to multiple models. For a pair of similar models j_1 and j_2, we can get l_{12} (the maximum number of operators can be batched between model j_1 and j_2) by LCS. However, greedy batching of all l_{12} operators maybe not be guaranteed to bring the most benefits. Figure 4(a) shows one possible scenario. Because the length of the operator sequence that can be continuously batched is too short, the benefit of batching operator h may be less than the additional cost of integrating and splitting tensors, leading to negative returns. In this case, the strategy that batching the maximum number of operators is suboptimal.

Besides, the benefits of operator batching are also related to the type of operators. As Fig. 4(b) shows, there are two batch strategies. Although strategy 2 batches fewer operators, it may yield greater benefits than strategy 1. The number of breakpoints also affects the training time of the batched model. As Fig. 4(c) shows, there are two types of fusion strategies that batch the same type and number of operators. It can be concluded that strategy 2 is better than strategy 1 because of fewer breakpoints and less overhead.

Therefore, in order to maximize the benefits of operator batching and reduce the additional overhead, we propose `DPBat` (Algorithm 3), which takes breakpoints, operator types, etc. into account. We guide the multi-model operator batching through the optimal batching strategy of the two models. The details are described in Algorithm 3. We use a 4-dimensional array to record the net benefit generated during the operator batching process. For $dp[i][j][0..1][0..1]$, the 0 and 1 in the last two dimensions indicate whether the ith and jth elements are in the *batched* state, respectively. When the last two dimensions are 1 simultaneously, it means that the ith and jth elements are identical and can

$$dp[i][j][1][1] = max \begin{cases} dp[i-1][j-1][1][1] + p & \text{continuous batching} \\ dp[i-1][j-1][0][1] + p - batch\ cost & \text{start a new continuous batching} \\ dp[i-1][j-1][1][0] + p - batch\ cost & \text{start a new continuous batching} \\ dp[i-1][j-1][0][0] + p - batch\ cost & \text{start a new continuous batching} \end{cases}$$

$$dp[i][j][1][0] = max \begin{cases} dp[i][j-1][1][1] - unbatch\ cost & \text{end a continuous batching} \\ dp[i][j-1][1][0] & \text{phase without batch} \\ dp[i-1][j][0][0] & \text{phase without batch} \\ dp[i-1][j][1][0] & \text{phase without batch} \end{cases}$$

$$dp[i][j][0][1] = max \begin{cases} dp[i-1][j][1][1] - unbatch\ cost & \text{end a continuous batching} \\ dp[i-1][j][0][1] & \text{phase without batch} \\ dp[i][j-1][0][0] & \text{phase without batch} \\ dp[i][j-1][0][1] & \text{phase without batch} \end{cases}$$

$$dp[i][j][0][0] = max \begin{cases} dp[i-1][j][0..1][0] & \text{phase without batch} \\ dp[i][j-1][0][0..1] & \text{phase without batch} \end{cases}$$

Fig. 5. Transition equation

be batched. Other values indicate that the ith and jth elements are different and can not be batched. In addition to adding the benefit p of batching, dp also needs to subtract the corresponding batch/unbatch cost at the breakpoint. The specific transition equation is shown in Fig. 5. For a job set s_i, which includes multiple models with similar architectures. We select a model from s_i and $\check{s}_i$ respectively. Their similarity is the highest among all current model pairs. $\check{s}_i$ stores those models that have been batched (Line 6 to Line 8). Using the transition equation in Fig. 5 to calculate the maximum net benefit of batching two models. The batching strategy λ of two models corresponding to the maximum value in dp is optimal. We incorporate the strategy λ obtained at each round into the final result λ^* until job set s_i becomes empty. (Line 9 to Line 11).

Algorithm 3: DPBat

1 **Input** *similar job set* $s_i = \{j_{i1}, j_{i2}, ...\}$
2 **Output** *batching strategy* λ^*
3 $\check{s}_i \leftarrow \{j_{i1}\}$ and remove j_{i1} from s_i;
4 $\lambda^* \leftarrow \varnothing$;
5 **while** $s_i \neq \varnothing$ **do**
6 *Select a pair of jobs* j_1, j_2 *with the highest similarity,* $j_1 \in s_i$, $j_2 \in \check{s}_i$. *Let* H_1, H_2 *be the hash value lists of their topologically sorted nodes*;
7 $\check{s}_i \leftarrow \check{s}_i \cup j_1$;
8 Remove j_1 from s_i;
9 *Calculate the net benefit brought by different operator batching strategies using equation in figure 5* ;
10 *Let* λ *be the batching strategy corresponding to the maximum value in* dp;
11 $\lambda^* \leftarrow \lambda^* \cup \lambda$

5 Evaluation

In this section, we evaluate the performance of DPBat in real NAS scenarios and compare it with three baselines. Overall, the key findings include: DPBat significantly improves training efficiency and reduces the overhead of operator batching. DPBat achieves up to 3.7× higher training throughput than running each job serially, which is a common practice employed by the AutoML framework.

5.1 Experiment Settings

To evaluate DPBat in real scenarios, we used Microsoft's NAS tool NNI which can separate the cross model optimization from model generation. We follow the same configurations as Retiarii [14], select representative NAS solutions MnasNet [11], MobileNetV2-based model space and reinforcement learning exploration strategy. In the experiment, the NAS approach will generate 1000 models in 10 batches(100 models each batch). DPBat and the other baselines are given the same set of models in the same order for a fair comparison. These models use the same batch size, which is 8 images (ImageNet's training images [3]) per mini-batch. We implemented the experiments on 4 NVIDIA Tesla P100 GPUs of 16GB GPU memory. The performance is measured by averaging the throughput over 1000 mini-batches.

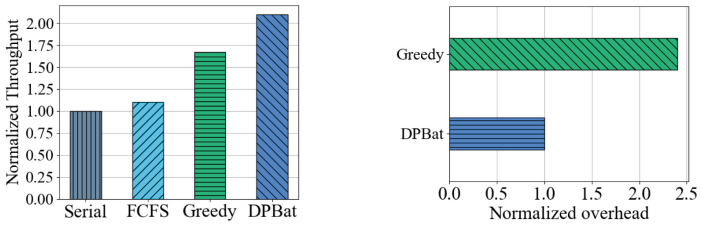
5.2 Three Baselines

We compare DPBat with the following three baselines.

- **Serial**: each training job is executed on a single accelerator, which is employed by most DL frameworks [14].
- **FCFS**: *FCFS* is the policy used by NNI's cross-graph optimization engine and it clusters the jobs by order of arrival rather than similarity. Training jobs arrive in batches of 100 models, sequentially dividing the task set $\mathcal{J}$ into several subsets. Each subset contains the maximum number of models before the GPU runs out of memory. For example, training jobs $\{j_1, j_2, ..., j_i\}$ are divided into subset s_1, $\{j_{i+1}, j_{i+2}, ..., j_k\}$ are divided into s_2 and so on. The design of the operator batching strategy is also extended from two models to multiple models. For a pair of models, use the idea of BFS to compare the DFG of the two models layer by layer and stop batching when the layer depth is the same but the layer nodes are different.
- **Greedy**: *Greedy* is the policy described in Retiarii [14] which fuses all common operators. It does not consider batch/unbatch cost and different benefits of batching different kinds of operators, which means setting the batch/unbatch cost and benefit in DPBat to 0.

5.3 Experiment Results

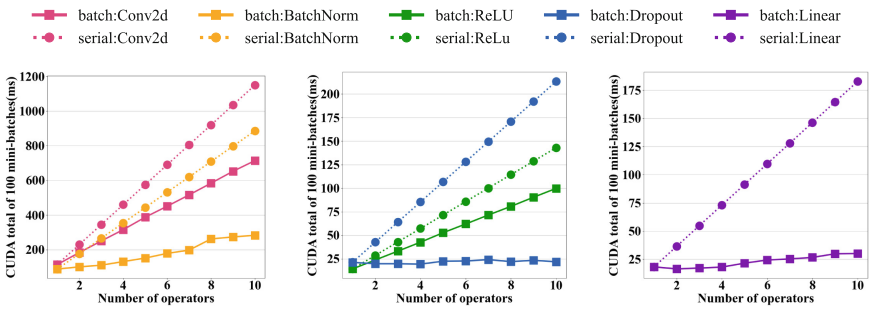
In this part, we present the experimental results on 1000 models and dissect the source of improvement brought by DPBat. In all cases, our algorithm DPBat outperforms the other baseline.



(a) Training throughput (b) Batching overhead

Fig. 6. Performance of different algorithms

The Overall Performance. Figure 6(a) illustrates the four algorithms' average throughput of 1000 models. `DPBat` achieves higher throughput than all baselines, 2.1× (up to 4.7×) over **Serial**, 1.92× over **FCFS**, 1.25× over **Greedy**. **FCFS** cannot make full use of the similarity between models because of the lack of clustering. Moreover, its batching strategy cannot select all the operators that can be batched either. **Greedy** focuses on the number of operators that can be batched. While maximizing the number of batched operators, the additional batch/unbatch overhead increases. It also ignores the fact that the benefit of batched operators is related to the operator's type. Figure 6(b) shows the average batch/unbatch cost of 1000 models. Because taking breakpoints into account, `DPBat` can significantly reduce additional overhead compared to **Greedy**.



(b) (c) (d)

Fig. 7. Performance of different operators

Sources of Improvements. To understand why `DPBat` achieves better performance than the other baselines, we perform a deeper analysis using the PyTorch profiler to measure the time and memory consumption of the model's operators. The advantage of `DPBat` and **Greedy** is that they can dynamically select batched models according to models' characteristics, which leads to much higher utilization of GPU memory. Batched operators enable more fine-grained GPU

sharing by using less GPU memory to increase SIMD utilization. `DPBat` performs better than Greedy mainly because of its awareness of operator batching costs.

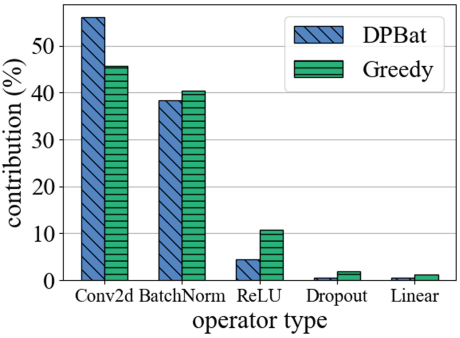


Fig. 8. Contribution of different operators to the performance improvement

We analyze the contribution of different types of operators to performance improvement. The operator types in the training models mainly include ReLU, Dropout, Linear, BatchNorm, and Conv2d. Figure 7 shows the running time of different types of operators. When batching the same number of operators, the benefits are obviously different. Batching Conv2d brings the highest benefits, followed by BatchNorm. As Fig. 8 shows, Conv2d and BatchNorm are the main sources of the benefit brought by operator batching. `DPBat` is better than **Greedy** in picking out the type of operator that brings the most performance improvement.

6 Conclusion

In this paper, we study the multi-model operator batching strategy in the NAS scenario. By characterizing the model architecture as a DFG, calculating the similarity of graphs approximately, and batching common operators of models to improve training efficiency. Our objective is to maximize the throughput of model training per unit time. We propose a heuristic algorithm named `DPBat` to guide the operator batching among multiple models. Based on Microsoft's AutoML framework NNI, we apply `DPBat` to real NAS scenarios. Experiment results show that `DPBat` significantly improves training efficiency and reduces the overhead of operator batching. Furthermore, `DPBat` achieves up to 3.7× higher training throughput than running each job on a separate accelerator, which is a common practice employed by the AutoML framework. Although we only focus on models whose DFGs are directed acyclic graphs, we believe our results will inspire future work on optimizing batching strategy between multiple models in a more general setting.

Acknowledgements. This work is partially supported by NSFC under Grant 62132009, and the Fundamental Research Funds for the Central Universities at China.

References

1. Brown, T., et al.: Language models are few-shot learners. Adv. Neural. Inf. Process. Syst. **33**, 1877–1901 (2020)
2. Bunke, H., Shearer, K.: A graph distance metric based on the maximal common subgraph. Pattern Recogn. Lett. **19**(3–4), 255–259 (1998)
3. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: a large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 248–255. IEEE (2009)
4. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
5. Fortin, S.: The graph isomorphism problem (1996)
6. Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Anal. Appl. **13**(1), 113–129 (2010)
7. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
8. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 25. Curran Associates, Inc. (2012)
9. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 779–788 (2016)
10. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
11. Tan, M., et al.: MnasNet: platform-aware neural architecture search for mobile. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2820–2828 (2019)
12. Wang, S., Yang, P., Zheng, Y., Li, X., Pekhimenko, G.: Horizontally fused training array: an effective hardware utilization squeezer for training novel deep learning models. Proc. Mach. Learn. Syst. **3**, 599–623 (2021)
13. Wu, Y., et al.: Google's neural machine translation system: bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144 (2016)
14. Zhang, Q., et al.: Retiarii: a deep learning exploratory-training framework. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020), pp. 919–936 (2020)