國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

基於架構生成器及搜尋進行硬體平台感知之神經網路剪枝

Platform-aware Network Pruning with Architecture Generator and Search

陳柏穎

Bo-Ying Chen

指導教授：簡韶逸 博士

Advisor: Shao-Yi Chien, Ph.D.

中華民國 110 年 07 月

July 2021

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

國立臺灣大學碩士學位論文
口試委員會審定書

基於架構生成器及搜尋進行硬體平台感知之神經網路剪枝
Platform-aware Network Pruning with Architecture Generator and Search

本論文係陳柏穎君（R08943002）在國立臺灣大學電子工程學研究所完成之碩士學位論文，於民國 110 年 07 月 13 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

簡韶逸 （指導教授）

黃朝宗　　鄭文皇

系主任、所長

doi:10.6342/NTU202101554

致謝

論文付之梨棗前回首這段碩士生涯，過程雖幾經曲折，有幸受到無數人的幫助，使得我順利完成碩士學業。朝人生的新階段邁進。

首先感謝做為我最堅實後盾的家人，一路栽培至今，給予我極大的空間發展，感謝在我碩士期間默默的支持、鼓勵、關心，及充分的信任，讓我心無旁騖地專注在研究中，最終才有順利的研究成果。

感謝我的指導教授－簡韶逸教授，給我研究範圍後讓我從中自由研究與鑽研，在我研究不順利時依然對我有信心，並定期確認我的研究進度及給予研究上的建議，讓我度過自由且自律的碩士生涯。

感謝實驗室的學長姐及同學們，感謝博班致廷學長在我研究過程遇到瓶頸時與我討論及給予精闢的建議，以及到碩士尾聲投稿時協助我一起修改論文，讓我對研究有更深一層的認識。感謝禹澄學長與我討論研究題目，激盪出可能的研究方向。感謝洋彬學長與汶璁協助我研究前人留下的艱深研究成果，由其汶璁在我口試前多次陪我練習，讓我得以順利完成口試。感謝華揚與我擔任助教時與業界先進們多次對課程討論、溝通及協調，是難得且難忘的經驗。除此之外感謝實驗室的其他人，和諧的實驗室氣氛使得研究路上有人互相照應。

祝各位未來都能朝心之所嚮前進。

2021.07.20 陳柏穎

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

中文摘要

卷積神經網絡廣泛用於計算機視覺。需要存儲的大量參數和高計算複雜度導致它們需要具有大量資源的平台來實現某些場景，例如實時應用。為了在資源有限的平台上適應這些計算密集型網絡，研究人員開發了許多網絡壓縮技術，以在減少資源消耗的同時保持性能。成功的壓縮方法的關鍵是在給定資源（例如，參數、推理延遲）約束下產生具有最高性能的網絡。

剪枝是一種有效的網絡壓縮方法。它估計每個濾波器的重要性並消除那些不太重要的過濾器，直到滿足資源限制。雖然現有方法僅考慮網絡中的參數總量或浮點運算 (FLOPs) 作為約束，但這些指標忽略了網絡如何在目標平台上執行。在本論文中，將推理延遲等平台特性引入到修剪指標中。我們提出了一種新的平台感知過濾器修剪方法，可以擴大整個網絡的搜索空間。稱為平台感知架構生成器和搜索（PAGS），它可以生成給定延遲約束的網絡架構並擴展整體模型架構搜索空間。在搜索階段，我們從生成器構建的候選集中搜索最佳修剪結構。最後，進行典型的剪枝程序將預訓練的模型剪枝為最佳剪枝結構和微調它以恢復性能。大量實驗表明，在相同的延遲約束下，我們的方法可以實現比最先進的方法更好的性能和更低的延遲。

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Platform-aware Network Pruning with Architecture Generator and Search

Bo-Ying Chen

Advisor: Shao-Yi Chien

Graduate Institute of Electronics Engineering

National Taiwan University

Taipei, Taiwan

July 2021

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Abstract

Convolutional neural networks (CNNs) are widely used in computer vision. A large amount of parameters to be stored and the high computation complexity of CNNs lead to that they need platforms with plenty of resources to achieve some scenarios, such as real-time applications. To adapt these computation-intensive networks on platforms with limited resources, researchers have developed many network compression techniques to reduce resource consumption and maintain performance at the same time. The key to a successful compression method is to optimize the network with the highest performance under the given resource (e.g., parameters, inference latency) constraints.

Filter pruning is an effective method for network compression. It estimates the importance of each filter and eliminates those who are less critical until the constraints are met. While existing methods only consider the total number of parameters or floating-point operations (FLOPs) in a network as constraints, these metrics neglect how networks are executed on target platforms. In this thesis, platform characteristics such as inference latency are introduced to pruning metrics. We propose a novel method for platform-aware filter pruning that enlarges the total networks' searching space. It is called Platform-aware Architecture Generator and Search (PAGS), which can generate network architectures given a latency constraint and expand the overall model architecture searching space, followed by the search algorithm. In the searching stage, we search for the best-pruned structure from the candidate set constructed by our generator. Lastly, typical pruning procedure will be conducted to prune the pre-trained model to the best-pruned structure and

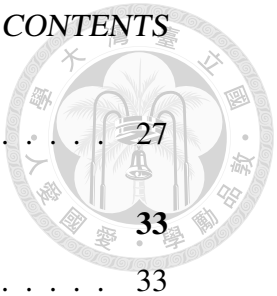
doi:10.6342/NTU202101554

fine-tune it to recover performance. Extensive experiments demonstrate that under the same latency constraint, our method can achieve better performance and lower latency than state-of-the-art methods.

doi:10.6342/NTU202101554

Contents

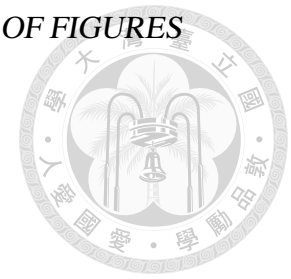
doi:10.6342/NTU202101554



doi:10.6342/NTU202101554

List of Figures

doi:10.6342/NTU202101554



doi:10.6342/NTU202101554

List of Tables

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Chapter 1

Introduction

The rapid growth of machine learning and deep learning has been adopted in many research fields such as computer vision, natural language processing, etc. Convolutional Neural Networks (CNNs) are extensively used in various vision tasks in computer vision, for example, image recognition [4], object detection [5], image super resolution [6]. The breakthrough performance is attributed to the elaborate design of CNNs. To achieve higher performance in those vision tasks, CNNs are manually designed to be deeper, containing more different operations, and more complicated. These designs enable CNNs to achieve better performance while these networks become more computation-intensive. The latest technological advancements make computer hardware more powerful than ever, dealing with these computation-demanding workflows. However, these computation-intensive and memory-intensive CNNs are hardly deployed to platforms with limited computation and storage resources since their constraints include run-time latency, storage for parameters, and energy consumption. Therefore, the most critical part is how to strike a balance between accelerating the original large network and performance drop.

Enormous network compression methods have been proposed to solve the problem mentioned above, include knowledge distillation [7, 8], quantization [9, 10, 11], network pruning [12, 3, 13, 14, 15, 16, 17, 18, 19], and network design [20,

doi:10.6342/NTU202101554

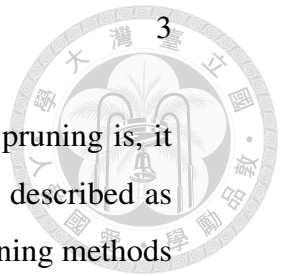
21, 22].

Among the methods mentioned above, quantization converts the floating-point parameters of the network into low bit width and thus reduce the storage of parameter. However, it couldn't reduce the number of computations. Knowledge distillation consists of two networks: the original network (teacher) and a compact network (student). Then the teacher network will transfer its informative knowledge to the student network. Both knowledge distillation and network design need human experts with domain knowledge to carry out more compact models. Network pruning has the advantage of reducing both network parameters and the number of computations. Moreover, pruning methods only need to set criteria for evaluating the importance of pruned units and then eliminating those that are less important. After the pruning process, it will generate a compact model compared to the original one.

1.1 Network Pruning

Consider an over-parameterized neural network; the idea of network pruning is to investigate the network importance, followed by eliminating the redundant part of the net, and finally yield its optimal sub-network with a tolerable performance drop. The unit being pruned varies in size. It can be a single value in the weight of the network, a channel in the feature map, or even a whole convolutional filter. Based on that, the pruning process can be divided into unstructured and structured pruning. Weight pruning resulting in sparse matrix belongs to the former. Channel pruning or filter pruning belong to the latter. Since unstructured pruning yields sparse matrices, it needs dedicated hardware to speed up, which is not unrealistic for regular hardware. On the other hand, structured pruning can fit all kinds of hardware. Therefore, it's more popular and commonly used. The objective of pruning is to reduce the storage and the number of computations. Therefore, the constraints in the pruning process would be the total number of parameters

doi:10.6342/NTU202101554



left, floating-point operations (FLOPs). No matter what type the pruning is, it often follows a similar pruning procedure. The procedure can be described as follow: Given a trained CNN, set the pruning constraints. The pruning methods repeatedly evaluate the importance of the unit being pruned and removed that are less important until the sub-network meets the constraints. Then the pruned network's fine-tuning process will be conducted to recover the performance drop during the pruning process.

1.1.1 Unstructured Pruning

CNNs imitate neuron connections in brains. If the information (feature map) in the network is important, it will pass through the activation function to the next neuron. If the value of weight in the network is small, it's possible that the information would not activate the neuron, which means it will not pass on to the next layer. The common weight pruning procedure is illustrated with Figure 1.1. Since neural networks commonly have a vast number of parameters to retain their superior performance, it might cause redundancy. If the redundancy in the model can be removed, a more compact model can be generated without sacrificing too much performance. Many studies [3] have shown that eliminate a portion of the original network would not cause too much performance drop. [23] considers the magnitude of each weight in the networks. Small values of weights are considered to be redundant. Removing the smallest weight will generate a sparse network, which can reduce the number of parameters. [24] combine weight pruning and other compression techniques to lower the amount of storage further. Weight pruning addresses the mass amount of storage in CNNs. However, it can not save the actual total number of computations. Since weight pruning yields a pruned model with sparse weights, the sparse matrices need specialized hardware to deal with. Thus unstructured pruning methods need to combine with dedicated hardware to truly decrease computation, which is not suitable in real-world scenarios.

doi:10.6342/NTU202101554

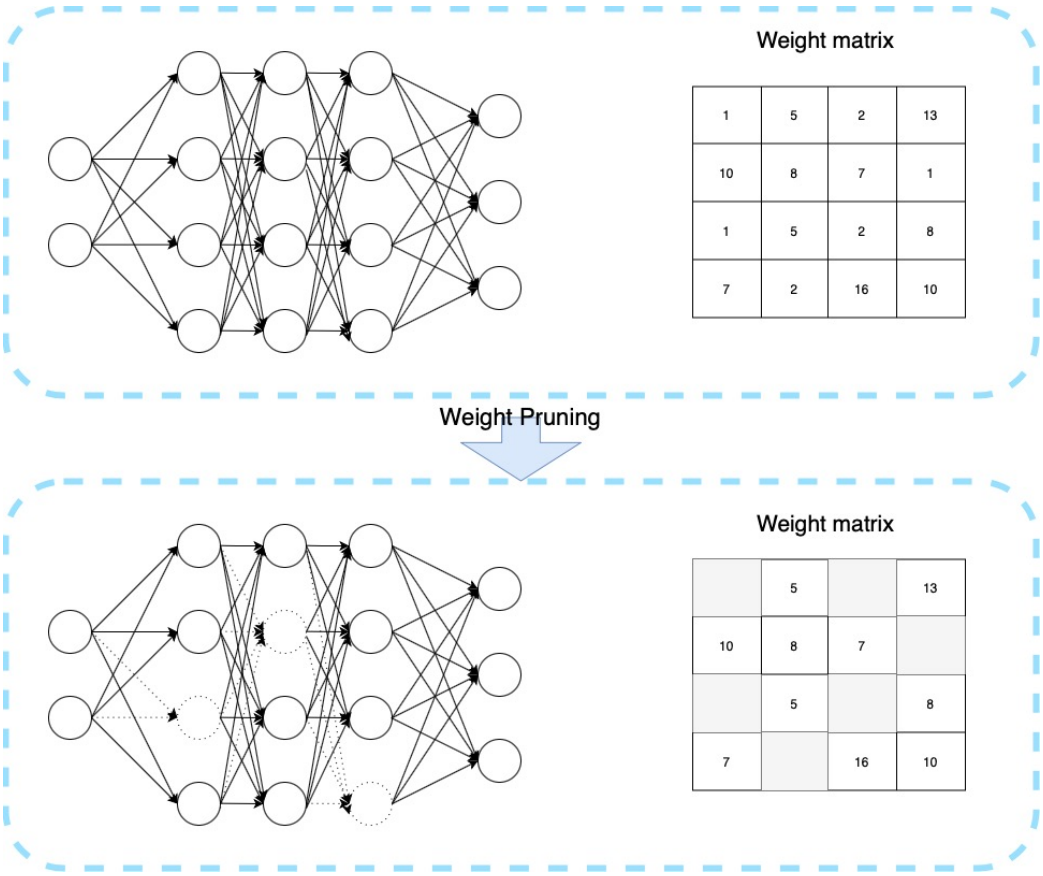


Figure 1.1: **Illustration of weight pruning.** In the process of weight pruning, suppose we set a threshold, weights smaller than the threshold will be set to zeros, and only important weights are left.

doi:10.6342/NTU202101554

1.1.2 Structured Pruning

Unlike unstructured pruning, structured pruning can fit all kinds of hardware without specialized design. Structured pruning, also known as channel pruning or filter pruning, determines the importance for each structured unit, such as a channel in the feature map or a filter in a layer. Then it classifies those units into two sets, an important one and a less important one for each iteration. Followed by iteratively pruning the unimportant ones and fine-tuning in the network. The common structured pruning procedure is illustrated with Figure 1.2. The pruning process is often executed iteratively rather than one-shot because iteratively prune the structured units can make less impact on overall performance. In recent studies, various criteria are proposed to determine the "importance" for each structured unit. For example, the norm [3] or sparsity [25] of a filter is used. More recently, complicated evaluations [26, 17] achieve exceptional performance by estimating the importance of a filter based on its impact on the loss of accuracy drop when being eliminated.

1.2 Challenge

Lack of information of the hardware platform. Different hardware platforms have their own design characteristics depending on their application purposes. Figure 1.3 shows some well-known hardware such as Central Processing Unit (CPU), Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA), to Application Specific Integrated Circuit (ASIC). CPU is more flexible than the other platforms. ASIC is the most efficient platform among them. The same network executed on these platforms will have different results. In other words, the same number of parameters or FLOPs in a network can't represent real-world network execution on the hardware platform.

Moreover, we found that most pruning methods usually use the total number of parameters or total FLOPs as pruning constraints, which are indirect metrics for the

doi:10.6342/NTU202101554

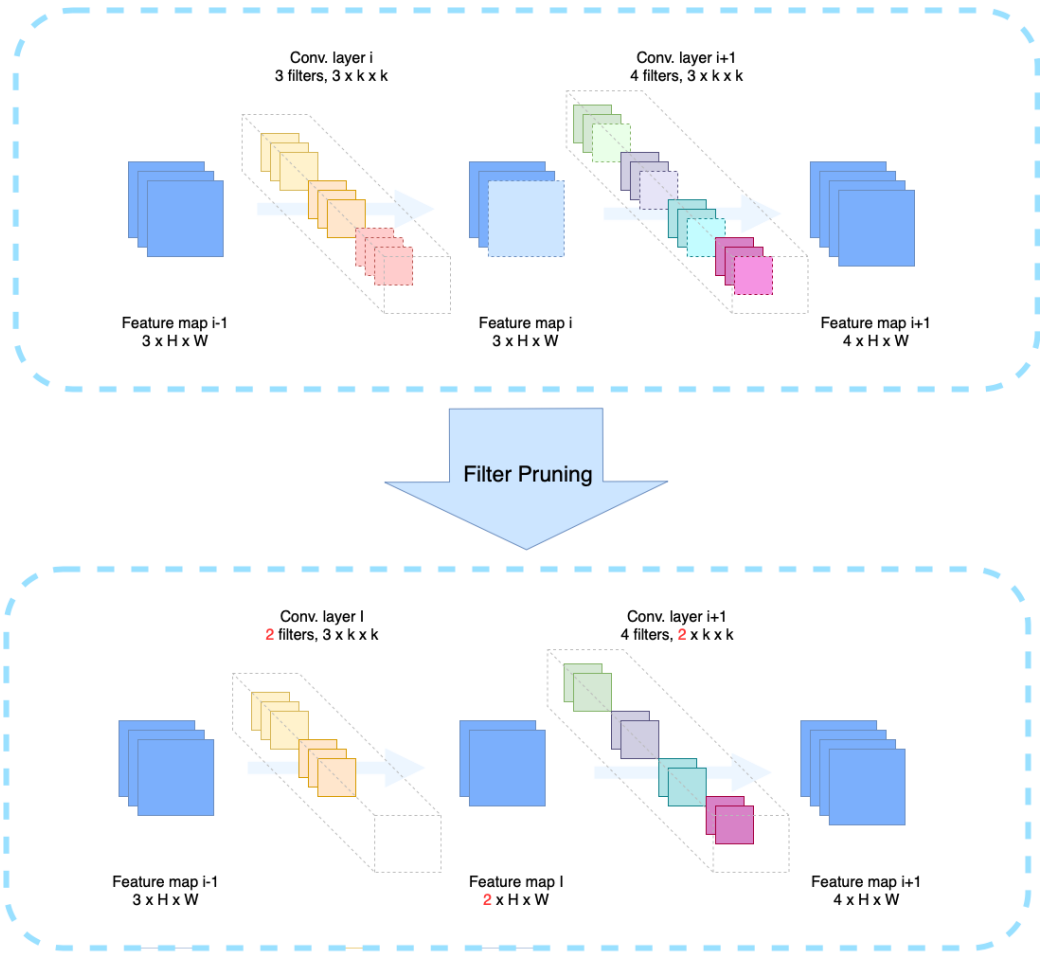


Figure 1.2: **Illustration of filter pruning.** In the process of filter pruning, suppose that we want to remove the 1^{st} filter of convolutional layer i in a CNN, which results in that the 1^{st} channel of output feature map i would disappear, the 1^{st} channel for each filter in the next convolutional layer, layer $i+1$, become ineffective and should be removed accordingly.

doi:10.6342/NTU202101554

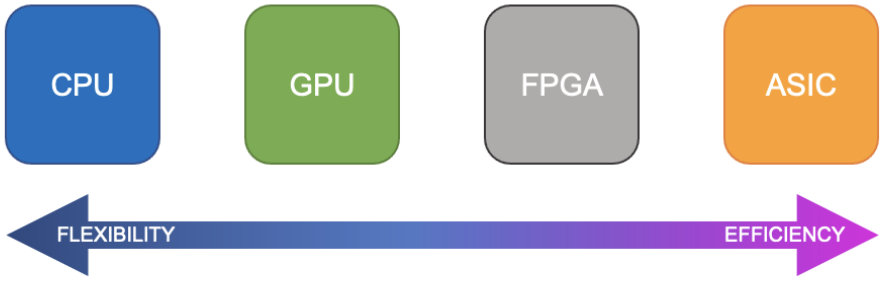


Figure 1.3: **Hardware comparison.** From the leftmost to the rightmost: CPU, GPU, FPGA, and ASIC. CPU has higher flexibility, and ASIC has higher efficiency.

target devices. For example, 30% pruning ratio on the total number of parameters can't guarantee that the inference latency will exactly drop by 30%. To make pruned networks executed efficiently on the device, direct metrics such as inference latency and energy consumption should be considered. The pruning process should collaborate with the platform deeply to yield a pruned network being executed efficiently on the platform and acceptable performance drop simultaneously.

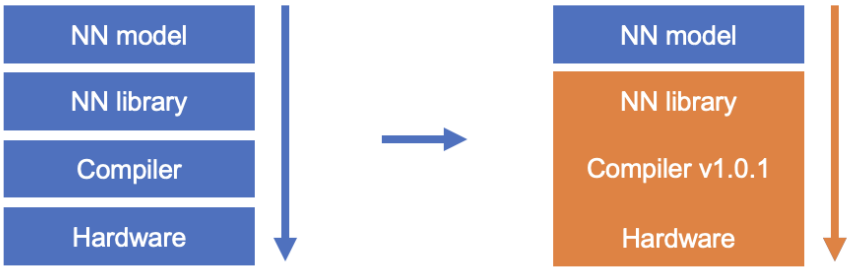


Figure 1.4: **Illustration of platform.** We view all parts underneath the Neural Network model (NN model) as a whole platform. In the process of filter pruning, suppose the deep learning compiler of the platform has been updated; we can view them as a new platform and execute our pruning process.

Moreover, the execution efficiency of the network depends not only on the hardware but also on the related acceleration library and the deep learning compiler.

doi:10.6342/NTU202101554

Any improvements among them have impacts on the inference speed. Thus, we regard all parts underneath the model as a complete platform, including acceleration library, deep learning compiler, and hardware. This allows the pruning process to fast adapt to platforms. If one component of the platform has been updated, we can view them as a new platform and execute the pruning process, which is shown in Figure 1.4.

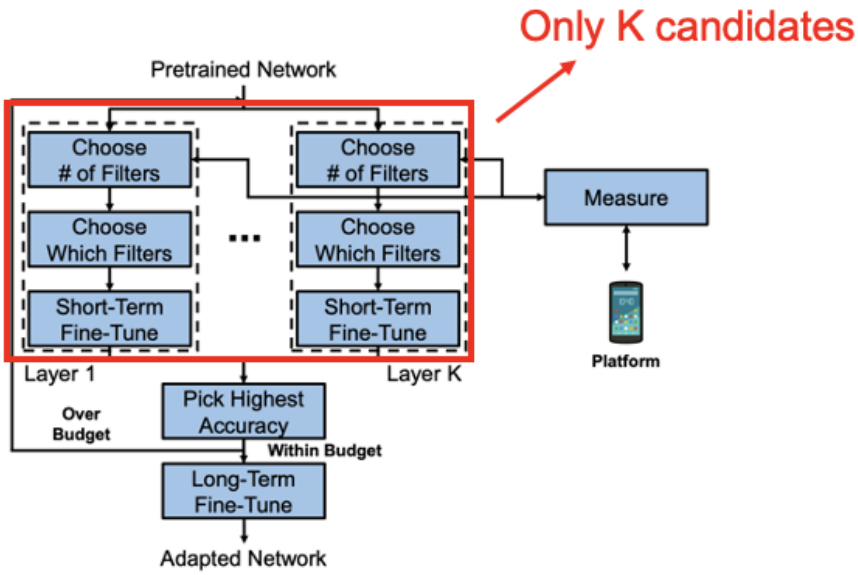


Figure 1.5: **Hardware comparison.** From the leftmost to the rightmost: CPU, GPU, FPGA, and ASIC. CPU has higher flexibility, and ASIC has higher efficiency.

Limited search space. Another challenge is that most iteratively pruning methods have limited candidates at each pruning iteration, which narrow the search space of pruning candidates and use the greedy method to select the current best candidate for the next pruning iteration. This might result in missing the opportunity to find the optimal pruned model. For example, NetAdapt [2] generate K (number of layers of the pretrained model) pruned network candidates meeting the current resource constraint at each pruning iteration, shown in Figure 1.5. This pruning search space shrinkage approach hinders the pruning method from

doi:10.6342/NTU202101554

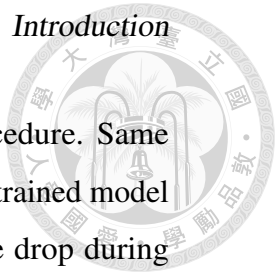
producing better-pruned network meeting the resource budget.

To address the challenges mentioned above, we propose a novel method named *Platform-aware Architecture Generator and Architecture Search* (PAGAS), which integrates the information of inference latency of the platform into our pruning procedure. The platform-aware architecture generator will generate network architectures meeting the target latency constraint. Moreover, we expand the overall search space of pruning candidates initially, followed by the architecture searching algorithm to find the best model architecture given the resource constraint. Lastly, we use fine-tuning to recover the performance drop during the pruning process.

1.3 Contribution

To sum up, two problems are stated in existing methods for filter pruning: 1) Lacking information of the hardware platform. Only using indirect metrics can not adapt the pruning methods to a variety of platforms. 2) Limited search space. With limited search space, it's not capable of obtaining better pruning candidates, also leading to sub-optimal performance. To address these two problems, we propose *Platform-aware Architecture Generator and Search* (PAGS), which consists of a platform-aware model architecture generator (AG) and a searching algorithm. We first aim to expand the search space of pruning candidates. Given the target device, we train an architecture generator which is an auto-encoder network structure with self-generated data collected on the target device. After the training process is completed. At the generating stage, we can use the decoder network of the auto-encoder network structure as the generator to yield various network structures meeting the given inference latency budget. Secondly, we use architecture search (AS) inspired by [1] to search for the best model structure out of the architecture set generated from our AG. Different from them, we automatically search overall architectures rather than manual settings maximum pruning ratio for each layer. Furthermore, we can also prune models more precisely given the constraints.

doi:10.6342/NTU202101554



Lastly, the searched best structure is the target of our pruning procedure. Same as [2], we use the L2-norm as the filter importance to prune the pre-trained model with the best structure and fine-tune it to recover the performance drop during the pruning process. We can achieve higher accuracy under the given inference latency constraint compared with the state-of-the-art of platform-aware pruning method. Furthermore, we can achieve higher accuracy and faster inference speed at the same time under the given inference latency constraint compared with the non-platform-aware method.

To the best of our knowledge, our work is the first to apply the generative model to solve the pruning problems.

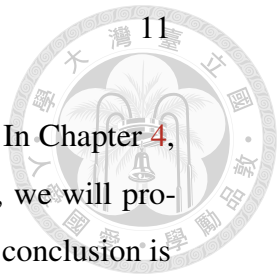
The main contributions of our work are:

- We propose a platform-aware architecture generator that can generate different model architectures meeting the given latency constraints. Therefore, with more candidates, there is a higher opportunity to find the best-pruned models satisfying the constraints.
- We integrate architecture search in our work. With the help of the search process, we can find the best-pruned model over candidates generated by the generator.
- Extensive experiments validate that our proposed method not only outperforms state-of-the-art methods in accuracy but also requires less inference time than existing methods.

1.4 Thesis Organization

In this chapter, we introduce platform-aware pruning and present the main contributions in this thesis. The related works are mentioned in Chapter 2. In Chapter 3, we will briefly give the problem formulation and some notation used in our thesis. Then we present our architecture generator and give the details regarding the theory of the deep conditional generative model and how we train the architecture

doi:10.6342/NTU202101554



generator. Lastly, we'll introduce our architecture search method. In Chapter 4, we will demonstrate the effectiveness of the generator. Moreover, we will provide experimental results to support our method. At last, our thesis conclusion is in Chapter 5.

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Chapter 2

Platform-aware Filter Pruning

2.1 Related Work

2.1.1 Filter Pruning

Network pruning is a common compression approach to obtain a compact pruned network from a large pre-trained network by eliminating the less important parts. Compared with early weight pruning [12], which only removes the individual redundant parameter of the weight matrices in CNN, filter pruning is a more efficient method to reduce the total computation consumption by regrading a whole convolutional filter in the model as a pruned unit. To distinguish between important filters and less important ones, some works focus on determining the importance of filters in each layer and remove unimportant ones "layer-by-layer". Besides, some works pay attention to the global methods that simultaneously evaluate and prune filters in the whole network.

Layer-wise filter pruning. Inspired by the mechanisms for propagating action potential in neurons, the larger action potential is more likely to transmit the information to the next neuron. Some works [3, 23, 25, 19] presume a connection between the importance of a filter and its corresponded parameter values, such as its L1-norm [3], L2-norm [23], sparsity [25] or the distance to the geometric median

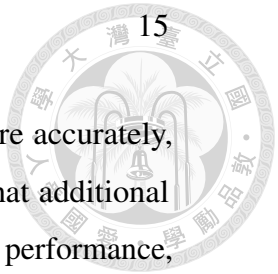
doi:10.6342/NTU202101554

of filters in a single layer [19]. The larger the value is, the more important the filter is. On the other hand, some works use training data to yield the criterion for filter removal [13, 15, 27]. [15] and [13] choose filters that can minimize feature reconstruction error in each layer by solving LASSO problem [28]. Furthermore, DCP [27] introduces additional discrimination-aware losses to guide the selection of redundant filters and increase the discrimination ability of features. However, each layer is connected to its previous and the next layer, and the layer dependency should be considered. All of the layer-wise methods mentioned above can only compare filters in the same layer; in other words, lacking cross-layer evaluation. Moreover, layer-wise filter pruning is inefficient since it requires a pre-defined pruning ratio for each layer, which is less flexible for the left network and leads to a worse performance result.

Global filter pruning. Since the dimensions of the filters in each layer are different, making filters in different layers comparable is a problem to be solved. To make the estimated values for filter importance globally comparable, Molchanov *et al*. [26] takes advantage of a layer-wise normalization approach to re-scale the original importance score, which utilizes Taylor's expansion of the loss impact caused by the removal of filters. NISP [16] measures the importance of features in the final response layer (FRL). FRL is the second-to-last layer before classification, and it should play key roles in full network pruning since they are the direct inputs of the classification task. Then it propagates the importance score of each filter in the whole network from the final response layer back to each layer before. These methods only take the total number of parameters or the number of computations as their pruning constraint. None of them consider the difference between deployed devices.

Some works [14] try to utilize the batch-normalization (BN) layers [29]. They enforce the sparsity of the scaling factor γ in the BN layer by adding a regularization term in the training stage and prune filters depending on a global threshold over

doi:10.6342/NTU202101554



the value of γ. Recently, to evaluate the importance of a filter more accurately, Molchanov *et al*. [17] modifies the Taylor expansion method so that additional normalization is not required. Although they can obtain a promising performance, none of these methods consider the information of the target devices.

2.1.2 Platform-based Network Optimization

Suppose there is a network being executed on two different platforms; it can't guarantee the network will have the same inference latency since the two platforms have different hardware designs, which will affect how hardware platforms process and execute networks. Here we focus on viewing the whole platform as a unit rather than the hardware itself. The platform usually consists of several components such as acceleration library, deep learning compiler, and underlying hardware. Improvement of any of them will enhance the overall performance, make the overall inference latency lower.

However, most pruning methods focus on reducing the total number of parameters or total number of FLOPs. Very few works introduce the information of the platform into the whole pruning process. NetAdapt [2] is the first work that introduces the platform information into the pruning process. It eliminates the requirement of platform-specific knowledge by using empirical measurements to evaluate the direct metrics such as the inference latency of the network. The measurements guide NetAdapt to generate the network that meets the resource constraint. At each pruning iteration, it will generate a set of network proposals meeting the current resource budget for the next iteration. It constructs a look-up table (LUT) containing inference latency on the target platform for each layer of a given pre-trained network with various input and output channels. During the iterative pruning process, it will only change the dimension of a single layer at a time, and other layers remain the same dimensions. Then it searches the LUT and tries to find the layer's dimension to meet the current constraint. If a network has n layers, it will generate n proposals at each iteration. Followed by the greedy

doi:10.6342/NTU202101554

selection, it picks the current best model for the next round. Repeat the process until the resource budget is met.

However, this LUT-based method is not as accurate as the direct measurement for the whole network. Since every measurement has fluctuation, the combination of these measurements will accumulate the fluctuation, making it more inaccurate. Secondly, the number of network proposals is very limited at each iteration, leading to a sub-optimal pruned network. Compared with NetAdapt, Our proposed method uses direct measurement on the whole network to produce accurate latency measurement and enlarges the network proposal search space to produce a pruned network result with lower performance drop and closer to the given resource budget.

2.1.3 Deep Generative Models

Deep generative models have been widely used in many research areas such as super-resolution and natural language processing. Among them, variational autoencoder (VAE) [30] is famous for its encoder-decoder network structure and its decent performance in generating predictions. At the training stage, input data samples are fed into the encoder to extract the input features to low dimensional latent space; the decoder will reconstruct it back to the same dimension as the inputs. At the generating stage, VAE uses a Gaussian latent variable for output prediction.

Nevertheless, it can't control the output since the latent variable is drawn from the Gaussian distribution. To address this problem, conditional variational autoencoder (CVAE) [31] is proposed. The training and the generating stage of CVAE is illustrated in Figure 2.1. The generated output class can be assigned first. At the training stage. the data and the corresponding label form a training pair as the inputs for the encoder network. The encoder network will encode the inputs into a low latent space. Then the decoder takes the latent variable and the assigned output class as the inputs and generates the desired output class prediction. At

doi:10.6342/NTU202101554

generating stage, the predicted output class can be assigned. The decoder, or usually being called the generator, takes a latent variable and an assigned output class, it will generate the desired output prediction. Even the same latent variable with a different assigned label can generate the correct prediction with the conditional generative model.

The deep conditional generative model is usually used to generate images. Our platform-aware architecture generator is based on the CVAE, which can generate model structures meeting the given latency constraint. To the best of our knowledge, our work is the first to apply the deep conditional generative model to solve the pruning problems.

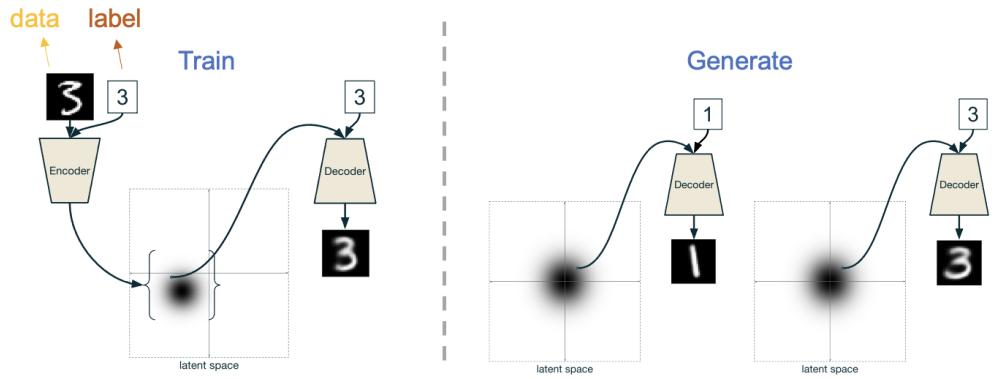


Figure 2.1: **Conditional Generative Model.** Left: At training stage. the data and the corresponding label form a training pair as the inputs for the encoder network. The decoder will reconstruct the input data. Right: At generating stage, the decoder will be used as the generator. Sample latent variable from known distribution and concatenate the desired output class label as the inputs for the decoder, the decoder network will generate predicted output.

doi:10.6342/NTU202101554

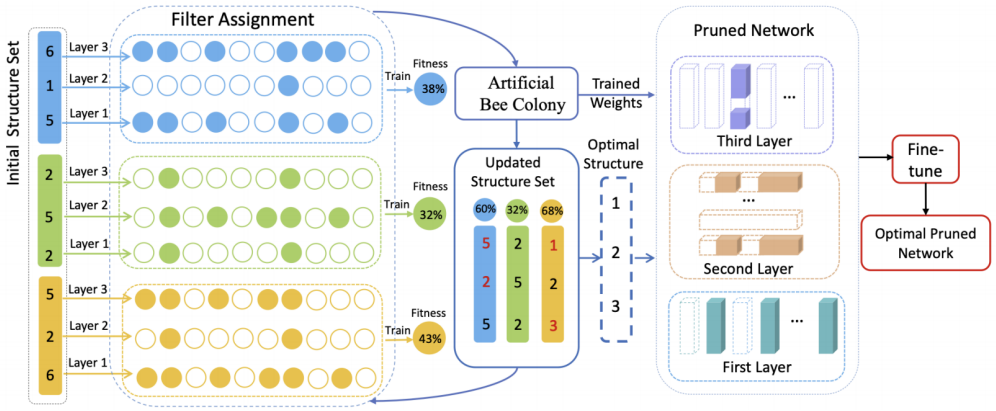


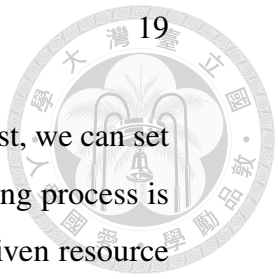
Figure 2.2: **Framework of ABCpruner [1].** ABCpruner integrates the automatic searching algorithm (artificial bee colony [32]) to search for a better-pruned network.

2.1.4 Searching Algorithm

Searching algorithms aim to find or plan things more efficiently and therefore have been adopted in enormous fields. They are commonly used to solve optimization problems. Some of them are applied to deep learning since it is one of the optimization problems. In the pruning area, ABCpruner [1] is a structured filter pruning method that integrates the artificial bee colony (ABC) [32] searching algorithm in the pruning process. Different from other works, using the automatic search algorithm would reduce human experts' trail-and-error and rule-of-thumb settings.

We also use the ABC algorithm to find the best-pruned structure. Our work differs from ABCpruner lies in two aspects: ABCpruner has to define that at the most percentage of channels are preserved in each layer while our method doesn't. Our proposed method focuses on overall model architecture and hence provides higher flexibility. Another one is ABCpruner defines that at the most percentage of channels are preserved in each layer, which means it can't set the pruning rate beforehand. It can only calculate the number of parameters or computations when the searching process is completed. Thus, given a resource constraint, it can't

doi:10.6342/NTU202101554



accurately find the model satisfying the budget at a time. By contrast, we can set the resource budget in the beginning. Then, every time the searching process is finished, we can precisely generate a pruned model meeting the given resource constraint.

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Chapter 3

Proposed Method

This thesis proposes the Platform-aware Architecture Generator and Search (PAGS) algorithm for network pruning. Our algorithm comprises two parts; the architecture generator and the architecture search. To expand the search space for pruning candidates, we use encoder-decoder architecture to train our decoder, namely, the architecture generator. Given two inputs, latent code and latency constraint, the architecture generator will yield pruned network model meeting the latency constraint. Numerous pruned candidates are followed by the architecture search that will search for the best candidates from them. The framework of the proposed method is illustrated in Figure 3.1. We will describe the details of each part of our proposed method in this chapter.

doi:10.6342/NTU202101554

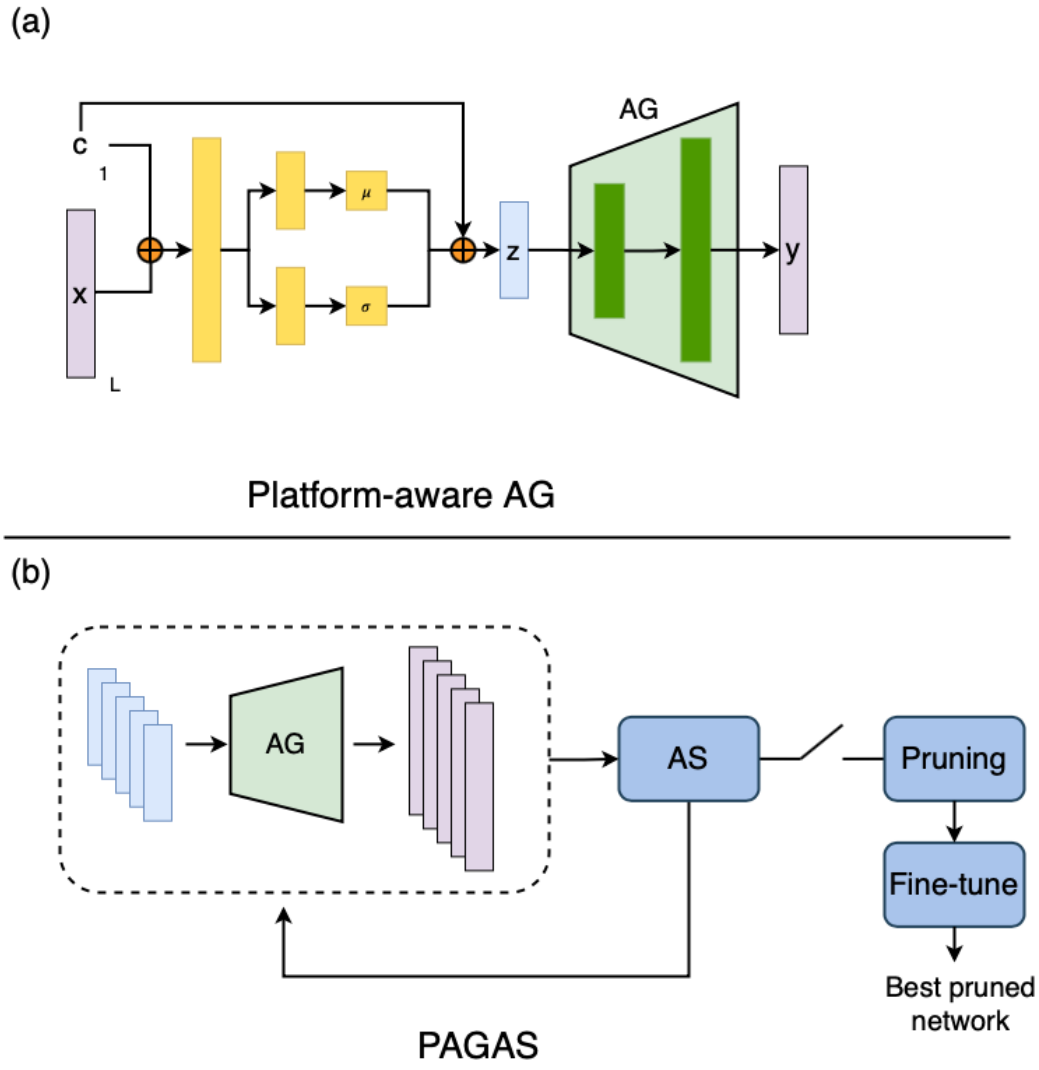


Figure 3.1: **(a) Training stage for platform-aware architecture generator. (b) Framework overview..** The framework incorporates AG and architecture searching (AS). When AS is completed, the searched best structure is the target of our pruning procedure. We use the L2-norm as filter importance to prune the pre-trained model with the best structure and fine-tune it to recover the performance.

doi:10.6342/NTU202101554

3.1 Preliminary

Given a L-layer model $\mathcal{N}$ with filter set $\mathcal{F} = \{f_1, f_2, ..., f_L\}$, where f_i is the filters of the i-th layer, and its network architecture set $\mathcal{A} = \{a_1, a_2, ..., a_L\}$, where a_i is the filter number of the i-th layer, and the training dataset $\mathcal{D}_{train}$. The objective of network training is to minimize the given loss function $\mathcal{L}(\mathcal{A}, \mathcal{F}; \mathcal{D}_{train})$. In filter pruning, given a pruned model $\mathcal{N}'$ with filter set $\mathcal{F}'$, the corresponding structure set $\mathcal{A}' = \{a'_1, a'_2, ..., a'_L, \}$, where $a'_i \leq a_i$ is the filter number of the pruned model in the i-th layer. We aim to find the architecture set $\mathcal{A}'$ that satisfies the given resource budget and find the best filter set $\mathcal{F}'$ such that the pruned model $\mathcal{N}'$ fine-tuned on $\mathcal{D}_{train}$ yields the best test accuracy on $\mathcal{D}_{test}$. We formulate it as follow:

$$\underset{\mathcal{F}'}{\operatorname{argmin}} \mathcal{L}(\mathcal{A}', \mathcal{F}'; \mathcal{D}_{train}) \quad s.t.\ \mathcal{R}(\mathcal{A}') \approxeq C \tag{3.1}$$

C is the given pruning constraint, and $\mathcal{R}(\mathcal{A})$ is the resource consumption of the model with the architecture of $\mathcal{A}$. In our pruning setting, C is the model inference latency measured on the target platform. In Eq. (3.1), it's almost impossible to find all pruned architectures meeting the constraint C since the search space is huge. To reduce the search space, among previous works, many of them would use an iterative shrinkage approach, i.e., at each iteration, they will evaluate the importance of the filters and remove a small part of the filters, which are relatively unimportant. Therefore, the architecture set of the model would shrink at each iteration until the resource consumption of the pruned network satisfies the constraint. The final pruned model is only one of architecture that meets the resource budget. These methods may lead to sub-optimal results among all architecture sets satisfying the constraint. Our proposed method is to address this problem and will be explained in detail in the following sections.

doi:10.6342/NTU202101554

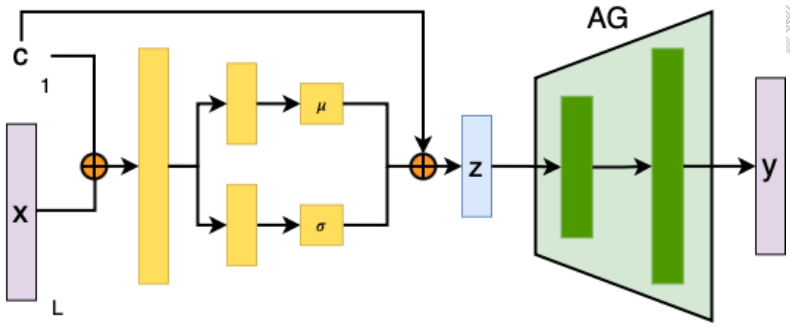


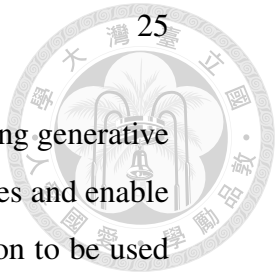
Figure 3.2: **Encoder-Decoder structure.** Given latency constraints and normalized architecture vectors as inputs, the decoder will reconstruct normalized architecture vectors as outputs at the training stage.

3.2 Platform-aware Architecture Generator

Given a target platform, like a mobile CPU, we aim at finding the best network structure at the same number of computations but with the lowest inference latency while maintaining the acceptable performance drop. In other words, our goal is to find the best-pruned model at the same latency constraint. Since enormous model structures meet the specific latency constraint and hence it's impractical to evaluate all of them. Furthermore, the true model architectures distribution given a specific constraint is unknown. Hence, we propose a platform-aware architecture generator to address these problems.

Inspired by CVAE [31], We aim to sample network architectures meeting the constraint from a known distribution such as normal distribution. Suppose there are many architecture and latency label pairs, we replace input variable and condition in CVAE with network architecture and the latency respectively, and train a platform-aware architecture generator (AG) . Our platform-aware AG can take latency constraint as the condition and sample latent codes from known latent

doi:10.6342/NTU202101554



space distribution to generative desired model architectures. Applying generative model to pruning problem can efficiently generate network structures and enable more pruning candidates to be selected. We first define the notation to be used in this paper. There are four variables in our encoder-decoder structure which is illustrated in Figure 3.2: input normalized architecture vectors $\mathbf{x}$, the corresponding latency label $\mathbf{c}$, latent variable $\mathbf{z}$ drawn from prior Gaussian distribution $p_\theta(\mathbf{z}|\mathbf{c})$ and reconstructed output variable $\mathbf{y}$, where the encoder is a recognition network with parameters ϕ and the decoder is a generative model with parameters θ. This encoder-decoder structure is suitable for the approximation of model architecture distribution since the true structure distribution is unknown. In the encoder-decoder structure, the likelihood can be written as follows:

$$\log p_\theta(\mathbf{x}|\mathbf{c}) = KL(q_\phi(\mathbf{z}|\mathbf{x},\mathbf{c}) \,||\, p_\theta(\mathbf{z}|\mathbf{x},\mathbf{c})) \; + \; \mathcal{L}(\mathbf{x},\mathbf{c};\theta,\phi) \tag{3.2}$$

The first RHS term is the KL divergence of the approximate from the true posterior. Since the KL-divergence is non-negative, the second RHS term $\mathcal{L}(\mathbf{x},\mathbf{c};\theta,\phi)$ is called the (variational) lower bound on the likelihood and is used as the objective function. It can be written as:

$$\mathcal{L}(\mathbf{x},\mathbf{c};\theta,\phi) = -KL(q_\phi(\mathbf{z}|\mathbf{x},\mathbf{c}) \,||\, p_\theta(\mathbf{z}|\mathbf{c})) \; + \; \mathbb{E}_{q_\phi(z|x,c)}[\log p_\theta(\mathbf{x}|\mathbf{z},\mathbf{c})] \tag{3.3}$$

where $\mathbf{z} = g_\phi(\mathbf{x},\mathbf{c},\epsilon), \epsilon \backsim \mathcal{N}(\mathbf{0},\mathbf{I})$. The distribution $q_\phi(\mathbf{z}|\mathbf{x},\mathbf{c})$ is reparameterized with a differentiable function $g_\phi(\cdot,\cdot,\cdot)$. This enables backpropagation through the Gaussian latent variables, which can be trained using stochastic gradient descent (SGD). The second RHS term $\mathbb{E}_{q_\phi(z|x,c)}[\log p_\theta(\mathbf{x}|\mathbf{z},\mathbf{c})]$ is the reconstruction error. The goal is to minimize the KL divergence of encoder $q_\phi(\mathbf{z}|\mathbf{x},\mathbf{c})$ from Gaussian distribution and minimize the reconstruction error between the input $\mathbf{x}$ and reconstructed output $\mathbf{y}$. Here we give the solution when both the prior $p_\theta(\mathbf{z}|\mathbf{c})$ = $\mathcal{N}(\mathbf{0},\mathbf{I})$ and the posterior approximation $q_\phi(\mathbf{z}|\mathbf{x},\mathbf{c})$ are Gaussian. Our objective function is written as:

$$\mathcal{L}(\mathbf{x},\mathbf{c},\mathbf{y};\theta,\phi) = \frac{1}{2}\sum_{j=1}^{J}(1+\log(\sigma_j^2)-\sigma_j^2-\mu_j^2) + \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i-\mathbf{x}_i)^2 \tag{3.4}$$

doi:10.6342/NTU202101554

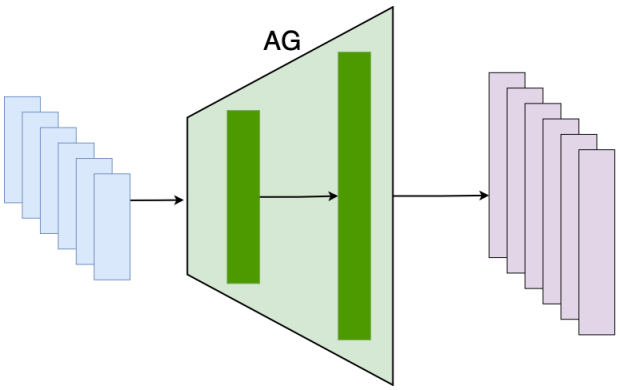


Figure 3.3: **Architecture Generator.** The generator can take inputs of assigned latency budget and latent vector and generate the normalized architecture vector meeting the constraint.

where approximate posterior, μ and σ, are variational mean and s.d. of the encoding. J is the dimensionality of z, μ_j and σ_j denote the j-th element of these vectors, and N is the number of samples. The training data pair ($\mathbf{x}$, $\mathbf{c}$) are collected by ourselves based on the target platform. Each training data $\mathbf{x}$ is a normalized architecture vector $\mathcal{V}' = \{v_1, v_2, ..., v_L\} = \{\frac{a_1}{a_1'}, \frac{a_2}{a_2'}, ..., \frac{a_L}{a_L'}\}$. Each dimension in $\mathcal{V}'$ is the ratio of the number of filters in the pruned network to that of the pretrained network. The corresponding network's average latency is measured on the platform for 50 times to reduce the variation of the measurements. Then based on the latency, a latency label $\mathbf{c}$ is assigned to $\mathbf{x}$ to form a training data pair ($\mathbf{x}$, $\mathbf{c}$). The recognition network with parameters ϕ is learned jointly with the generative model with parameters θ. Once the parameters are learned, generated model (the decoder network) is used as our platform-aware architecture generator to generate network architectures under the given resource budget.

Given a pre-trained model $\mathcal{M}$, its inference latency t on the target platform, and pre-given 9 latency constraint classes, i.e., $\{0.1t, 0.2t, ..., 0.9t\}$. Random sample sub-structures of $\mathcal{M}$ and measure their inference latency to form sub-structure and latency pairs $(\mathcal{M}', t')$. Then these pairs are picked based on the t' and allocated to its corresponding latency constraint class. For example, if a sub-model's latency is

doi:10.6342/NTU202101554

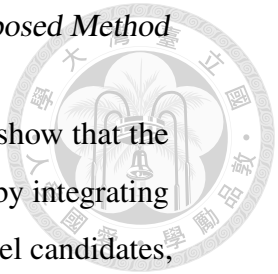
$0.52t$, it's assigned to a $0.5t$ class label. If the latency is $0.88t$, then it's assigned to $0.9t$ class label. After all data sample pairs are assigned, the dataset collection is completed. The dataset is composed of sub-structures and class label pairs. Next, the dataset is divided into the training set and testing set. We aim to train an architecture generator to yield model structures satisfying the given latency constraint. The training data and the corresponding label pairs are fed into the encoder at the training stage. The encoder will combine the reparameterized trick to map the architecture to a low dimension, latent space. The decoder will sample the latent variables to reconstruct the model architecture. By minimizing the KL divergence and the mean square error, when the training process is completed, only the decoder network will be used.

With empirical latency measurement and only a small number of training sample pairs, our platform-aware AG illustrated in Figure 3.3 can yield the model architecture meeting the latency budget without the deep knowledge of the target device. Moreover, the input samples are drawn from known latent space, so our method can consistently and efficiently generate various architectures that all meet the latency constraint. This allows us to expands the search space for the pruned candidates. Then, we will search for the best structure among the generated pruning candidates. The following section will further give the details about the searching algorithm.

3.3 Architecture Search

Given a set of model architecture meeting the latency budget, our goal is to find the best architecture and the corresponding filter set. For each candidate, which is a potential solution generated from platform-aware AG, We first turn the normalized architecture vector into the pruned network architecture. Then we use the L2-norm as the filter importance to prune the pre-trained model with the pruned architecture. To make the searching process more efficient, without fine-tuning, we apply fast

doi:10.6342/NTU202101554



evaluation to determine the fitness of each pruning candidate. We show that the searching of the best-pruned model can be optimized automatically by integrating the searching algorithm ABC [32] to search from those pruned model candidates, shown in Algorithm 1. The details of the core parts, constructing three types of "Bees", are elaborated in the following:

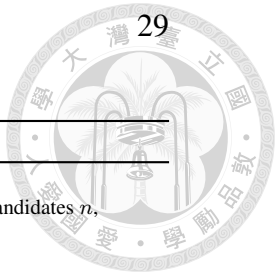
Step 1: Employed Bee. The employed bee aims to exploit architecture candidates. At each cycle i, given the N candidate architectures under the latency constraint, $\{\mathcal{A}'_1, \mathcal{A}'_2, ..., \mathcal{A}'_N\}^{(i)}$, the platform-aware AG will generate a new different set that all candidates in the new set meet the latency constraint, called $\{\mathcal{S}'_1, \mathcal{S}'_2, ..., \mathcal{S}'_N\}^{(i)}$, to compete with the existing candidates, where $\mathcal{A}'^{(i)}_j$ will compete with $\mathcal{S}'^{(i)}_j$. The employed bee will decide whether to keep the original candidate or to replace it with newly generated architecture based on the fitness score shown below:

$$fitness_{\mathcal{A}'^{(i)}_j} = TestAcc(FastEval(\mathcal{A}'^{(i)}_j, \mathcal{F}'^{(i)}_j; \mathcal{D}_{train}); \mathcal{D}_{test}) \quad \forall\, 1 \leq j \leq N \tag{3.5}$$

where TestAcc is the test accuracy of the pruned model on the test dataset ($\mathcal{D}_{test}$). To recover the accuracy drop of the pruned network caused by the pruning process, the fine-tuning approach is usually adopted. However, fine-tuning takes a great amount of time in the searching process, which is the bottleneck of the overall algorithm. To address this problem, rather than fine-tuning each candidate for several epochs as common pruning methods, we applied FastEval [33], which is a fast evaluation approach for the pruned model. Therefore, it's more efficient to evaluate the fitness score of each candidate. If the fitness score of $\mathcal{S}'^{(i)}_j$ is higher than $\mathcal{A}'^{(i)}_j$, $\mathcal{S}'^{(i)}_j$ will replace $\mathcal{A}'^{(i)}_j$. Otherwise, $\mathcal{A}'^{(i)}_j$ remains as a candidate. Then the updated candidate set will be passed to the next stage, onlooker bee.

Step 2: Onlooker Bee. The onlooker bees gather the information from employee bees. Then they evaluate and select some promising candidates to be further competed based on the normalized fitness. First, given the updated $\{\mathcal{A}'_1, \mathcal{A}'_2, ..., \mathcal{A}'_n\}^{(i)}$,

doi:10.6342/NTU202101554



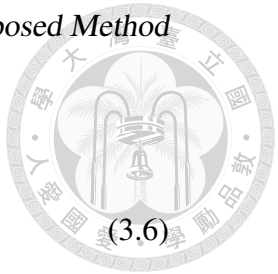
Algorithm 1: Architecture Search

Input: Pre-trained network $\mathcal{N}$, latency constraints l, cycle C, max time T, number of pruned candidates n, counter $\{t_j\}_{j=1}^{n}$

Output: Best pruned network architecture $\mathcal{A}'^{*}$

Initialize n pruned architectures $\{\mathcal{A}'_j\}_{j=1}^{n}$ from $\mathcal{N}$
Set counter $\{t_j\}_{j=1}^{n}$=0
for $i \leftarrow 1$ to C **do**
  for $j \leftarrow 1$ to n **do**
    Generate a new pruned architecture $\mathcal{S}'_j$ from the generator
    Calculate the fitness of $\mathcal{A}'_j$ and $\mathcal{S}'_j$ with (3.5)
    if $fitness_{\mathcal{S}'_j} > fitness_{\mathcal{A}'_j}$ **then**
      $\mathcal{A}'_j = \mathcal{S}'_j$
      $fitness_{\mathcal{A}'_j} = fitness_{\mathcal{S}'_j}$
      $t_j = 0$
    else
      $t_j = t_j + 1$
    end if
  end for
  for $j \leftarrow 1$ to n **do**
    Sample $\epsilon_j \in (0, 1)$
    Calculate of P_j with (3.6)
    if $\epsilon_j < P_j$ **then**
      Generate a new pruned architecture $\mathcal{S}'_j$ from the generator
      Calculate the fitness of $\mathcal{A}'_j$ and $\mathcal{S}'_j$ with (3.5)
      if $fitness_{\mathcal{S}'_j} > fitness_{\mathcal{A}'_j}$ **then**
        $\mathcal{A}'_j = \mathcal{S}'_j$
        $fitness_{\mathcal{A}'_j} = fitness_{\mathcal{S}'_j}$
        $t_j = 0$
      else
        $t_j = t_j + 1$
      end if
    end if
  end for
  for $j \leftarrow 1$ to n **do**
    if $t_j > T$ **then**
      generate a new pruned architecture $\mathcal{S}'_j$ from the generator
    end if
  end for
end for
$\mathcal{A}'^{*} = \underset{\mathcal{A}'}{\operatorname{argmax}}\, TestAcc(\mathcal{N}'(\mathcal{A}', \mathcal{F}'; \mathcal{D}_{train}); \mathcal{D}_{test})$

doi:10.6342/NTU202101554



normalized fitness score is applied as in Eq. 3.6.

$$P_j = \frac{fitness_{\mathcal{A}'_j}}{max(fitness_{\mathcal{A}'_j})} \quad \forall\, 1 \leq j \leq N \tag{3.6}$$

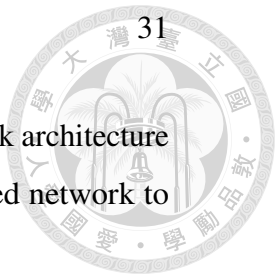
Then, to ensure the robustness of the candidate architecture with a high fitness score, we sample a random number $\epsilon \in (0, 1]$ as a threshold. If a pruned network candidate's P_j is higher than ϵ, our platform-aware AG will generate a new network architecture candidate meeting the given latency constraint to compete with the existing one via Eq. 3.5. That is, if the high fitness score of the architecture is not robust enough, it may have a chance to be replaced by a newly generated architecture candidate. Then the updated candidate set will be passed to the scout bee, the last stage of the searching.

Step 3: Scout Bee. To avoid trapping in the local optimal, there is a counter for each pruned network candidate. If a candidate is not updated in this searching round, its corresponding counter will be increased by one. And if a candidate remains unchanged for many cycles, the scout bee will generate a new model architecture to replace it.

At each cycle, the three types of bees mentioned above will take turns to exploit, gather, avoid trapping in the local optimal, respectively. The searching algorithm will be conducted for several cycles. When the searching process is finished, we have the best-pruned network architecture meeting the given latency constraint.

Unlike ABCpruner [1], rather than fine-tuning several epochs to calculate the fitness score for each candidate, we apply fast evaluation [33] for each candidate by only updating mean and variance in the batch-norm layer in the pruned network for several iterations. This approach enables the searching algorithm to be more efficient and has a similar effect compared with the fine-tuning method. Moreover, we don't have to manually set the maximum pruning ratio for each layer as the ABCpruner. Once the searching process is completed, the searched best structure is the target of the pruning procedure. That is, we find the model architecture meets the latency constraint with the best accuracy. We use the L2-norm as the

doi:10.6342/NTU202101554



filter importance and prune the pretrained model with the best network architecture to produce the pruned network. Lastly, we will fine-tune the pruned network to recover the performance.

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Chapter 4

Experiments

In this section, the implementation details are provided, and several experiments are conducted to show the effectiveness of our proposed method.

4.1 Architecture Generator

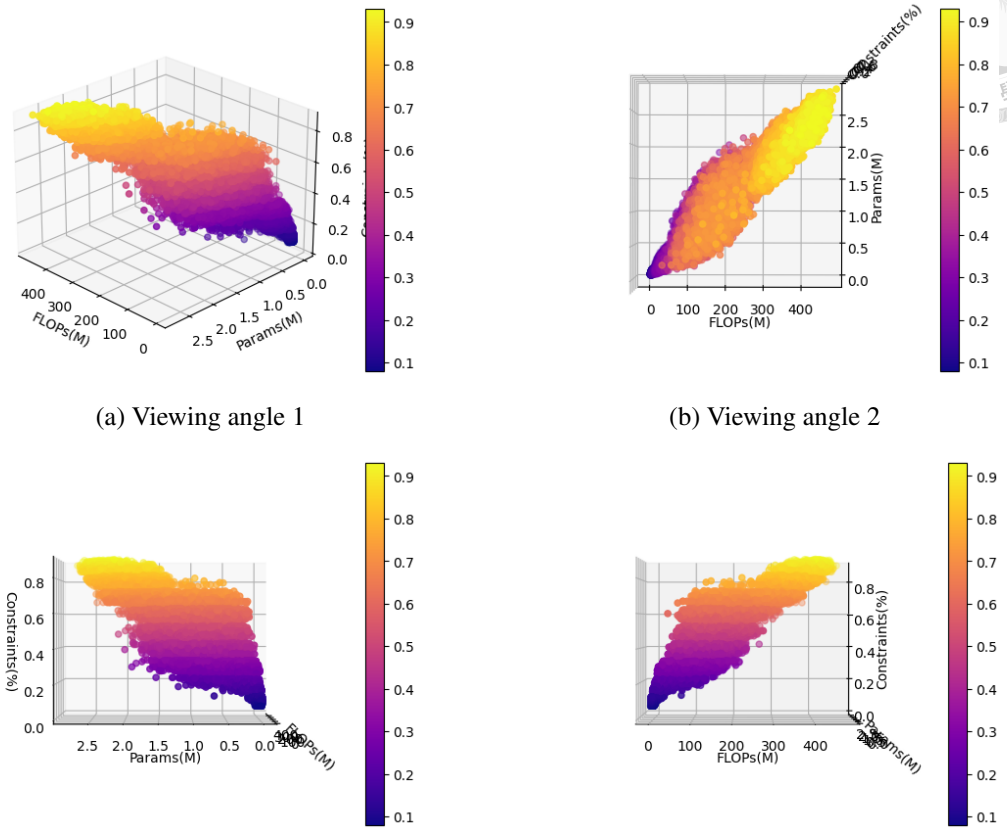
4.1.1 Implementation Details

We sample enormous pruned model architectures and measure them on desktop GPU to collect the training and test dataset. However, how many samples are sufficient for a class is a tradeoff. A larger dataset could give the architecture generator better performance and generalization, but it takes a great amount of time for inference latency measurement. Therefore, we test two settings: 10,000 samples per class and 1,000 samples per class. If the smaller one has a similar performance to the larger one, 1,000 samples per class will be used. Otherwise, 10,000 data samples per class will be adopted.

Dataset

We collect 10,000 samples for 9 latency constraint classes, 90,000 data samples in total. For each class, nine-tenths of samples is for trainset; one-tenth is for testset.

doi:10.6342/NTU202101554



(a) Viewing angle 1

(b) Viewing angle 2

(c) Viewing angle 3

(d) Viewing angle 4

Figure 4.1: **Large data sample distribution.** There are 3 dimensions in these figures: FLOPS, parameters, and constraints. Color of samples represent which constraint class they belong to.

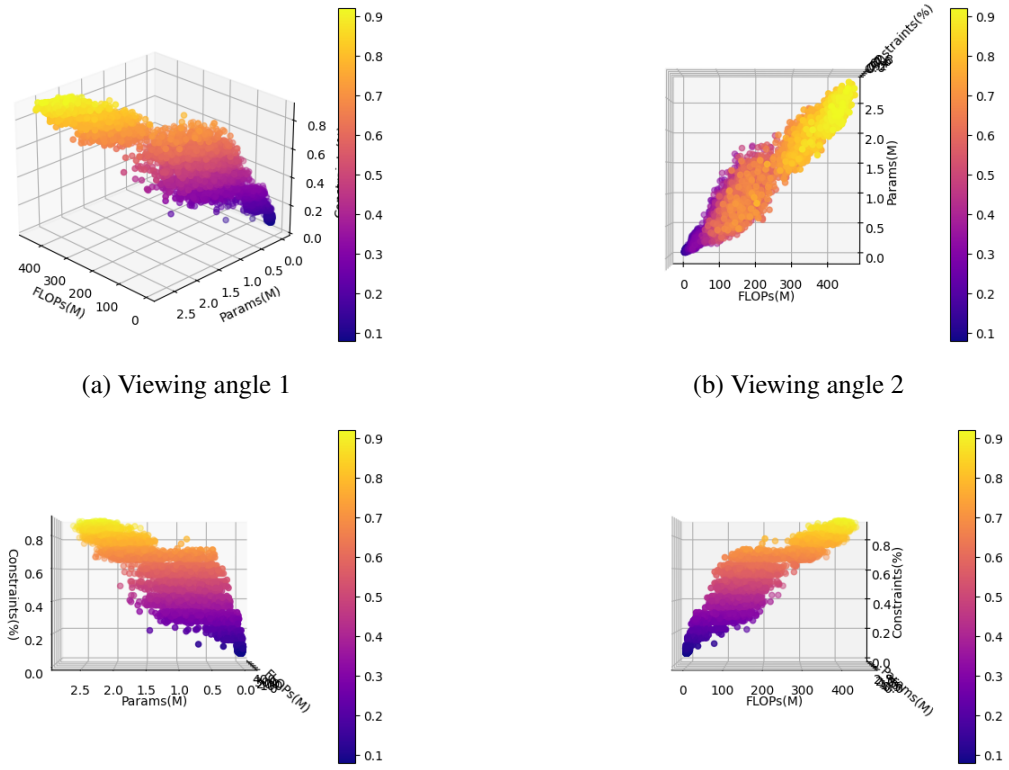
Large data samples shown in Figure 4.1 is the 90,000 data samples distribution, and Figure 4.2 is the smaller one with 9,000 data samples distribution. The overall small data distribution is similar to that of the larger one, with slightly sparse, so does each constraint class. Therefore, to efficiently collect the data samples on the different target platforms, we decide to use 1,000 data samples per class.

Latent Space

At training stage, given a L-layer pretrained model architecture $\mathcal{A}$, turn each data sample, i.e., the pruned model architecture $\mathcal{A}'$, into a normalized feature

doi:10.6342/NTU202101554

vector $\mathcal{V}' = \{v_1, v_2, ..., v_L\} = \{\frac{a_1}{a_1'}, \frac{a_2}{a_2'}, ..., \frac{a_L}{a_L'}\}$. we use encoder to encode the normalized vector to a latent space, a low dimension feature. Then the reconstruct The dimension of the latent space could affect the performance. Thus, we test different dimension of latent space, the result is shown in Figure 4.3. In each constraint class, dimension difference has little impact on the final performance. The generated samples with different latent dimension are all close to the desired latency constraint.



(a) Viewing angle 1

(b) Viewing angle 2

(c) Viewing angle 3

(d) Viewing angle 4

Figure 4.2: **Small data sample distribution.** There are 3 dimensions in these figures: FLOPS, parameters, and constraints. Color of samples represent which constraint class they belong to.

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

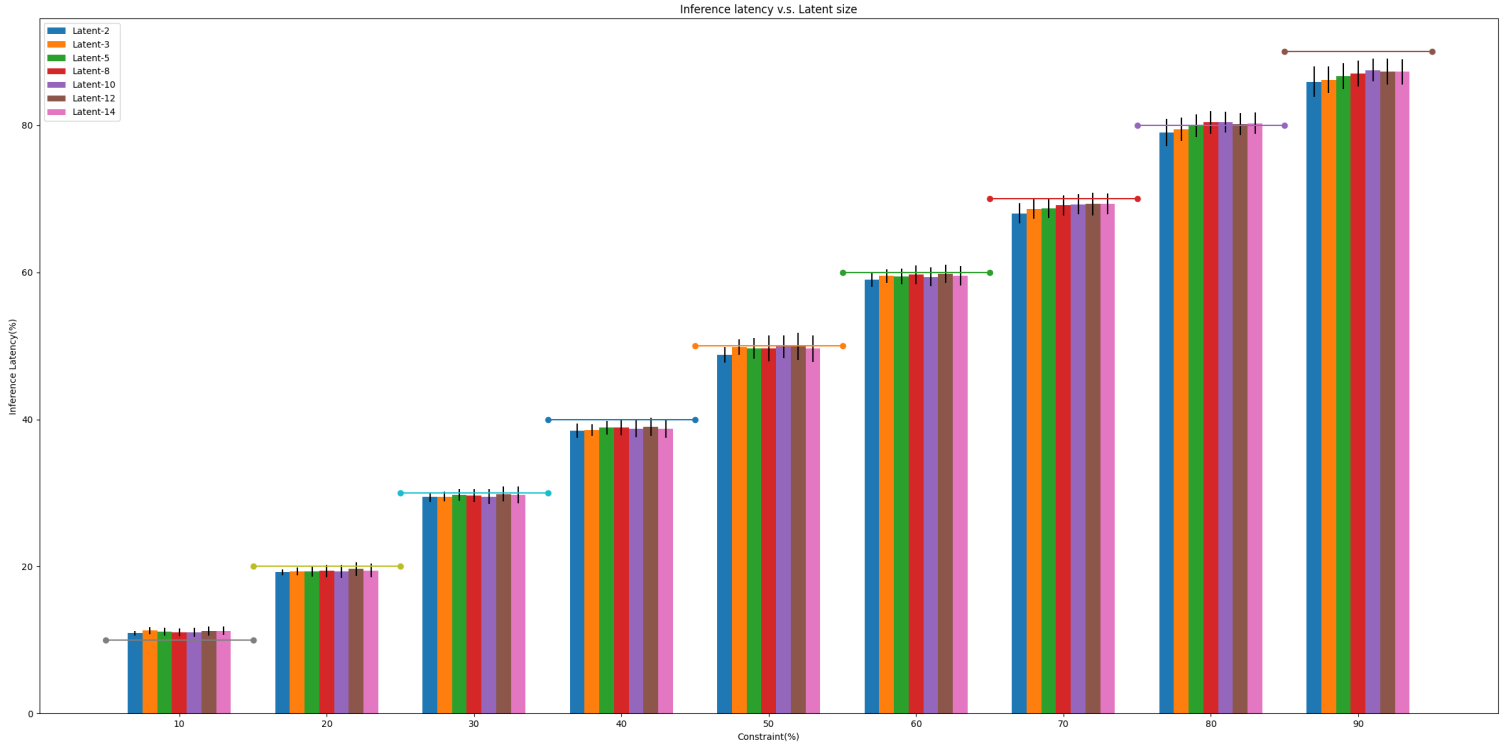


Figure 4.3: **Latent dimension evaluation.** Color of bars represents the latent size, the horizontal line stands for the exact inference latency percentage for each constraint. The closer distance between the line and bar is, the more accurate result is.

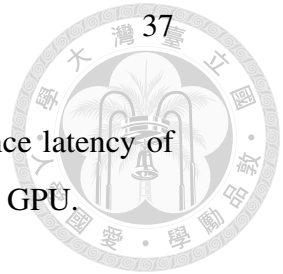


Table 4.1: **Architecture generator evaluation.** Measured inference latency of pruned models at each constraint are measured 50 times on desktop GPU.

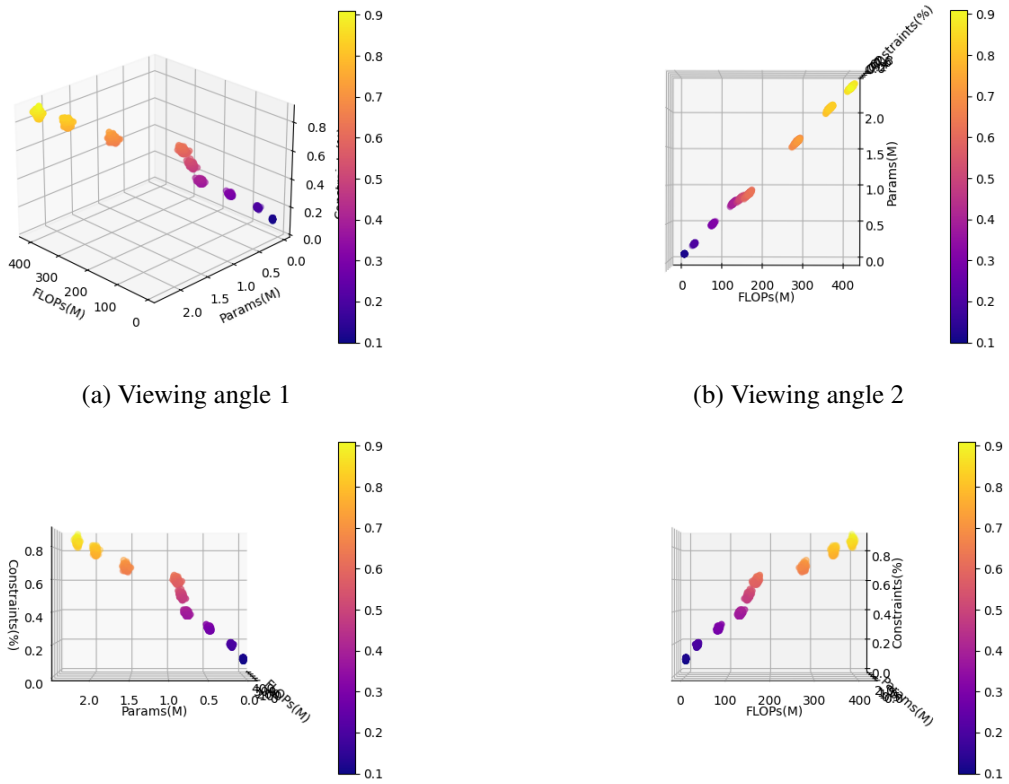
Model	Constraint (%)	Measured lat. Mean (SD)
MobileNet	10	10.90 (±0.54)
	20	19.90 (±0.83)
	30	29.60 (±0.49)
	40	39.00 (±1.10)
	50	50.50 (±1.20)
	60	60.30 (±0.90)
	70	69.50 (±1.02)
	80	81.20 (±1.25)
	90	87.50 (±1.20)

4.1.2 Evaluation

After the training process is completed, 100 sample latent vectors are randomly drawn from Gaussian distribution for each latency constraint class at the generating stage. Then the platform-aware architecture generator takes these latent vectors, concatenates the given constraint label as the inputs, and generates corresponding pruned model normalized vectors. We visualize the generated result of MobileNet-v1 in Figure 4.4. Color represents the constraint class. Each color cluster consists of 100 samples, and the distribution of each cluster is similar to the normal distribution. These vectors are reconstructed to the pruned model architecture and followed by the one-shot pruning process to prune the pretrained model according to the pruned model architectures. Then all pruned models are measured on the platform to obtain the inference latency. We kill background processes to ensure lower latency fluctuation during the measurements and measure each model 50 times to get the average inference latency. Table 4.1 shows the effectiveness of the architecture generator. For each constraint, the generator can yield pruned model structures whose latency is close to the resource budget. Furthermore, the standard deviation

doi:10.6342/NTU202101554

is small, which shows that the generator can generate accurate output consistently with low deviation given the latency constraint.



(a) Viewing angle 1

(b) Viewing angle 2

(c) Viewing angle 3

(d) Viewing angle 4

Figure 4.4: **Generated sample distribution.** There are 3 dimensions in these figures: FLOPS, parameters, and constraints. Color of samples represent which constraint class they belong to.

4.2 Experiments

4.2.1 Implementation Details

We conduct some experiments for well-known convolutional neural networks on both CIFAR-10 [34] and ImageNet ILSVRC-12 [35]. Furthermore, we benchmark our method against the state-of-the-art network pruning method.

doi:10.6342/NTU202101554

Dataset

Our method is evaluated on two datasets, the CIFAR-10 [34] and ImageNet ILSVRC-12 [35]. The CIFAR-10 dataset comprises 50,000 training images and 10,000 test images in 10 classes, while the ImageNet dataset contains a million training images and 50,000 test images in 1000 classes. To compare with other methods, We use the standard process of data augmentation in both datasets. For the training set in CIFAR-10, the process includes re-sizing images to 256×256, randomly cropping a 224×224, then normalizing with the mean and standard deviation of the dataset; for testing data of CIFAR-10, we re-size images to 256×256, crop them into a 224×224 patch then apply normalization. For the training set in ImageNet, the process contains re-sizing images to 256×256, randomly cropping a 224×224 patch, randomly flipping horizontally, and normalizing them with the mean and variance of ImageNet. For the testing set in ImageNet, the process is the same as that of the test data in CIFAR-10.

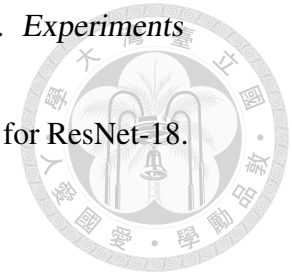
Model

We conduct some experiments on two well-known networks: ResNet-18 [4] and MobileNet [36]. ResNet-18 possesses 11 million parameters and 1.8 billion FLOPs, and it is famous for its skip connection and therefore improves the performance on different tasks. Nonetheless, the design may not be executed efficiently on the platform due to its large memory footprint. MobileNet has 3.2 million parameters and 567 million FLOPs and is known for its depthwise separable design to reduce computations.

Fine-tuning Strategy

On CIFAR-10, we fine-tune the pruned network for 200 epochs with the learning rate of 0.1 decayed by 10 every 50 epoch. While on ImageNet, the pruned model will be fine-tuned for 50 epochs with the learning rate of 0.1 decayed by 10 at epoch 35 and 45. Optimizer for MobilenNet is Stochastic Gradient Decent(SGD) and

doi:10.6342/NTU202101554



Adam [37] for ResNet-18. Batch-size is 256 for MobileNet and 128 for ResNet-18.

Platform inference and Latency measurement

We use two low-power platforms: Qualcomm Snapdragon Neural Processing Engine (SNPE) for inference on a mobile CPU and Nvidia Jetson TX2 for inference on embedded GPU. For experiments on the CPU, the latency is measured with the SNPE benchmark tool on Google Pixel 3. As for experiments on GPU, the latency is defined as the execution time between the process start time and end until all CUDA cores are synchronized. Each inference latency number is the average of 50 measurements.

4.2.2 Results on Mobile CPU

For Google Pixel 3 mobile CPU, we apply our method to MobileNet and ResNet-18 networks on both small and large datasets. The experiment results are reported in Table 4.2 and Table 4.3

Results on CIFAR-10. In Table 4.2, we compare our proposed method with the state-of-the-art platform-aware pruning method. For MobileNet on CIFAR-10, at the same latency constraint, our method outperforms the state-of-the-art. At similar inference latency left measured on the deployed CPU, we can improve more than 1 percent or even 2.25 percent with lower inference latency. As for ResNet-18, our performance has slight improvements. The reason why the improvement of ResNet-18 is much less than that of MobileNet is that ResNet-18 has more complex network designs such as skip connections. This skip connection will affect the execution of the deployed platform, which affects the inference latency. This leads to that there are not various model architectures to choose from at the same constraint. Therefore, the final searched best model structure is slightly better.

doi:10.6342/NTU202101554

Table 4.2: **Comparison to state-of-the-arts [2] on CIFAR-10 on CPU.** Inference latency of pruned models is measured on Google Pixel 3 mobile CPU. Lat. left: measured latency of the pruned model over that of the pretrained model. P. Top-1: Top-1 test accuracy. Top-1↓: Top-1 accuracy drop. Improv.: improvement between the NetAdapt and our work.

Model	Orig. Top-1 (%)	Method	Lat. left (%)	P. Top-1 (%)	Top-1↓ (%)	Improv. (%)
MobileNet	94.56	NetAdapt-90%	90	92.16	2.4	
		Ours ($lat_{.90}$)	84	**94.41**	**0.15**	**2.25**
		NetAdapt-80%	80	91.92	2.64	
		Ours ($lat_{.80}$)	75	**94.15**	**0.41**	**2.23**
		NetAdapt-70%	75	92.34	2.22	
		Ours ($lat_{.70}$)	64	**93.85**	**0.71**	**1.51**
		NetAdapt-60%	56	92.49	2.07	
		Ours ($lat_{.60}$)	55	**93.67**	**0.89**	**1.18**
		NetAdapt-50%	45	91.35	3.21	
		Ours ($lat_{.50}$)	49	**93.29**	**1.27**	**1.94**
		NetAdapt-40%	37	91.78	2.78	
		Ours ($lat_{.40}$)	41	**93.04**	**1.52**	**1.26**
		NetAdapt-30%	28	90.62	3.94	
		Ours ($lat_{.30}$)	30	**91.97**	**2.59**	**1.35**
ResNet-18	94.96	NetAdapt-90%	90	92.54	2.42	
		Ours ($lat_{.90}$)	94	**92.82**	**2.14**	**0.28**
		NetAdapt-80%	84	92.13	2.83	
		Ours ($lat_{.80}$)	83	**92.69**	**2.27**	**0.56**
		NetAdapt-70%	78	92.01	2.95	
		Ours ($lat_{.70}$)	73	**92.2**	**2.76**	**0.19**
		NetAdapt-60%	59	92.07	2.89	
		Ours ($lat_{.60}$)	65	**92.09**	2.87	**0.03**
		NetAdapt-50%	46	**91.83**	**3.13**	
		Ours ($lat_{.50}$)	53	91.67	3.29	-0.16
		NetAdapt-40%	43	90.98	3.98	
		Ours ($lat_{.40}$)	41	**91.32**	**3.64**	**0.34**

Results on ImageNet. We also perform some experiments on a large-scale dataset like ImageNet. The results are reported in Table 4.3. For MobileNet on ImageNet, as the latency constraint becomes harder to achieve, our method

doi:10.6342/NTU202101554

Table 4.3: **Comparison to state-of-the-arts [2] on ImageNet on CPU.** Inference latency of pruned models are measured on Google Pixel 3 mobile CPU

Model	Orig. Top-1 (%)	Method	Lat. left (%)	P. Top-1 (%)	Top-1↓ (%)	Improv. (%)
MobileNet	69.35	NetAdapt-90%	87	67.11	2.24	
		Ours ($lat_{.90}$)	84	**67.33**	**2.02**	**0.22**
		NetAdapt-80%	80	66.02	3.33	
		Ours ($lat_{.80}$)	76	**66.99**	**2.36**	**0.97**
		NetAdapt-70%	69	65.16	4.19	
		Ours ($lat_{.70}$)	64	**66.7**	**2.65**	**1.54**
		NetAdapt-60%	52	**62.31**	**7.04**	
		Ours ($lat_{.60}$)	56	61.13	8.23	-1.19
		NetAdapt-50%	42	59.22	10.13	
		Ours ($lat_{.50}$)	49	**61.1**	**8.25**	**1.88**
		NetAdapt-40%	32	55.76	13.59	
		Ours ($lat_{.40}$)	42	**59.15**	**10.2**	**3.39**
		NetAdapt-30%	25	48.33	21.02	
		Ours ($lat_{.30}$)	30	**53.84**	**15.51**	**5.51**
ResNet-18	69.76	NetAdapt-90%	90	67.15	2.61	
		Ours ($lat_{.90}$)	92	**68.28**	**1.48**	**1.13**
		NetAdapt-80%	83	66.76	3	
		Ours ($lat_{.80}$)	83	**67.68**	**2.08**	**0.92**
		NetAdapt-70%	68	65.85	3.91	
		Ours ($lat_{.70}$)	73	**67.57**	**2.19**	**1.72**
		NetAdapt-60%	56	65.43	4.33	
		Ours ($lat_{.60}$)	65	**67.65**	**2.11**	**2.22**
		NetAdapt-50%	45	65.64	4.12	
		Ours ($lat_{.50}$)	53	**66.06**	**3.7**	**0.42**

achieves more improvement up to 5.51 percent at 30% latency constraint. Since ImageNet is a complex dataset, compact pruned models usually suffer from large accuracy drops. However, the pruned model searched by our method can drop less. ResNet-18 has similar results; we can achieve less performance drop and improve more than the state-of-the-art at the same resource constraint.

From Table 4.2 and Table 4.3, our method can yield compact pruned models and also satisfy the given constraints with acceptable test accuracy drop against the

doi:10.6342/NTU202101554

pretrained model. Moreover, our method can consistently search for the best model structure from any given resource budget for different networks and datasets.

4.2.3 Results on Mobile GPU

We apply our method to MobileNet and ResNet-18 networks on both small and large datasets for Nvidia Jetson mobile GPU. The experiment results are reported in Table 4.4 and Table 4.5

Table 4.4: **Comparison to state-of-the-arts [2] on CIFAR-10 on GPU.** Inference latency of pruned models are measured on Nvidia TX2

Model	Orig. Top-1 (%)	Method	Lat. left (%)	P. Top-1 (%)	Top-1↓ (%)	Improv. (%)
MobileNet	94.56	NetAdapt-90%	92	91.25	3.31	
		Ours ($lat._{90}$)	83	**94.4**	**0.16**	**3.15**
		NetAdapt-80%	80	91.58	2.98	
		Ours ($lat._{80}$)	80	**94.07**	**0.49**	**2.49**
		NetAdapt-70%	69	91.33	3.23	
		Ours ($lat._{70}$)	70	**93.47**	**1.09**	**2.14**
		NetAdapt-60%	63	91.23	3.33	
		Ours ($lat._{60}$)	61	**93.39**	**1.17**	**2.16**
		NetAdapt-50%	47	90.57	3.99	
		Ours ($lat._{50}$)	50	**93.25**	**1.31**	**2.68**
		NetAdapt-40%	41	90.28	4.28	
		Ours ($lat._{40}$)	42	**92.61**	**1.95**	**2.33**
ResNet-18	94.96	NetAdapt-90%	86	92.33	2.63	
		Ours ($lat._{90}$)	86	**94.4**	**0.56**	**2.07**
		NetAdapt-80%	74	92.81	2.15	
		Ours ($lat._{80}$)	81	**94.28**	**0.68**	**1.47**
		NetAdapt-70%	67	92.76	2.2	
		Ours ($lat._{70}$)	66	**94.23**	**0.73**	**1.47**
		NetAdapt-60%	53	92.7	2.26	
		Ours ($lat._{60}$)	60	**94.15**	**0.81**	**1.45**
		NetAdapt-50%	52	94.87	2.09	
		Ours ($lat._{50}$)	53	**94.13**	**0.83**	**1.26**

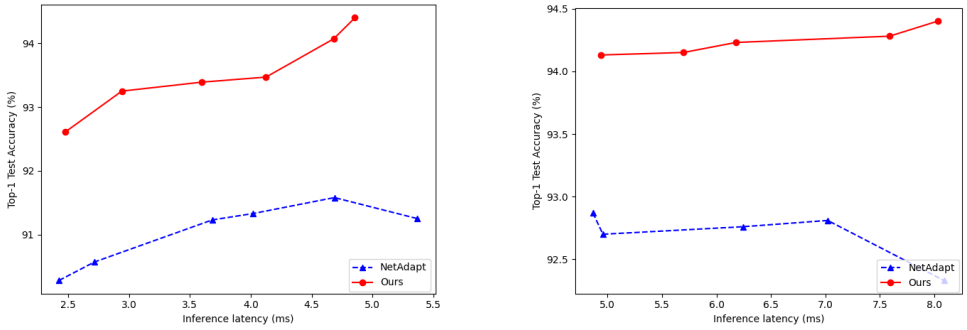
doi:10.6342/NTU202101554

Table 4.5: **Comparison to state-of-the-arts [2] on ImageNet on GPU.** Inference latency of pruned models are measured on Nvidia TX2

Model	Orig. Top-1 (%)	Method	Lat. left (%)	P. Top-1 (%)	Top-1↓ (%)	Improv. (%)
MobileNet	69.35	NetAdapt-90%	89	66.7	2.65	
		Ours ($lat._{90}$)	87	**66.93**	**2.42**	**0.23**
		NetAdapt-80%	83	66.16	3.19	
		Ours ($lat._{80}$)	80	**66.77**	**2.58**	**0.61**
		NetAdapt-70%	69	65.6	3.75	
		Ours ($lat._{70}$)	76	**65.83**	**3.52**	**0.23**
		NetAdapt-60%	58	58.39	10.96	
		Ours ($lat._{60}$)	63	**61.81**	**7.54**	**3.42**
		NetAdapt-50%	49	55.94	13.41	
		Ours ($lat._{50}$)	52	**59.24**	**10.11**	**3.3**
ResNet-18	69.76	NetAdapt-90%	85	66.9	2.86	
		Ours ($lat._{90}$)	83	**67.97**	**1.79**	**1.07**
		NetAdapt-80%	79	66.59	3.17	
		Ours ($lat._{80}$)	74	**67.72**	**2.04**	**1.13**
		NetAdapt-70%	68	65.8	3.96	
		Ours ($lat._{70}$)	70	**67.43**	**2.33**	**1.63**
		NetAdapt-60%	64	65.02	4.74	
		Ours ($lat._{60}$)	58	**67.21**	**2.55**	**2.19**

Results on CIFAR-10. In Table 4.4 and Figure 4.5, we compare our proposed method with the state-of-the-art platform-aware pruning method. For MobileNet on CIFAR-10, at the same latency constraint, our method outperforms the state-of-the-art. At similar inference latency left measured on the deployed CPU, we can improve more than 1 percent or even 2.25 percent with lower inference latency. As for ResNet-18, our performance has slight improvements. The reason why the improvement of ResNet-18 is much less than that of MobileNet is that ResNet-18 has more complex network designs such as skip connections. This skip connection will affect the execution of the deployed platform, which affects the inference latency. This leads to that there are not various model architectures to choose from at the same constraint. Therefore, the final searched best model structure is slightly better.

doi:10.6342/NTU202101554



(a) MobileNet (b) ResNet-18

Figure 4.5: **Comparison with state-of-the-arts on CIFAR-10.** This figure compares our method with NetAdapt[2] at the same latency constraint on Nvidia TX2. At the same constraint, our method has lower inference latency and higher test accuracy.

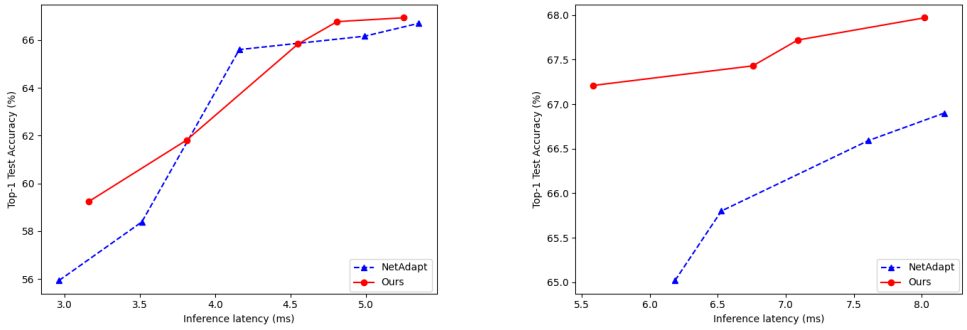
Results on ImageNet. We also perform some experiments on a large-scale dataset like ImageNet. The results are reported in Table 4.5 and Figure 4.6. For MobileNet on ImageNet, as the latency constraint becomes harder to achieve, our method achieves more improvement up to 3.3 percent at 50% latency constraint. Since ImageNet is a complex dataset, compact pruned models usually suffer from large accuracy drops. However, the pruned model searched by our method can drop less. ResNet-18 has similar results; we can achieve less performance drop and improve more than the state-of-the-art at the same resource constraint.

From Table 4.4 and Table 4.5, our method can yield compact pruned models and satisfy the given constraints with acceptable test accuracy drop against the pretrained model. Moreover, our method can consistently search for the best model structure from any given resource budget for different networks and datasets.

4.2.4 Comparison with Traditional Method

Previous works focus on reducing the total number of parameters and the number of computations. However, they are indirect performance metrics since each deployed

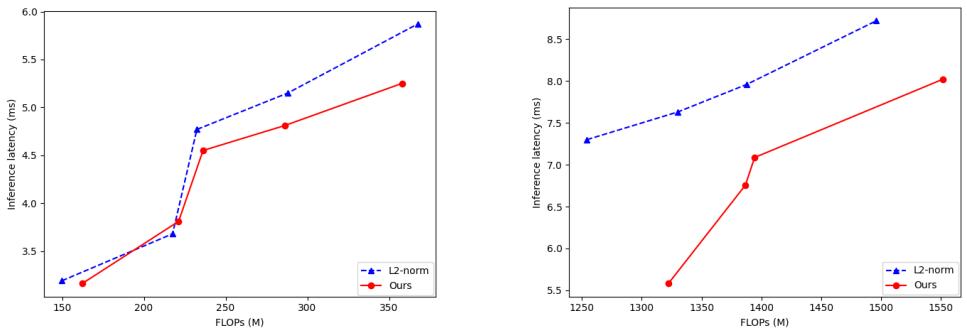
doi:10.6342/NTU202101554



(a) MobileNet (b) ResNet-18

Figure 4.6: **Comparison with state-of-the-arts on ImageNet.** This figure compares our method with NetAdapt[2] at the same latency constraint on Nvidia TX2. At the same constraint, our method has lower inference latency and higher test accuracy.

platform has its own characteristics. Reduction of 50% number of computations can't guarantee that the inference latency is 50% lower. We perform some experiments to show that our method can yield pruned models with faster inference and higher performance than non-platform-aware pruning. The experiments are



(a) MobileNet (b) ResNet-18

Figure 4.7: **Comparison with traditional pruning on ImageNet.** This figure compares our method with l2-norm magnitude pruning at the same FLOPs constraint on Nvidia TX2. At the same constraint, our method has lower inference latency and higher test accuracy.

doi:10.6342/NTU202101554

Table 4.6: **Comparison to non-platform-aware pruning [3] on ImageNet.** Inference latency of pruned models are measured on Nvidia TX2

Model	Method	Latency (ms)	FLOPs left (%)	Param. left (%)	P. Top-1 (%)	Top-1↓ (%)
MobileNet	Magnitude Pruning-90%	5.87	60.86	51.72	66.51	2.84
	Ours ($lat_{.90}$)	5.25	62.91	80.2	**66.93**	**2.42**
	Magnitude Pruning-80%	5.15	47.47	28	65.22	4.13
	Ours ($lat_{.80}$)	4.81	50.31	64.53	**66.77**	**2.58**
	Magnitude Pruning-70%	4.77	40.87	31.78	63.76	5.59
	Ours ($lat_{.70}$)	4.55	41.53	50.54	**65.83**	**3.52**
	Magnitude Pruning-60%	3.68	38.26	25.69	61.79	7.56
	Ours ($lat_{.60}$)	3.81	38.91	35.03	**61.81**	**7.54**
	Magnitude Pruning-50%	3.19	26.36	18.47	59.05	10.3
	Ours ($lat_{.50}$)	3.16	28.53	26.97	**59.24**	**10.11**
ResNet-18	Magnitude Pruning-90%	8.72	82.48	95.77	66.56	3.2
	Ours ($lat_{.90}$)	8.02	85.53	86.03	**67.97**	**1.79**
	Magnitude Pruning-80%	7.96	76.5	94.41	66.41	3.35
	Ours ($lat_{.80}$)	7.09	76.86	84.29	**67.72**	**2.04**
	Magnitude Pruning-70%	7.63	73.31	91.53	66.08	3.68
	Ours ($lat_{.70}$)	6.75	76.44	78.67	**67.43**	**2.33**
	Magnitude Pruning-60%	7.3	69.13	78.83	65.65	4.11
	Ours ($lat_{.60}$)	5.58	72.89	73.93	**67.21**	**2.55**

reported in Table 4.6 and Figure 4.7. The experiments are conducted on ImgeNet since it's a more complicated dataset. For each best model we search in Table 4.5, we calculate the total number of parameters and computations of it, and then we apply L2-norm magnitude pruning to prune the pretrained network until the FLOPs is close to the best model. From the table, Both MobileNet and ResNet-18 have similar results: compared with non-platform-aware magnitude L2-norm pruning. At the same or slightly fewer FLOPs, our method has lower inference latency and better performance at the same time. This indicates that by expanding the search space, we can search model architectures that can be executed more efficiently on the deployed platform. Also, from the Table 4.6, it shows that FlOPs left and latency constraint is not highly correlated, which means using latency as the constraint can produce more accurate results.

doi:10.6342/NTU202101554

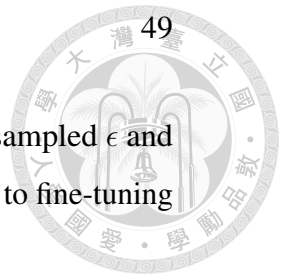
4.3 Ablation Study

Fitness Evaluation. During the searching process, fitness score plays an important role, and it will affect candidate selection. To have an accurate fitness score, fine-tuning for several epochs to recover performance drop is a common method. However, it's impractical since every network candidate needs to be fine-tuned, which takes a great amount of time. Therefore, we take a different evaluation approach: we only fine-tuning for 100 iterations and only updating the mean and variance in the batch-norm layers of the network. Therefore, for a batch size of 256 in the CIFAR-10 dataset, 100 iterations account for 0.05 epochs, which is far more efficient than the traditional method. We conduct experiments on two latency constraints: 40% and 90% to validate the effectiveness of the fast evaluation. The model used in the experiment is MobileNet, and the dataset is CIFAR-10. Table 4.7 shows the experiment result. The fast evaluation method achieves nearly the same performance as fine-tuning. Since fast evaluation uses much less training data than fine-tuning in the fitness evaluation, the fast evaluation uses more epochs fine-tuning the pruned network to recover performance drop. Epochs-Search in Table 4.7 indicates the lower bound of epochs used in the fitness calculation since

Table 4.7: **Comparison between fine-tuning and fast evaluation.** Statistics of performance and overall epochs used in our framework on different evaluation approaches. P. Top-1: Top-1 test accuracy. Epochs-Search: Epochs used in the searching process. Epochs-FT: Epochs used in fine-tuning the final pruned network to recover the performance drop. Overall epochs: Total epochs of both search and fine-tuning.

Method	P. Top-1(%)	Epochs-Search	Epochs-FT	Overall epochs
Fine-tune-90%	94.4	500	150	650
FastEval ($lat._{90}$)	**94.41**	**5**	**200**	205
Fine-tune-40%	92.6	500	150	650
FastEval ($lat._{40}$)	**92.61**	**5**	**200**	205

doi:10.6342/NTU202101554



the step 2 and 3 in the searching algorithm depend on the random sampled ϵ and the counter. Overall, fast evaluation can achieve similar performance to fine-tuning evaluation method and be more efficient.

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Chapter 5

Conclusion

In this thesis, we propose a novel pruning approach named Platform-aware Architecture Generator and Search (PAGS) to integrate the platform information into our pruning procedure. Using empirical measurements to evaluate the direct metrics and our architecture generator, we don't need to know the deep platform-specific knowledge. Several experiments are conducted to demonstrate the effectiveness of our generator, and we can achieve accurate pruned models that meet the resource budget compared with other works. Moreover, Followed by architecture search, we can select a better-pruned model. Finally, in various constraint settings of inference latency, we demonstrate the efficacy of our platform-aware network compression method.

doi:10.6342/NTU202101554

doi:10.6342/NTU202101554

Reference

[1] M. Lin, R. Ji, Y. Zhang, B. Zhang, Y. Wu, and Y. Tian, "Channel pruning via automatic structure search," in *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2020. v, 9, 18, 30

[2] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300. vii, 8, 10, 15, 41, 42, 43, 44, 45, 46

[3] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016. vii, 1, 3, 5, 13, 47

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. 1, 39

[5] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Proceedings of Neural Information Processing Systems (NIPS)*, 2015, pp. 91–99. 1

[6] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *arXiv preprint arXiv:1501.00092*, 2015. 1

[7] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015. 1

doi:10.6342/NTU202101554

[8] T. Chen, I. Goodfellow, and J. Shlens, "Net2Net: Accelerating Learning via Knowledge Transfer," in *International Conference on Learning Representations*, nov 2016. 1

[9] A. Tulloch and Y. Jia, "High performance ultra-low-precision convolutions on mobile devices," *Proceedings of Neural Information Processing Systems (NIPS)*, dec 2017. 1

[10] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization," *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2015. 1

[11] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, vol. 6, 2016, pp. 4166–4175. 1

[12] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proceedings of Neural Information Processing Systems (NIPS)*, 2015, pp. 1135–1143. 1, 13

[13] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5058–5066. 1, 14

[14] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2736–2744. 1, 14

[15] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1389–1397. 1, 14

doi:10.6342/NTU202101554

[16] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 9194–9203. 1, 14

[17] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 11 264–11 272. 1, 5, 15

[18] X. Ding, G. Ding, Y. Guo, and J. Han, "Centripetal sgd for pruning very deep convolutional networks with complicated structure," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4943–4953. 1

[19] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4340–4349. 1, 13, 14

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520. 2

[21] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, 2016. 2

[22] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 2

doi:10.6342/NTU202101554

[23] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," *arXiv preprint arXiv:1808.06866*, 2018. 3, 13

[24] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *The International Conference on Learning Representations (ICLR)*, 2016. 3

[25] C.-T. Liu, Y.-H. Wu, Y.-S. Lin, and S.-Y. Chien, "Computation-performance optimization of convolutional neural networks with redundant kernel removal," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5. 5, 13

[26] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016. 5, 14

[27] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *Proceedings of Neural Information Processing Systems (NIPS)*, 2018, pp. 875–886. 14

[28] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996. 14

[29] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015. 14

[30] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv:1312.6114*, 2014. 16

doi:10.6342/NTU202101554

[31] K. Sohn, H. Lee, and X. Yan, "Learning structured output representation using deep conditional generative models." in *Proceedings of Neural Information Processing Systems (NIPS)*, 2015. 16, 24

[32] D. Karaboga and B. Basturk, "Artificial bee colony (abc) optimization algorithm for solving constrained optimization problems," in *Proceedings of the 12th International Fuzzy Systems Association World Congress on Foundations of Fuzzy Logic and Soft Computing*, ser. IFSA '07. Springer-Verlag, 2007, p. 789–798. 18, 28

[33] B. Li, B. Wu, J. Su, G. Wang, and L. Lin, "Eagleeye: Fast sub-net evaluation for efficient neural network pruning," *arXiv preprint arXiv:2007.02491*, 2020. 28, 30

[34] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," *Technical report, University of Toronto*, 2009. 38, 39

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. 38, 39

[36] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017. 39

[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv:1412.6980*, 2014. 40

doi:10.6342/NTU202101554