LC-NAS: Latency Constrained Neural Architecture Search for Point Cloud Networks

Guohao Li [1] Mengmeng Xu [1] Silvio Giancola [1] Ali Thabet [2]
Bernard Ghanem [1]
[1] King Abdullah University of Science and Technology (KAUST) [2] Meta

Abstract

Point cloud architecture design has become a crucial problem for 3D deep learning. Several efforts were made to manually design architectures targeting high accuracy in point cloud tasks such as classification, segmentation, and detection. Recent progress in automatic Neural Architecture Search (NAS) minimizes the human effort in network design and optimizes architectures for high performance. However, those efforts fail to consider crucial factors such as latency during inference, which is of high importance in time-critical and hardware bounded applications like self-driving cars, robot navigation, and mobile applications. In this paper, we introduce a new NAS framework, dubbed LC-NAS, that searches for point cloud architectures constrained to a target latency. We implement a novel latency constraint formulation to the trade-off between accuracy and latency in our architecture search. Contrary to previous works, our latency loss enables us to find the best architecture with latency near a specific target value, which is crucial when the end task is to be deployed in a limited hardware setting. Extensive experiments show that LC-NAS is able to find state-of-the-art architectures for point cloud classification in ModelNet40 with a minimal computational cost.

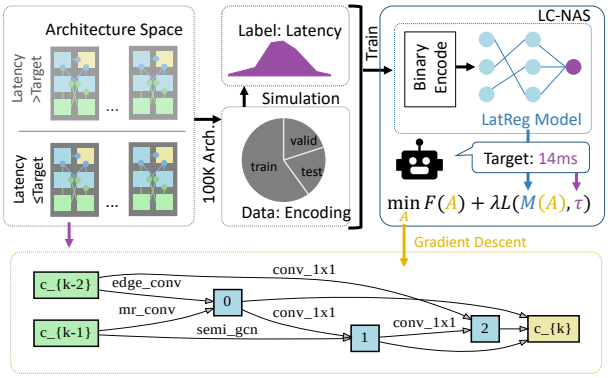

Figure 1: **LC-NAS pipeline.** 100k architectures are sampled from our defined search space and their latencies are measured. These architectures are used to train a latency predictor which takes as input a binary encoding of the architecture and estimates a latency. We integrate this predictor as a constraint into the NAS loss, where the target latency is part of the loss. The search method is optimized to discover point cloud networks that meet a target latency.

1. Introduction

In 3D computer vision, the seminal PointNet work [9] opened the way to increasingly performing architectures for processing point clouds. Similar to the 2D case, SOTA architectures achieve impressive gains in point cloud classification [7], segmentation [10], and detection [2]. However, this success comes at the expense of increased computation. Of particular interest is the computational latency during inference, a factor that is paramount to time-critical applications like self-driving cars, robot navigation, and mobile applications. These applications require fast decision making and have access to limited hardware. It is thus imperative to equip them with latency-optimized architectures. In this work, we propose a framework to constrain point cloud NAS methods in order to increase accuracy and minimize latency. We achieve this by introducing a differentiable latency estimator, which can be used to define targeted latency constraints in NAS pipelines. Since these constraints are targeted, *i.e.*, we can specify a target latency for the optimal architecture, we can use NAS solutions to search for the best architecture under a given latency constraint. This effectively allows us to move along the trade-off curve between accuracy and latency to an optimal point for any given application. We integrate our latency constraint to differentiable NAS pipeline, in order to search for latency aware Graph Convolutional Networks (GCNs), that effectively operate on point clouds. We dub this new framework Latency Constrained Neural Architecture Search (LC-NAS).

2. Methodology

2.1. Latency Constrained NAS

In order to calibrate mismatching rankings between architectures during the search phase and the evaluation phase, SGAS [6] proposes to solve the bi-level optimization in DARTS in a sequential greedy fashion. The goal of our work is to automatically design a well-performing GCN architecture that is able to run on a specific hardware platform within a given target latency. To achieve this goal, we need to take the latency constraint into consideration in the optimization during the search phase. Previous works either build a lookup table for the latency of each operations and approximate the latency by summing up corresponding latency sequentially, or use reinforcement learning to optimize the latency by treating it as a part of the score. However, none of them is feasible to apply to differential NAS methods with a search space as a DAG, since the latency of a network with a general DAG structure depends on its topology and can not be estimated by simply summing up the latency of each operations. To this end, we propose to first learn a latency regressor that is able to predict the latency of all the possible architectures on a specific device within the search space. Then, we incorporate the learned latency regressor as a constraint in the bi-level optimization objective and regularize the architectural parameters.

Search Space. We use the same search space as SGAS. Each edge in our graph convolutional cell has 10 candidate operations including *Skip-Connect*, *Conv-1×1*, *EdgeConv* [12], *MRConv* [5], *GAT* [11], *SemiGCN* [4], *GIN* [15], *SAGE* [3], *RelSAGE* [6] and *Zero*. Please refer to the original SGAS paper [6] for more details of these GCN operators. K Nearest Neighbours (KNN) is used to take the input point features of a cell to build a KNN graph by constructing dynamic edges. Obtained edges are then shared with others operations within the cell.

Architecture Encoding. To encode the model architectures, we need to encode the graph convolutional cell, the basic building block in cell-based neural architecture search. Our search space is a DAG with 9 edges and 10 candidate operations for each edge. After the search, 6 edges are retained in total. Thus, we can use a 9×10 binary encoding matrix $\mathbf{E} \in \{0,1\}^{9\times 10}$ to represent the cell, where $e_{m,n} = 1$ indicates that the operation n is chosen for the edge m. Since we only retain 6 edges, the encoding matrix $\mathbf{E}$ is a sparse matrix with exactly 6 entries with values of 1.

Latency Regressor. In this work, we consider point cloud classification on ModelNet40 as our target task and NVIDIA RTX 2080 as our target hardware platform. We sample and measure 100k random cell-architectures from the search space. We stack the sampled cell 3 times with a channel size of 128. For more details about the model hyper-parameters, please refer to the Section 3. The latency is measured by the inference time of the sampled architectures with randomly initialized weights and a random tensor of shape (*batch size* = 1, *feature dims* = 3, *num of points* = 1024). The data is then split into three folds with 60% as the training subset, 20% as the validation subset, and 20% as the testing subset. The latency ranges from 5.9 to 23.5 milliseconds (ms). We leverage a Multi Layer Perceptron (MLP) as our latency regressor (LatReg). Given the encoding matrix $\mathbf{E}$, we first vectorize $\mathbf{E}$ and feed it into three fully-connected layers. We respectively set 256, 128 and 1 neurons in each of the three layers, interleaved with sigmoid activation functions. Finally, the MLP produces one scalar value as the latency prediction. During training, we normalize the latency by the mean $\mu_{train} = 15.32$ms and the standard deviation $\sigma_{train} = 2.24$ms. We train the network from scratch, and set the batch size to 256. We employ Adam optimizer in PyTorch with the default parameters: learning rate equal to 0.001 and betas equal to (0.9, 0.999). We use mean square error (MSE) as the loss function. From the loss curve of the validation set, we find the model saturated efficiently on the 70-th epoch. The LatReg model reaches an average absolute error of 0.16ms on the test subset.

Loss function with a Target Latency as Constraint. The learning procedure of neural architecture search can be formulated as a bi-level optimization problem [8]. Previous resource-aware differentiable NAS methods usually add or multiply the latency loss as a regularizer to the cross-entropy loss [1, 13, 14]. However, these methods are not able to minimize the latency of architecture to be lower than a certain constraint, which is considered to be very important for the deployment on a specific hardware platform. Moreover, the regularization weight is hard to tune. A big regularization weight leads to efficient/fast models but with low capacity. On the other end, a small regularization weight fails to obtain efficient models. Therefore, we propose to use a hinge-loss-like regularization loss for the latency constraint as follows:

$$\min_{\mathcal{A}} \quad \mathcal{L}_{val}(\mathcal{W}^*(\mathcal{A}), \mathcal{A}) + \lambda \max(LatReg(\mathcal{E}(\mathcal{A})) - target,\ 0) \tag{1}$$

$$\text{s.t.} \quad \mathcal{W}^*(\mathcal{A}) = \text{argmin}_{\mathcal{W}}\ \mathcal{L}_{train}(\mathcal{W}, \mathcal{A}) \tag{2}$$

where $\mathcal{L}_{val}$ is the cross-entropy loss on validation set, $\mathcal{L}_{train}$ is the cross-entropy loss on training set, $\mathcal{A}$ is the architectural parameters, $\mathcal{W}$ is the network weights, λ is the regularization factor, $LatReg(\cdot)$ is the learned latency regressor, $target$ is the target latency and $\mathcal{E}(\cdot)$ is a non-differentiable binarized function that takes as input continuous architectural parameters $\mathcal{A}$ and outputs a binarized architecture encoding $\tilde{\mathbf{E}}$. Binarizing the continuous architectural parameters $\mathcal{A}$ before predicting the latency is necessary since $LatReg(\cdot)$ is trained on binary inputs. If we use

$\mathcal{A}$ as input, the latency prediction of $LatReg(\cdot)$ would be inaccurate due to the discrepancy between continuous architectural parameters and the binary architecture encoding. However, the binarized function $\mathcal{E}(\cdot)$ is non-differentiable since it involves some rules/heuristics to derive a discrete architecture encoding such as choosing 6 edges out of 9 and selecting a non-zero operation with the highest weight. In order to obtain the gradient of the latency loss with respect to architectural parameters $\mathcal{A}$, we introduce an approximated gradient-based approach.

Optimizing the Latency Constraint. We propose a modified approximated gradient-based approach to optimize the architectural parameters. We denote the softmax output of $\beta_{m,n} = softmax(\alpha_{m,n}|\boldsymbol{\alpha}_m) = \frac{\exp(\alpha_{m,n})}{\sum_k \exp(\alpha_{m,k})}$, where $\alpha_{m,n}$ is the architectural parameter of operation n of edge m. To compute the gradient of $\mathcal{E}(\cdot)$ with respect to $\mathcal{A}$, we trust the selection rules/heuristics as a linear operation by approximating with multiplying an element-wise mask $\boldsymbol{\zeta}$, where $\zeta_{m,n} = \frac{1}{\beta_{m,n}}$ if $n = n^*$ and $\zeta_{m,n} = 0$ if $n \neq n^*$. Note that n^* is the chosen operation of edge m. Therefore the binarized function becomes $\mathcal{E}(\alpha_{m,n}) = \tilde{e}_{m,n} \approx \beta_{m,n} \cdot \zeta_{m,n}$ and the gradients can be obtained. We denote the latency loss term as $\mathcal{L}_{lat}$. We have:

$$\frac{\partial \mathcal{L}_{lat}}{\partial \alpha_{m,n}} = \sum_k \frac{\partial \mathcal{L}_{lat}}{\partial \beta_{m,k}} \cdot \frac{\partial \beta_{m,k}}{\partial \alpha_{m,n}} = \sum_k \frac{\partial \mathcal{L}_{lat}}{\partial \tilde{e}_{m,k}} \cdot \frac{\partial \tilde{e}_{m,k}}{\partial \beta_{m,k}} \cdot \frac{\partial \beta_{m,k}}{\partial \alpha_{m,n}} \tag{3}$$

where $\frac{\partial \beta_{m,k}}{\partial \alpha_{m,n}} = \beta_{m,n} - \beta_{m,n}^2$ if $n = k$ and $\frac{\partial \beta_{m,k}}{\partial \alpha_{m,n}} = -\beta_{m,n} \cdot \beta_{m,k}$ if $n \neq k$. Since $\frac{\partial \tilde{e}_{m,k}}{\partial \beta m,k} = \zeta_{m,k}$. We obtain the gradient as follows:

$$\frac{\partial \mathcal{L}_{lat}}{\partial \alpha_{m,n}} = \frac{\partial \mathcal{L}_{lat}}{\partial \tilde{e}_{m,n^*}} \cdot \frac{1}{\beta_{m,n^*}} \cdot \frac{\partial \beta_{m,n^*}}{\partial \alpha_{m,n}} = \begin{cases} (1 - \beta_{m,n^*}) \cdot \frac{\partial \mathcal{L}_{lat}}{\partial \tilde{e}_{m,n^*}} & \text{for } n = n^* \\ -\beta_{m,n} \cdot \frac{\partial \mathcal{L}_{lat}}{\partial \tilde{e}_{m,n^*}} & \text{for } n \neq n^* \end{cases} \tag{4}$$

In this way, we can update the architectural parameters $\mathcal{A}$ using the gradient above. Therefore, the latency constraint can be optimized during search.

3. Experiments

Our target task is classification on ModelNet40 using NVIDIA RTX 2080. We sample 100k architectures from the search space and measure their latency to build a dataset. Then, we train a latency regressor on the latency dataset. After that, we use the pre-trained latency regressor to constrain the architecture search on ModelNet10 with latency targets on ModelNet40 ranging from 6ms to 18ms. We then evaluate the performance of obtained architectures on ModelNet40 by training from scratch. We also transfer our architectures to a completely different task, part segmentation, to show the generalization of the architectures. Finally, we conduct a thorough ablation study to demonstrate the effects of regularization strength, loss function, and the choice of hyper-parameters for our models.

3.1. LC-NAS on ModelNet10

Training Settings. During the training of the search phase, 1024 points with only 3D coordinates (x, y, z) are sampled from the 3D models in ModelNet10 as input. We set the regularization factor λ as 0.5 and vary the target latency from 6ms to 18ms with a step of 2ms. The other settings follow those of SGAS. We use 2 cells with 32 initial channels and search for the architectures for 50 epochs with batch size 28. Two different optimizers are used for optimizing model weights $\mathcal{W}$ and $\mathcal{A}$. We use SGD with an initial learning rate 0.005, momentum 0.9, and weight decay 3×10^{-4} to optimize the model weights. An Adam optimizer with an initial learning rate 3×10^{-4}, momentum $(0.5, 0.999)$, and weight decay 10^{-3} is used for the architecture parameters $\mathcal{A}$. We use the same edge selection criterion as SGAS Criterion 2. LC-NAS begins to determine one operation for a selected edge every 7 epochs after warming up for 9 epochs. A history window of size 4 is used for selection stability. We increase the batch size by 4 after each decision. A search run costs 0.15 GPU days on one NVIDIA RTX 2080, which is the same for SGAS. This means the extra computation overhead of adding the latency constraint is negligible.

3.2. Architecture Evaluation on ModelNet40

After searching on ModelNet10, we get 7 searched cell structures with target latency 6ms, 8ms, 10ms, 12ms, 14ms, 16ms and 18ms respectively. We build a large network for each cell with the same hyper-parameters that are used to generate the architecture for learning the latency regressor. We then train them on ModelNet40 from scratch. We evaluate the performance on ModelNet40 using two metrics: the overall accuracy (O.A.) and class accuracy (C.A.).

Training Settings. 1024 points with 3D coordinates are used as input. We stack the searched cell 3 times with channel size 128. An MLP with 1024 neurons is used to fuse the concatenation of all the output features of 3 cells. And then, the fused features are aggregated through a max-pooling layer and an average pooling layer. We concatenate the aggregated features from two pooling layers and feed them into a 3-layer MLP classifier with $\{512, 256, 40\}$ neurons to classify the input point clouds into 40 categories. Our architectures are all trained for 400 epochs with batch size 32.

Evaluation Results and Analysis. We report the best overall accuracy (O.A.) and the corresponding class accuracy (C.A.) on the test dataset for all the 7 discovered architectures in Table 1. We observe in Table 1 that LC-NAS is able to meet the target latency in the vast majority of cases (only

Table 1: **Evaluation on ModelNet40.** We report the target, predicted and measured latencies, as well as the number of parameters, the overall accuracy and class accuracy. We observe that our architectures consistently meet the target constraint while achieving high accuracy.

	Latency (ms)				Accuracy (%)	
Method	**Target**	**Predicted**	**Measured**	**# Param. (M)**	**O.A.**	**C.A.**
LC-NAS-18	18	17.06	16.66	3.91	**92.79**	89.66
LC-NAS-16	16	13.71	13.57	3.91	92.62	90.13
LC-NAS-14	14	12.64	12.41	3.91	92.42	89.16
LC-NAS-12	12	10.07	9.96	3.85	92.34	89.57
LC-NAS-10	10	11.02	11.09	3.86	92.75	**90.76**
LC-NAS-8	8	7.84	7.51	3.71	90.40	85.36
LC-NAS-6	6	6.12	5.47	3.61	90.51	84.71
Average	-	11.21	10.95	3.82	91.98	88.48

exception is LC-NAS-10, where the actual latency is only over target by 1ms).

4. Conclusion

We presented an automatic neural architecture search that considers the latency factor in the search. We designed a loss function that constrains the latency for a given hardware. We show with empirical results that our architectures LC-NAS reach the latency it has been designed for on ModelNet10 and generalize on ModelNet40. We envision LC-NAS to be used in time-constrained applications such as autonomous driving, robotics and embedded systems, where latency is of paramount importance for the fulfillment of the vision task.

Acknowledgments. This work is supported by the KAUST Office of Sponsored Research through VCC funding.

References

[1] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018. 2

[2] Zhipeng Ding, Xu Han, and Marc Niethammer. Votenet: A deep learning label fusion method for multi-atlas segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 202–210. Springer, 2019. 1

[3] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017. 2

[4] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 2

[5] Guohao Li, Matthias Müller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *The IEEE International Conference on Computer Vision (ICCV)*, 2019. 2

[6] Guohao Li, Guocheng Qian, Itzel C Delgadillo, Matthias Müller, Ali Thabet, and Bernard Ghanem. Sgas: Sequential greedy architecture search. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 2

[7] Yangyan Li, Rui Bu, Mingchao Sun, Wei Wu, Xinhan Di, and Baoquan Chen. Pointcnn: Convolution on x-transformed points. In *NeurIPS*, 2018. 1

[8] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 2

[9] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 1(2):4, 2017. 1

[10] Hugues Thomas, Charles R. Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J. Guibas. Kpconv: Flexible and deformable convolution for point clouds. *ArXiv*, abs/1904.08889, 2019. 1

[11] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017. 2

[12] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 2019. 2

[13] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019. 2

[14] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018. 2

[15] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *ArXiv*, abs/1810.00826, 2018. 2