# Behavior-based DNN Compression: Pruning and Facilitation Methods

March 2021

Graduate School of Systems Engineering
Wakayama University

Koji Kamma

# ニューロンの振舞いに基づくニューラルネットワークの圧縮：プルーニング手法およびより効果的な圧縮のための補助的手法

令和３年３月

和歌山大学大学院システム工学研究科

菅間　幸司

**Abstract**

This thesis presents a set of methods for compressing Deep Neural Network (DNN) models. In various Computer Vision tasks, DNN models show excellent performances. However, due to their heavy computational requirements, it is difficult to use them on edge devices with limited computational resources. Therefore, it is desired to develop the methods to compress large DNN models without degradation.

One of the effective approaches for this purpose is pruning. Pruning is a technique to reduce the computational cost of pretrained DNN models by removing redundant neurons. The most important requirement for the pruning methods is to preserve the accuracy of the pruned models as well as possible, while making them smaller. Therefore, we should design the pruning methods so as to prevent accuracy degradation.

This thesis presents 2 pruning methods and 2 methods for facilitating pruning as follows.

## Pruning methods

We propose Neuro-Unification (NU) and Reconstruction Error Aware Pruning (REAP). These methods do not only prune but also conduct *reconstruction* to prevent accuracy degradation.

In NU, we unify a pair of neurons having similar *behaviors*. Having similar behaviors is, in other words, those neurons' outputs have strong correlation. We prune one of them and update the weights connected to other one so as to reconstruct the behavior of the pruned one.

REAP is an extended version of NU. In REAP, when a neuron is pruned, the weights connected to all the remaining neurons are updated by using least squares method so as to minimize the error caused by pruning.

The problem of REAP is the large computational cost for selecting the neurons to be pruned. For neuron selection, we once try to prune each one of them, conduct reconstruction, and observe the reconstruction error, which is computationally expensive. For efficient neuron selection, we developed a biorthogonal system-based algorithm with which we can compute the reconstruction errors for all the neurons in one-shot.

## Pruning ratio optimization method

REAP is the best method in terms of minimizing the layer-wise error. However, we do not know how much this layer-wise error will have an impact on the overall model performance, and thus, we cannot tune the pruning ratio (the ratio of the neurons to be pruned) in each layer based on only the layer-wise error.

Therefore, we propose Pruning Ratio Optimizer (PRO) with which we can optimize the pruning ratios efficiently. The idea of PRO is to tune the pruning ratio in each layer so that the error in the final layer of the model is minimized. By combining PRO and REAP, we can optimize the pruning ratios while performing pruning, which results in preserving the accuracy even better.

## Serialization technique for ResNet

A problem of the layer-wise pruning methods including REAP is that ResNet is difficult to prune, because some of its layers cannot be pruned due to the *identity shortcuts*. This limitation is significant, because ResNet is used for various DNN models.

We propose a technique to transform a ResNet model into an equivalent serial model whose weights are partially fixed to conduct identity mapping. We call this model *Serialized Residual Network* (SRN). Although SRN has more neurons than ResNet, we can reduce them drastically by pruning, because SRN has a serial architecture and we can perform pruning in any layer.

## 概要

本論文では，Deep Neural Network（DNN）モデルを圧縮するための手法を提案する．様々なコンピュータビジョンのタスクにおいて，DNN モデルは良いパフォーマンスを示している．しかし，高い計算コストゆえ，大規模かつ高精度な DNN モデルを，計算リソースが乏しいエッジデバイス上で利用することは難しい．そのため，大きな DNN モデルを，精度を犠牲にすること無く，圧縮できる手法の開発が望まれている．

DNN を圧縮する方法の一つに，プルーニングがある．プルーニングとは，学習済みの DNN モデルから冗長なニューロンを削除することにより，その計算コストを低減する方法である．プルーニングにおいて最も重要な要件は，モデルを圧縮しつつ，元の精度を維持することである．それゆえ，プルーニング手法を開発する際は，圧縮対象のモデルの精度がなるべく低下しないような工夫が必要である．

本論文では，プルーニング手法と，プルーニングの効果を高めるための補助的な手法を，それぞれ 2 つ提案する．提案手法の概要を以下に示す．

### プルーニング手法

Neuro-Unification（NU）および，Reconstruction Error Aware Pruning（REAP）を提案する．これらの手法の特徴は，ただプルーニングを行うのではなく，モデルの精度を保つため，ニューロンの振舞いに基づく再構成を行うことにある．

NU では，振舞いが似た 2 つのニューロンの統合を行う．ニューロンの振舞いが似ているということは，それらの出力値に強い相関があるということである．それらのうち，一方をプルーニングし，もう一方のニューロンの重みを更新することで，プルーニングされたニューロンの振舞いを再構成する．

REAP は NU を拡張した手法である．REAP では，あるニューロンをプルーニングすると，残るすべてのニューロンの重みを最小二乗法を用いて更新し，再構成を行う．これにより，さらなる誤差の低減が可能である．

REAP の問題点は，プルーニングするニューロンを選択する際の計算コストが大きいことである．どのニューロンをプルーニングするか決める際，各ニューロンについて，プルーニングおよび最小二乗法による再構成を試し，再構成後の誤差を見る必要があるが，これは計算効率が悪い．そこで，すべてのニューロンの再構成後の誤差を一度に計算できるように，双直交基底を用いたアルゴリズムを考案した．

### 圧縮率最適化手法

通常，DNN モデルは多数の層で構成されている．モデル全体に渡ってプルーニングを実施する場合，各層における圧縮率（プルーニングされるニューロンの割合）を適切に設定することが望ましい．REAP は層ごとの誤差を最小化するという点において，最良のプルーニング手法である．しかし，層ごとの誤差がモデルのパフォーマンスにどれほど影響するかは未知である．そのため，層ごとの誤差だけを見ていては，圧縮率を適切に調整することはできない．

そこで，Pruning Ratio Optimizer（PRO）を提案する．PRO は，モデルの最終層の誤差を基準に，各層の圧縮率を調整する手法である．PRO と REAP を組み合わせることによって，モデルを圧縮する際の，精度面での劣化をより小さくできる．

### **ResNet の直列化技術**

ニューロン単位でのプルーニングを行う手法（REAP を含む）の共通の課題は ResNet というモデルを圧縮しにくいことである．ResNet の多くの層が恒等写像パスにつながっており，分岐構造をもつが，分岐のある層ではプルーニングを行えない．

そこで，ResNet を，直列構造を持つモデル *Serialized Residual Network*（SRN）に変換する手法を提案する．SRN は，その重みの一部を単位行列にすることで，ResNet と等価の計算を行える．SRN は ResNet よりも多くのニューロンを持つが，直列構造であるため，どの層においてもプルーニングを行える．それゆえ，一旦直列化してからプルーニングを実行することにより，モデルの計算コストを効率的に落とすことが可能となる．

# Contents


**I Introduction** 8

**1 Background** 8

**2 Our proposals** 10

**3 Structure of this thesis** 13

**4 Mathematical notations** 15

**II DNNs on Edge Devices** 16

**1 DNN architectures and computational complexity** 16

**2 Existing works exploring efficient DNNs** 18

2.1 Pruning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.2 Sparsification . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.3 Factorization . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.4 Quantization . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.5 Distillation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.6 Neural Architecture Search (NAS) . . . . . . . . . . . . . . . . . . 23

**3 Developments of Hardware devices** 24

**III Neuro-Unification** 26

**1 Introduction** 26

**2 Neuron behavior-based unification** 26

2.1 How to encode the neuron behaviors and unify the neurons having the same behavior . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.2 The case of the neurons having linearly independent behavioral vectors 30
2.3 Criteria for selecting neurons to be unified . . . . . . . . . . . . . . 31
2.4 The case of unifying neurons that have already been unified . . . . . 33
2.5 Problem formalization based on graph theory . . . . . . . . . . . . 34

2.6 Extra reconstruction . . . 36
2.7 Applying Neuro-Unification to convolutional layers . . . 38
2.8 Relevant methods . . . 38

**3 Experiments** **42**
3.1 Datasets . . . 42
3.2 Models . . . 42
3.3 Experiments with VGG16 on ImageNet . . . 43
3.3.1 Setups . . . 43
3.3.2 Results for fully connected layers . . . 43
3.3.3 Results for convolutional layers . . . 44
3.3.4 Ablation study . . . 44
3.4 Experiments with ResNet-56 on cifar-10 . . . 44
3.4.1 Setups . . . 44
3.4.2 Results . . . 45

**4 Summary of Part III** **46**

**IV Reconstruction Error Aware Pruning** **47**

**1 Introduction** **47**

**2 From NU to REAP** **49**
2.1 Neuro-Unification (NU) . . . 49
2.2 Reconstruction Error Aware Pruning (REAP) . . . 50
2.3 Neuron selection algorithm based on biorthogonal system . . . 51
2.4 Even faster computation for selecting the second neuron to be pruned 53
2.5 Algorithm . . . 54
2.6 Relation to CP . . . 55

**3 Experiments** **55**
3.1 Datasets . . . 56
3.2 Models . . . 56
3.3 VGG16 on ImageNet . . . 57
3.4 ResNet-56 on CIFAR-10. . . . 60
3.4.1 Setups . . . 60
3.4.2 Results . . . 62

3.5 DenseNet-121 on Stanford Dogs. . . . 63
3.5.1 Setups . . . . 63
3.5.2 Results . . . . 63

**4 Summary of Part IV** **64**

**V Pruning Ratio Optimizer** **65**

**1 Introduction** **65**

**2 Related works** **66**

**3 How to optimize the pruning ratio with a layer-wise pruning method** **67**
3.1 Formulation of REAP . . . . 67
3.2 Pruning Ratio Optimizer (PRO) . . . . 68
3.3 Strategy for more efficient optimization . . . . 69
3.4 Algorithm . . . . 70

**4 Experiments** **70**
4.1 Datasets . . . . 70
4.2 Models . . . . 72
4.3 VGG16 on ImageNet . . . . 72
4.3.1 Setups . . . . 72
4.3.2 Results . . . . 73
4.4 ResNet-56 on CIFAR-10 . . . . 77
4.4.1 Setups . . . . 77
4.4.2 Results . . . . 77

**5 Summary of Part V** **78**

**VI Serialized Residual Network** **79**

**1 Introduction** **79**

**2 Related works** **81**

3 **Serialized Residual Network (SRN)** **81**
3.1 How to build SRN that emulates ResNet . . . . . . . . . . . . . . . . 81
3.2 Training strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
3.2.1 Problem caused by having both pretrained weights and untrained weights . . . . . . . . . . . . . . . . . . . . . . . . . . 84
3.2.2 Side effect of L2 regularization . . . . . . . . . . . . . . . . . . 85

4 **Experiments** **86**
4.1 Experiments to facilitate pruning . . . . . . . . . . . . . . . . . . . . . 86
4.1.1 Dataset . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.1.2 Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.1.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.2 Experiments to improve accuracy . . . . . . . . . . . . . . . . . . . . . 89
4.2.1 Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.2.2 Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
4.2.3 Results on CIFAR-10 . . . . . . . . . . . . . . . . . . . . . . . 90
4.2.4 Results on CUB-200 . . . . . . . . . . . . . . . . . . . . . . . . 93
4.2.5 Results on STL-10 . . . . . . . . . . . . . . . . . . . . . . . . . 94
4.3 Ablation studies and analyses . . . . . . . . . . . . . . . . . . . . . . . 94
4.3.1 What if we use conventional L2 regularization instead of EWR? 95
4.3.2 What if we unfix all the fixed weights at once and start training instead of AUWT? . . . . . . . . . . . . . . . . . . . . . . . . . 96
4.3.3 What the identity mapping portion of kernels will be like after reconstruction? . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

5 **Summary of Part VI** **97**

**VII Summary** **98**

**Acknowledgements** **99**

**Appendix** **107**

**A DNN model architectures** **107**
A.1 VGG16 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

A.2 ResNet-18 and ResNet-56 . . . . . . . . . . . . . . . . . . . . . . . . . 107
A.3 DenseNet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

**B Gram-schmidt process-based algorithm for neuron selection in REAP 107**
B.1 Gram-Schmidt process based algorithm . . . . . . . . . . . . . . . . . 111
B.2 Formalization and proof . . . . . . . . . . . . . . . . . . . . . . . . . . 113
B.3 Case studies with VGG16 . . . . . . . . . . . . . . . . . . . . . . . . . 114

**C Tips for implementation of REAP 114**

**D How adequate is REAP's solution? 116**

**List of Publications 119**

# Part I
# Introduction

## 1 Background

Since Hinton et al. won ImageNet Large Scale Visual Recognition Competition (ILSVRC) with AlexNet in 2012 [1], researchers have been developing various Deep Neural Networks (DNNs) for various Computer Vision tasks, such as recognition, detection, segmentation, and so on. Today, DNNs are already used in many industrial applications, and are expected to spread to wider range of industries in near future.

One of the bottlenecks of DNNs is that the inference (as well as training) is computationally expensive. In laboratories, large GPUs solve this problem. For example, VGG16 [2] model runs on NVIDIA Geforce GTX 1080 GPU at about 60 fps. This GPU consumes up to 180 W, thus a sufficient power supply is required to use it. Moreover, the financial cost is also quite high due to large power consumption. Thus, we need an environment with rich resources for using the DNN models.

On the other hand, we may want to use the DNN models on the edge devices with insufficient computational resources, such as in-vehicle cameras, security cameras, smartphones, drones, and so on. Some applications require high performance in accuracy and inference speed under severe constraints in power consumption, memory size, installation space, operating temperature, price, and so on. Therefore, if a large GPU is used to meet the performance requirements in accuracy and speed, it will not be able to meet the constraints. Conversely, if we select a device that meets the constraints, it will not be able to satisfy accuracy and/or inference speed requirements. For example, Movidius NCS, a USB stick that includes a processor designed for DNN inference, consumes only 1W, although the large DNN models, such as YOLOv3 [3], cannot be deployed on it due to small memory space.

Then, how can we meet the requirements and the constraints competing each other (e.g. accuracy and power consumption) simultaneously? One idea is to send the images captured by the edge devices to the servers equipped with large GPUs (or other types of strong computational resources) where DNN inference is performed, and send the inference results back to the edge devices [4]. Another idea is to compress pretrained large DNN models by reducing their redundancy, and use the compressed models on the edge devices [5, 6]. The advantages and the disadvantages

of these approaches are described below.

By sending the images from the edge devices to the severs, we can use strong computational resources on the servers. Then, we can run large and accurate DNN models as they are, and do not need to worry about the constraints of, for example, power consumption and memory space. However, this approach may end up in higher latency. Although the inference itself can be fast with the strong computational resources, there is a delay due to the communication between the edge devices and the servers. In a more severe case, if the communication is stalled or disconnected due to congestion, the systems composed of the edge devices and the servers cannot keep working, which is a concern for stability and reliability. These disadvantages are fatal for some applications, for example, safety assistant systems for car drivers, where low-latency, stability, and reliability are critically important. There are also privacy issues when sending images via Internet, as the images taken in public locations usually include a lot of personal information.

On the other hand, compressing and deploying the DNN models on the edge devices solves most of these problems. Because capturing the images and the inference with the DNN models can be done on the same device, there are no concern related to communication or privacy. However, the problem is that compressing the DNN models often ends up in accuracy degradation. Because the accuracy is normally the most important factor, significant accuracy degradation is not acceptable. Therefore, it is crucial how well we preserve the accuracy of the pruned models while making them smaller.

Therefore, there are strong demands for the methods that can compress the pretrained DNN models and preserve their accuracy simultaneously. The effective compression methods will make the DNN models easier to be used for many applications, removing the concerns related to communication and privacy. A good compression method will dramatically expand the usages of DNNs.

There might be an opinion that the compression methods do not matter, because the compressed DNN models are retrained anyway in order to recover their accuracy. However, as we will show in this thesis, the better we preserve the accuracy of the compressed models, the more accurate those models eventually become after retraining. If significant accuracy degradation happens, retraining will be more time-consuming. More importantly, as we will show in Part V, the compression methods that can preserve the model accuracy well enable us to tune the pruning ratio (the ratio of the neurons to be pruned) in each layer efficiently. Therefore, it is important to choose a good compression method that can prevent accuracy

degradation as well as possible.

## 2 Our proposals

One of the effective approaches for compressing pretrained DNN models is *pruning* [5, 7, 8]. Pruning is to remove the redundant neurons (or weights) from the models.

The most important point for pruning is to prevent accuracy degradation. For this purpose, it is important to evaluate the redundancy of the neurons properly. If we prune the neurons that are not redundant, the pruned model suffers significant degradation.

It is also important to conduct *reconstruction* when performing pruning. Here, reconstruction is to update the remaining weights so as to compensate the error caused by pruning. Some methods that perform reconstruction can preserve the accuracy of the pruned models well, which results in saving time for retraining and eventually achieving better accuracy after retraining [8, 9].

Moreover, it is important to know how many neurons can be pruned in each layer. If too many neurons are pruned in a layer, the representation ability of the layer becomes poor, and the accuracy of the model is significantly degraded.

In this thesis, we propose Neuro-Unification (NU) [10], Reconstruction Error Aware Pruning (REAP) [6], Pruning Ratio Optimizer (PRO), and Serialized Residual Network (SRN). NU is a pruning method and REAP is the extended version of NU. Especially, REAP has a high ability of preserving the accuracy of the pruned models, as it takes a sophisticated strategy for pruning and reconstruction. PRO is a method for optimizing the pruning ratio (ratio of the neurons to be pruned) in each layer. SRN is a method to facilitate pruning for specifically ResNet models [11]. The follows are the brief explanations of the proposed methods.

**Neuro-Unification (NU)**

NU is the pruning method that unifies a pair of neurons having similar outputs. In NU, the pruned neuron's outputs are reconstructed from another one's outputs so as to compensate the error caused by pruning. Therefore, pruning with NU causes less accuracy degradation.

**Reconstruction Error Aware Pruning (REAP)**

REAP is the extended version of NU. In REAP, the pruned neuron's outputs are reconstructed by using all the remaining neurons in the same layer. Therefore, the

error caused by pruning can be minimized. REAP is theoretically the best method among the methods that conduct pruning and reconstruction layer by layer.

**Pruning Ratio Optimizer (PRO)**

As the DNN models usually have several layers, we need to tune the *pruning ratio* in each layer properly in order to preserve the accuracy well. REAP is a powerful method in terms of preventing the layer-wise error. However, it is not obvious how much the layer-wise error will affect the accuracy, which makes it difficult to determine the pruning ratios.

Pruning Ratio Optimizer (PRO) is a method that can be combined with REAP for optimizing the pruning ratio in each layer. The idea of PRO is to tune the pruning ratios so that the error in the final layer of the pruned model will be minimized. In PRO, we repeatedly select the most redundant layer and prune some neurons, until the pruned model becomes small/fast enough. When selecting a layer, we once try pruning in each layer and observe the error in the final layer. We eventually conduct pruning in the layer where pruning has the smallest impact on the outputs in the final layer. By using PRO, we can conduct pruning while optimizing the pruning ratios, which enables us to perform more effective DNN compression.

**Serialized Residual Network (SRN)**

Generally, ResNet [11] is difficult to be pruned due to its architectural feature. ResNet architecture is composed of stacked blocks, and each block is composed of several layers and has branched paths. In each block, the inputs are propagated forward as they are in one path, and linear transformations (convolutions) are performed in the other path, and both are eventually added. At this addition, two inputs must have the same dimensions, which means the layer that have branched paths cannot be pruned.

Therefore, we propose a method to convert ResNet into an equivalent serial model that we call *Serialized Residual Network* (SRN). SRN can emulate ResNet by setting some weights to perform identity mapping. Even though SRN has more computational complexity than ResNet, it is easier to be compressed by pruning, because SRN has a serial architecture and any layer can be pruned.

We noticed that training SRN is difficult in the traditional way. The problem is the *side effect* of L2 regularization. L2 regularization strongly penalizes the weights that are far from zero. The identity mapping portion of SRN's weights contains the values that are equal to 1, and 1 is a large value for a weight of DNN models. These

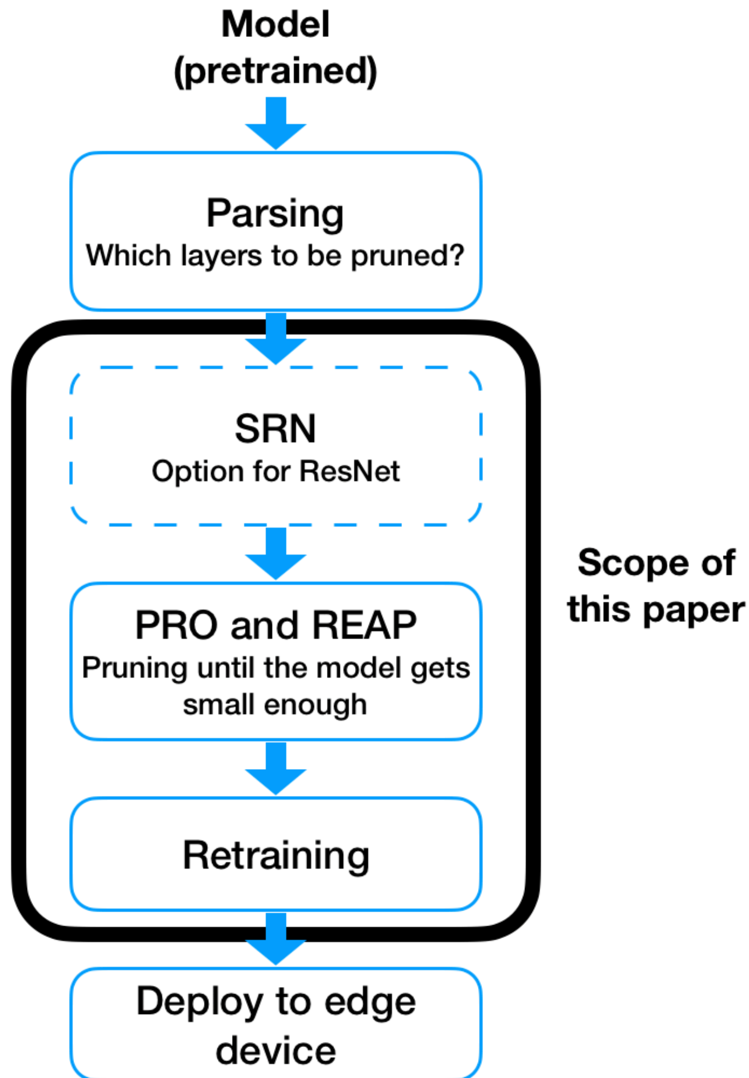


Figure I.1: The system where the proposed methods are used in. The user inputs a model. Parsing is done to analyze the model architecture, and identify which parts of the architecture can be serialized and which layers can be pruned. If the model contains ResNet architecture, serialize them. PRO and REAP are used to conduct pruning while optimizing the pruning ratio in each layer. The pruned model is retrained to recover its accuracy. Finally, the model can be deployed on an edge device.

weights are penalized too strongly by L2 regularization, and thus, they are updated drastically by training, which often ends up in accuracy degradation. In order to avoid this issue, we propose Elastic Weight Regularization (EWR). Differently form L2 regularization, EWR penalizes the weights that are far from their initial values. Therefore, no matter how large the initial values are, they are not penalized too strongly. With EWR, SRN can be trained stably.

Throughout this thesis, what we would particularly like to appeal is the neuron selection algorithm of REAP. In REAP, for selecting the neuron to be pruned, we once try to prune each neuron, conduct reconstruction with least squares method, and observe the error, which does not seem feasible due to huge computational cost. Therefore, we developed a biorthogonal system-based algorithm with which the reconstruction errors for all the neurons can be computed in one-shot, which results in significant reduction of the computational cost for neuron selection. This algorithm is a new application of biorthogonal system.

We eventually want to construct the system illustrated in Fig. I.1. When a model is given, parsing is conducted to investigate the architecture of the model. This step is to automatically check which layers can be pruned and which part of the architecture can be serialized. If the model has ResNet architecture, serialize them. Then, PRO and REAP are applied to perform pruning, while optimizing the pruning ratios. The pruned model is retrained so as to recover the damage of pruning. Finally, the pruned model can be deployed on an edge device. With this system, we just need to input the model we want to prune and tune a few parameters related to pruning and retraining. Therefore, one can use this system without special knowledge or the experience on pruning. Note that the scope of this thesis is to propose the pruning methods (REAP and NU) and its facilitation methods (PRO and SRN). The implementation for parsing function is our future work.

## 3 Structure of this thesis

The rest of the thesis are shown in Fig. I.2. The trends in DNN utilization on edge devices are summarized in Part II. The details of the proposed methods are explained in Part III-VI. The summary of this thesis is written in Part VII.

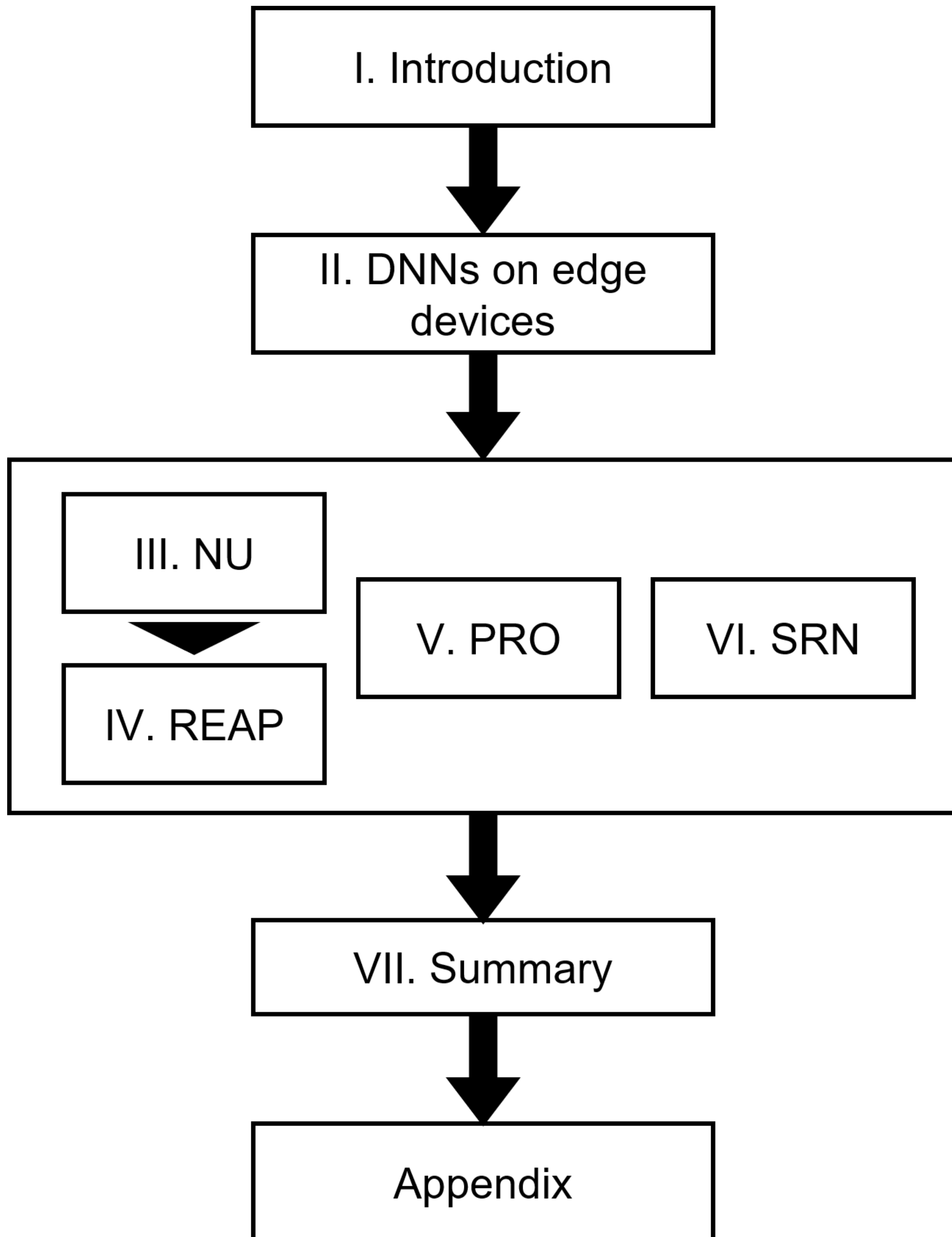


Figure I.2: The structure of this thesis.

# 4 Mathematical notations

Throughout this thesis, we use the following notation rules, unless otherwise noted.

- A fine lowercase letter, such as $x$, denotes a scalar.
- A bold lowercase letter, such as $\boldsymbol{x}$, denotes a vector.
- A fine uppercase letter, such as $X$, denotes a matrix or a more than 2-dimensional tensor.
- A calligraphy letter, such as $\mathcal{A}$, denotes a set.
- $\mathbb{R}$ denotes the set of whole real numbers, and we express tensor dimensions by using $\mathbb{R}$ with superscripts, for instance, $\boldsymbol{x} \in \mathbb{R}^n$ means that $\boldsymbol{x}$ is a real vector that has $n$ length, and $X \in \mathbb{R}^{n \times m}$ means that $X$ is a real matrix that has $n \times m$ shape.
- $\|\cdot\|$ denotes L2 norm and $\|\cdot\|_\mathrm{F}$ denotes Frobenius norm.
- $\langle\cdot,\cdot\rangle$ denotes inner product.
- $\leftarrow$ denotes substitution from RHS to LHS.

For other types of symbols, we define them accordingly.

# Part II
# DNNs on Edge Devices

In Part II, we first explain the typical architectures of DNNs briefly and discuss their computational cost. Then, we explain the existing works that aim for compressing pretrained DNN models. We also outline the recent developments of hardware for edge devices.

## 1 DNN architectures and computational complexity

In this section, we discuss the computational cost of DNNs. Recently, Convolutional Neural Networks (CNNs) are used as standard in Computer Vision. Therefore, we mainly focus on CNNs in this thesis. Henceforth, when we write "DNN", it shall mean CNN, unless otherwise noted.

For using DNN models on edge devices, there are two major concerns in terms of computational cost. One is time complexity, and the other is space complexity. The count of floating point operations per input (FLOPs) is normally used as the metric for time complexity, and the number of weights is used as the metric for space complexity. These two are due to different structural features of DNNs. As shown in Fig. II.1, a typical DNN model has two different types of layers, the convolutional layers and the fully connected layers. While the convolutional layers normally account for most of FLOPs, the fully connected layers account for majority of weights.

The operation in the fully connected layers is simple. Each neuron in a layer is connected to all the neurons in the next layer, as shown in Fig. II.1. In the fully connected layers, a simple matrix multiplication is performed. Due to the dense connections between the neurons in sequential layers, the fully connected layers have a significant number of weights. Let $n$ and $n'$ denote the numbers of neurons in a layer and the next layer, respectively. Then, the number of weights between these two layers is given by $nn'$. FLOPs per single input can also be given by $nn'$.

On the other hand, the operations in the convolutional layers are more complicated. In a convolutional layer, the outputs corresponding to a single image are represented by a 3-dimensional tensor. The kernel, which is also represented by a 3-dimensional tensor and is spatially smaller than the feature maps (e.g. width $\times$ height of $3 \times 3$), conducts sliding window operations. It moves on the feature maps,

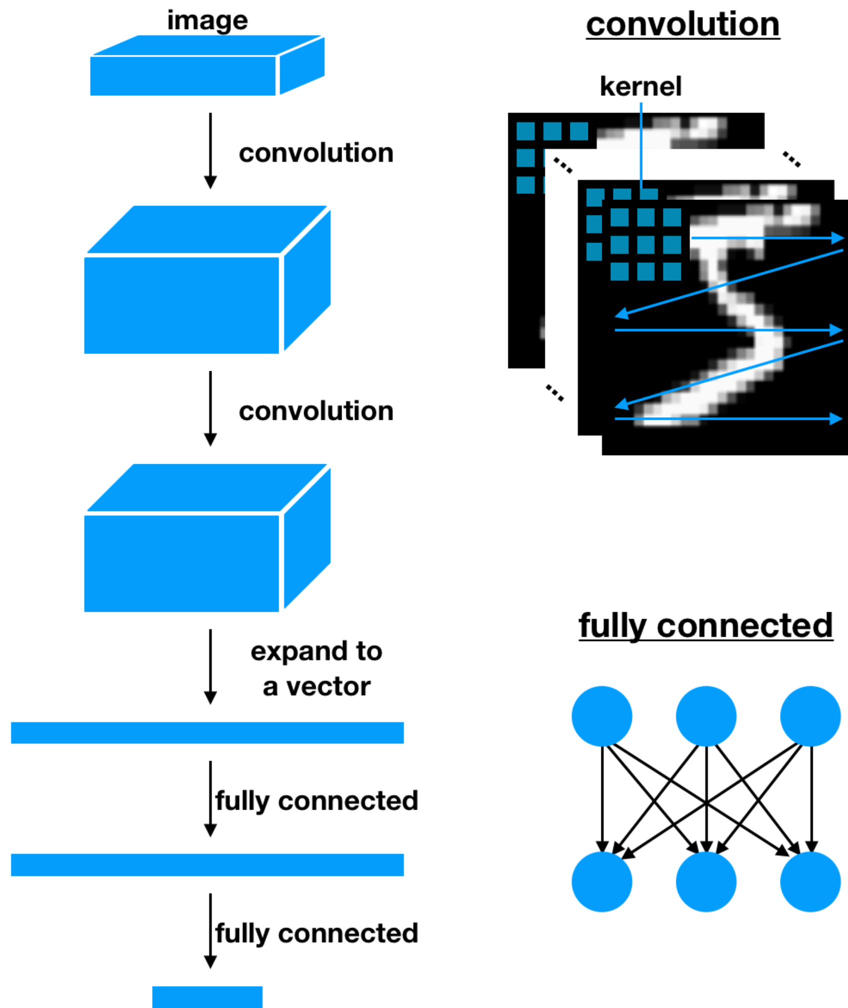


Figure II.1: An example of a DNN architecture with convolutional layers and fully connected layers. In convolutional layers, 3-dimensional kernel moves on feature maps and compute inner product of itself and the overlapping part of feature maps at each location. This operation usually requires larger FLOPs than the operation in fully connected layers. In fully connected layers, each neuron in a layer is connected to all the neurons in the next layer, and simple matrix multiplication is conducted. The number of weights in fully connected layers is typically larger than that in convolutional layers.

computing the inner product of itself and the overlapping part of the feature maps at each location. FLOPs in the convolutional layers depend on the feature map resolution $w_f \times h_f$, the kernel resolution $w_k \times h_k$, the number of input channels $c_{in}$ and output channels $c_{out}$, and horizontal and vertical strides $(s_w, s_h)$. The strides are the number of pixels that kernel moves at once. Therefore, FLOPs are given by $w_k h_k w_f h_f c_{in} c_{out} s_w^{-1} s_h^{-1}$. This requires a lot of FLOPs, especially with high feature map resolution and high kernel resolution. The number of parameters is given by $w_k h_k c_{in} c_{out}$. Thanks to the spatially small shapes of the kernels, the convolutional layers typically have fewer weights than the fully connected layers.

Thus, it happens that the convolutional layers account for most of FLOPs and the fully connected layers account for most of weights, in a typical DNN model. For example, VGG16 [2] has 13 convolutional layers and 3 fully connected layers. Only 3 fully connected layers account for approximately 90% of weights and 13 convolutional layers account for 99% of FLOPs.

Nonetheless, some recent works, such as [11], have successfully made the fully connected layers smaller by using Global Average Pooling. In other cases, some models that are called as Fully Convolutional Networks use convolutional layers instead of fully connected layers [12]. For this reason, in this thesis, we mainly focus on compressing convolutional layers, although we also try to compress fully connected layers in some experiments.

## 2 Existing works exploring efficient DNNs

A lot of works have been done to explore efficient DNNs. There are two major approaches. One is to reduce the redundancy of pretrained large DNN models, which includes pruning, sparsification, factorization, quantization, and distillation. The other is Neural Architecture Search (NAS) to search the architectures that can achieve high accuracy within a given computational budget in the context of training from scratch.

### 2.1 Pruning

Pruning is to remove the neurons or the weights that are unimportant or redundant. The pruning methods can be divided into two groups: the weight pruning methods [5, 13, 14, 15] and the structured pruning methods [9, 8, 16, 17, 18, 19].

The weight pruning methods prune the weights that are redundant or do not contribute to the performance of the model significantly. In practice, the pruned

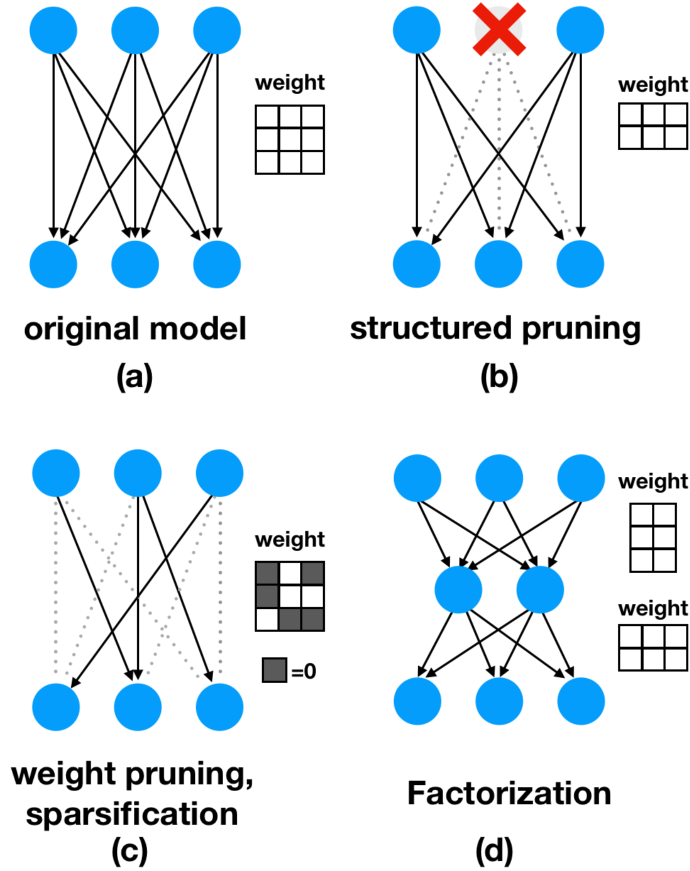


Figure II.2: The conceptual drawings of the compression methods for DNN models. (a) The original model. (b) The model compressed by structured pruning methods. (c) The model compressed by weight pruning methods or sparsification methods. (d) The model compressed by factorization methods. The advantage of structured pruning is that the weight matrix gets smaller by pruning, which means that the pruned model can be deployed with a general hardware device and a general library. On the other hand, the models compressed by the weight pruning methods and the sparsification methods require the environments that can conduct operations at only non-zero weights. The factorization methods may reduce the computational cost of the model, however, additional layers are added to the model which bring extra computational overheads.

weights are not actually removed but are set to zero.

The first work of weight pruning is Optimal Brain Damage (OBD) [5]. OBD evaluates the importance of the weights based on the Hessian of the cost function. As it is computationally intensive to calculate the whole Hessian, OBD only computes its diagonal entries, and assumes that the non-diagonals are zero. However, this is not a reasonable assumption, because the weights in the same model are obviously dependent on each other. Optimal Brain Surgeon (OBS) not only prunes but also conducts *surgery* (Note that, in our proposed methods, we call it *reconstruction* instead of *surgery*.) to compensate the damage of pruning by tuning the remaining weights based on the whole Hessian [7]. However, as already mentioned, it is not feasible to compute the whole Hessian for large DNN models that have millions of weights. Therefore, OBS can be applied to only small models. There are also the magnitude-based pruning methods that prune the weights based on their absolute values [20, 21]. However, pruning the weights having small absolute values may have significantly impact on accuracy, and vise versa.

The common drawback of the weight pruning methods is that the pruned model has sparse weight matrices/tensors and their shapes are still the same with before pruning, as shown in Fig. II.2 (c). Therefore, in order to take advantage of weight pruning, the pruned model should be deployed on an environment supporting sparse computation that skips multiplications with zero weights.

On the other hand, the structured pruning methods conduct neuron-level pruning. This group also includes the methods based on the derivative information of the cost function, such as [22], and the magnitude based methods, such as [23] and [24]. NU and REAP (the proposed methods in this thesis) and some relevant methods, such as ThiNet [9] and Channel Pruning (CP) [8] also belong to this group. The advantage of the structured pruning methods is that when a neuron is pruned, the whole weights connected to it are also removed, and thus, the weight tensor will have a smaller shape after pruning, as shown in Fig. II.2 (b). Therefore, the pruned model can be deployed in general devices with general libraries.

We can also categorize the pruning methods from another perspective: the holistic pruning methods [5, 13, 23, 22, 17] and the layer-wise pruning methods [10, 6, 8, 9, 14, 16, 25].

The holistic methods are designed for comparing the importance of the neurons/weights in the whole model together and removing the least salient one. For example, the method proposed in [22] aims for pruning convolutional layers and evaluates the importance of the channels based on the first derivative information

of the cost function. However, as the cost function and the weights normally have non-linear relationship, it is not reasonable to use only the first derivative information. Another example is Structured Probabilistic Pruning (SPP) [17], a pruning method using dropout. The idea of SPP is to drop some weights out once and conduct training, and if it ends up in high accuracy, it means the dropped weights are not important and can be eventually pruned. Although, as training has to be repeated many times to evaluate the importance of neurons, SPP is a computationally intensive way of pruning.

On the other hand, some recent works [10, 6, 8, 9, 14, 25] have offered the layer-wise pruning methods. The layer-wise methods conduct pruning in each layer separately based on the layer-wise error. As their optimization problems are simpler than those of the holistic methods, it is possible to use more theoretically sound criteria for selecting the neurons to be pruned. For example, Layer-Wise Optimal Brain Surgeon (LOBS) [14] prunes the weights and updates the remaining ones based on the Hessian of the MSE of layer-wise outputs over only the weights in that layer. While the original holistic OBS computes the Hessian of the cost function over all the weights of the model, LOBS computes the Hessian layer by layer, which significantly reduces the computational cost for computing the Hessian. Therefore, LOBS can be used for compressing larger DNN models. Channel Pruning (CP) [8] and ThiNet [9] prune the neurons based on the layer-wise error and conduct *reconstruction* with least squares method so as to compensate the error caused by pruning. Our REAP [6] are closely relevant to CP and ThiNet, however, we take a more sophisticated approach, as we will discuss in Part IV.

## 2.2 Sparsification

The sparsification methods make the weight matrices/tensors sparse by conducting extra training on the pretrained models with L1 regularization [26, 27]. The recent works include Sparse Convolutional Neural Network that combines L1 regularization for convolutional layers and tensor decomposition [26]. Zhao et al. proposed a group Lasso-based method for feature selection of multi-modal DNN models [28].

The theoretical weakness of sparsification is that L1 regularization shifts the global minimum of the cost function. Thus, weight selection results may not be optimal in terms of preserving the accuracy of the model. In addition, similarly with the weight pruning methods, the weight matrices/tensors retain the same dimensions after sparsification, as shown in Fig. II.2 (c). Thus, we need the environments dedicated for the sparsified models that skip the computations with the zeroed weights

to take advantage of sparsification.

Park et al. developed the way of implementation optimized for sparse kernel and dense feature maps in convolutional layers [29]. However, with such an implementation, the advantage of sparsification in inference speed may not be significant as it seems based on the FLOPs reduction, because they have overhead for finding which locations in the weight matrices/tensors are zero and have to be skipped.

## 2.3 Factorization

The idea of factorization is to decompose a large weight matrix/tensor into several smaller matrices/tensors, as shown in Fig. II.2 (d). The most fundamental method in this group is presented in [30]. They apply SVD to a large weight matrix, and approximate it by the product of small matrices by discarding the components with small singular values. This results in reducing the weights with small sacrifice of accuracy. For example, assume that a $m \times n$ matrix is approximated by the product of a $m \times o$ matrix and a $o \times n$ matrix. If $o \ll m, n$, the number of weights reduces from $mn$ to $(m+n)o$. Jaderbery et al. expanded this idea for convolutional layers [31]. They use row rank expansion technique for factorizing convolutional kernels. Recently, Kossaifi et al. proposed Spatio-Temporal convolution based on spatial redundancy of kernels [32]. They replace a $h \times w \times c$ (hight $\times$ width $\times$ depth) kernel by $h \times 1 \times c$ and $1 \times w \times c$ kernels. In [33], Wen et al. present Force Regularization that makes the weight tensor span a lower rank space, in order to make it easier to factorize the weight tensors. Some other methods [34, 35] also belong to this group.

The drawback of factorization is that they may indeed reduce weights and FLOPs, however, they add extra layers that have computational overheads. Therefore, the effectiveness of factorization may not be as significant as it seems, depending on the computational environments, model architecture, and so on.

## 2.4 Quantization

The methods in this group reduce the redundancy of each bit-wise operation, e.g. changing floating point precision from 32-bit to 8-bit. For lower-bit operations, special hardwares and libraries are required.

A further development of this idea is binarization. BinaryConnect [36] is a method to produce the DNN models with only 2 weight values (e.g. -1 and 1) so that the operations can be done by only additions and subtractions. As additions and subtractions are less expensive than multiplications, the inference can be faster

by binarization. The binarized models can be deployed on general hardwares and libraries. However, as they do not need multipliers, it is more ideal to use the dedicated environments optimized for additions and subtractions.

Rastegari et al. suggested the XNOR model that binarize both the weights and the input data to achieve further acceleration [37]. High-order Residual Quantization (HRQ) [38] is an extension of XNOR. In HRQ, as binarization of the input data causes residuals, they conduct second binary approximation for compensating the residuals. CLIPQ [39] is a method that conducts quantization, pruning, and retraining in parallel. Other than compression purpose, Xu et al. used quantization technique for preventing overfitting during training [40].

## 2.5 Distillation

Hinton et al. proposed Knowledge Distillation [41], a method to transfer the knowledge learned by a pretrained large model (teacher) into a small model (student). When training the student, its weights are updated so as to minimize the output difference from the convex combination of the ground truth and the teacher's outputs. The student trained in this way shows good performance for its size. Mirzadeh et al. proposed multi-step distillation [42]. They found out that Knowledge Distillation tends to fail when the student has much smaller architecture than the teacher, and multi-step distillation using the intermediate-sized models can ease this problem.

Yim et al. reports that Knowledge Distillation can also be used for improving the accuracy of the model, other than compression purpose [43]. Compared to the teacher that is trained without Knowledge Distillation, the student having the same architecture with the teacher but trained using the teacher's knowledge tend to achieve higher accuracy.

The most significant drawback of Knowledge Distillation is that it is difficult to search the proper architecture for the student. Typically, the student has fewer layers and fewer neurons in each layer than the teacher, although we do not know how fewer they can be. Therefore, we need to conduct training to judge if the current architecture is proper or not, which is time-consuming.

## 2.6 Neural Architecture Search (NAS)

Apart from the compression methods mentioned above, Neural Architecture Search (NAS) methods have also been developed. The NAS methods can be divided into two groups: the evolutionary algorithm-based methods [44, 45, 46], and the rein-

forcement learning-based ones [47, 48, 49]. The idea of these approaches is to prepare a graph that the DNN model will be built on, put a layer (or a block composed of several layers) on each node, and train the model built on the graph. Reinforcement learning or evolutionary algorithm are used to optimize the architecture in each node. These approaches are computationally intensive.

For more efficient architecture search, Progressive NAS (PNAS) has been proposed [50]. The idea of PNAS is to begin with a single node, search an optimal architecture on the only node, and then, stack the same architecture a desired number of times, which makes it less time-consuming to optimize the model architecture. Nonetheless, even this method is still computationally intensive. In addition, the NAS methods require people to determine lots of things including the graph structures, the layer types that may be put on each node, the hyper-parameters for training, reinforcement learning, and evolutionary algorithm.

## 3 Developments of Hardware devices

In this section, we quickly review the recent developments of hardware devices for edge applications.

Movidius, a company that is now in a group of Intel, released Neural Computing Stick (NCS). NCS is a USB stick that includes Myriad$^{\mathrm{TM}}$ 2 Vision Processing Unit that can perform up to 15 GFLOPS with approximately 1 W of power, and has 4 GB memory. NCS can be found in lots of applications, such as gesture-controlled drones, smart security cameras, and so on. The supported frameworks are Caffe and TensorFlow. However, its ability is limited to relatively small DNN models. For example, YOLOv3 [3], a object detection model, cannot be deployed on NCS due to memory shortage, while TinyYOLO, the inferior and smaller version of YOLOv3, can be deployed on it.

NCS2 is the successor model of NCS and uses a chip called Myriad VPU, which has the computational ability of 1 TOPS. It is relatively inexpensive at around 7,000 JPY (70 USD). Although, even with 4 units of NCS2, YOLOv3 runs at only 13 FPS, while their total power consumption is still large (up to 8 W with 4 units) and a somewhat large installation space is required for 4 of them. Thus, their computational ability and efficiency still need to be improved in order to be used as standard in edge devices.

NVIDIA manufactures Jetson Nano, a single board computer designed for DNN inference whose price is approximately 10,000JPY (95 USD). As it uses Linux Op-

eration Systems, all the major DNN frameworks can be used on it. With dedicated library TensorRT, the computational flow with the deployed model is optimized to accelerate the inference. It has 128-core GPU, 4 GB memory, and the computational ability of 472 GFLOPS in FP16 mode. It can run a relatively large model called ResNet-50 at 36 FPS. On the other hand, power consumption is up to 10 W, which may be too large for many applications.

The successor model of Jetson Nano, Jetson Xavior NX, has a 384-core GPU, which has the computing ability of 21 TOPS and 8 GB of memory. It can run DNN models 5 to 10 times faster than Jetson Nano, although with a power consumption of 10 to 15 W, which is quite large for many edge applications. Also, the high price of 40,000 JPY (400 USD) is not appreciated for industries.

Sony developed IMX500/501, chips with an image sensor and a small processor for DNN inference. The price is 10,000 JPY and 20,000 JPY (100 USD and 200 USD), respectively. It has the computing ability of running MovileNet V1 [51] at more than 30FPS.

There is a move from Microsoft team to promote the use of DNNs on FPGAs. There is also a growing trend to use smartphones with SoCs designed for DNNs. For example, HiSilicon's Kirin990 chips have been developed for smartphones that can perform at the competitive level to some legacy NVIDIA GPUs [52].

As mentioned above, the recent developments on hardware devices are remarkable. Some may argue that if the hardware continues to evolve at this rate, it will be easier to use large DNN models on the edge devices, eliminating the demands for DNN compression methods. But will it really be true?

For example, in the applications where safety is critically important, such as a driver assistance system in a car, the accuracy cannot be compromised. In other context, one may want to use a better model for their product in order to improve its quality. Then, there will be the demands for compressing the models that perform better than the ones currently used but cannot be deployed as they are due to the limitation of computational budget.

Also, better devices are generally pricer. For some products, there will be demands for using as good a model as possible within severe financial budget. Then, it will be an option to use the compressed models on the smaller devices.

For these reasons, the demands for DNN model compression methods are not likely to disappear, no matter how well the evolution of the hardware devices will be.

# Part III
# Neuro-Unification

## 1 Introduction

In Part III, we propose Neuro-Unification (NU), a pruning method to compress DNN models. The idea of NU is to unify the neurons that behave similarly. When we feed the images into a model, each neuron outputs a scalar value corresponding to each image. By recording those outputs, we create a behavioral vector for each neuron. If there are a pair of neurons that have similar behavioral vectors, we unify those neurons. Unifying a pair of neurons is, in other words, 1) pruning one of them and 2) having the other one to emulate the pruned one's behavior by updating the weights so as to compensate the error caused by pruning. We call the former *pruning*, the latter *reconstruction*, and two together *unification*.

Neuro-Unification can be applied to convolutional layers as well. In convolutional layers, we reshape the feature maps so that the sliding window operation can be described as a simple matrix multiplication. Then, we conduct channel-level unification.

We conducted the experiments with several models and datasets for recognition tasks. The results demonstrate that the proposed method can compress both the fully connected layers and the convolutional layers with the small sacrifice of accuracy compared to the existing methods.

The rest of Part III are as follows. Sec. 2 explains the theory of NU. Sec. 3 shows experiments to verify the effectiveness of NU. In Sec. 4, we show summary of Part III and discuss the possibility of NU to improve.

## 2 Neuron behavior-based unification

Fig. III.1 shows the flow chart of NU. For each layer, we unify the neurons in the following scheme.

1) Feed several images into a model and encode the behavior of each neuron.

2) Compute the similarity of every possible neuron pair based on their behavioral vectors.

3) Unify a pair of neurons that have the most similar behaviors.

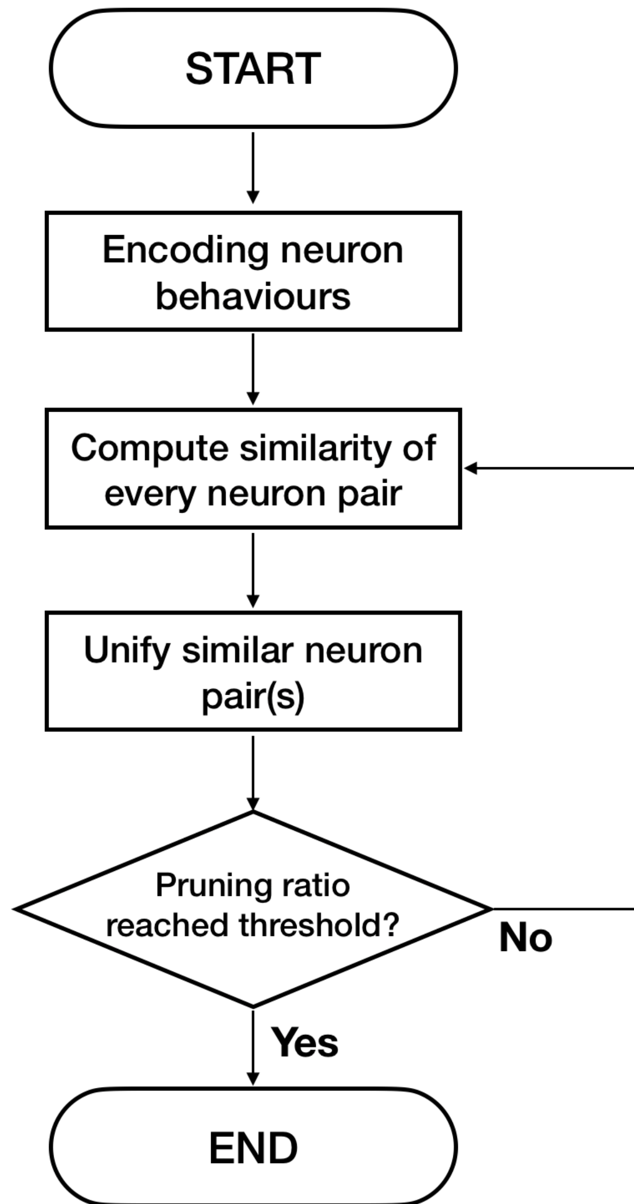


Figure III.1: The flow chart of NU. We encode the neuron behaviors by feeding several images into a pretrained DNN model, compute behavioral similarity between the neurons, and unify the similar ones. These procedures are repeated until we have unified as many neurons as we want.

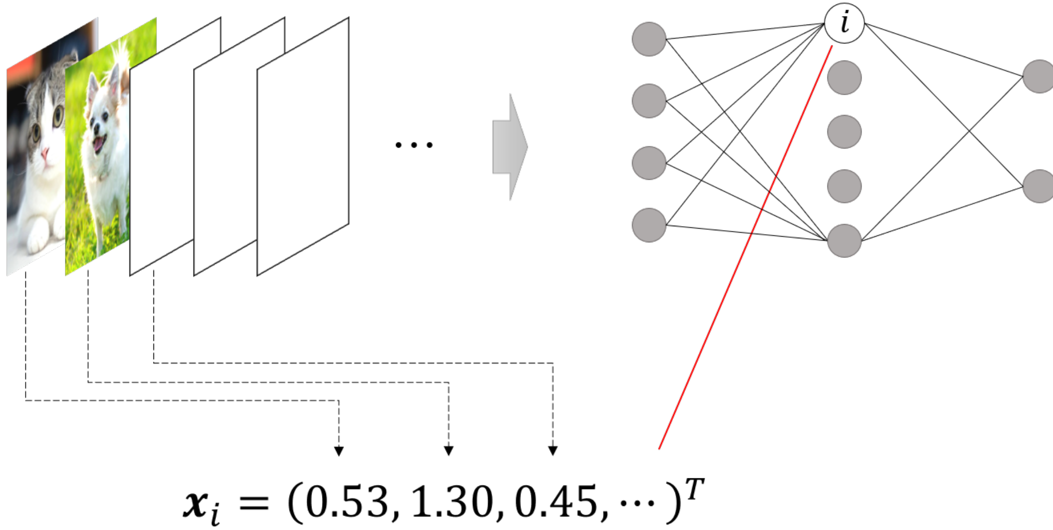


Figure III.2: The conceptual drawing of neuron behavior encoding. When we feed several images into a DNN model, each neuron obtains a behavioral vector composed of their own outputs.

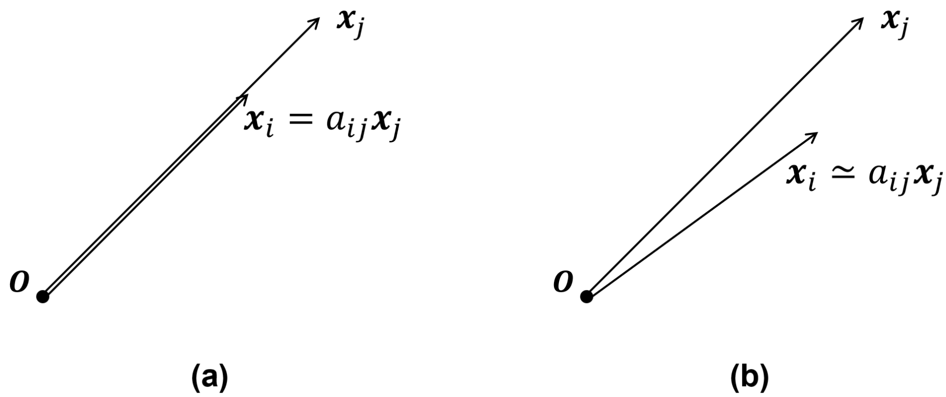


Figure III.3: The definition of behavioral similarity of the neurons. (a) The case of the neurons having the same behaviors. Having the same behaviors means that their behavioral vectors are linearly dependent on each other. (b) The case of the neurons having similar behaviors. Their behavioral vectors are linearly independent, although one of them can be well reconstructed by the other.

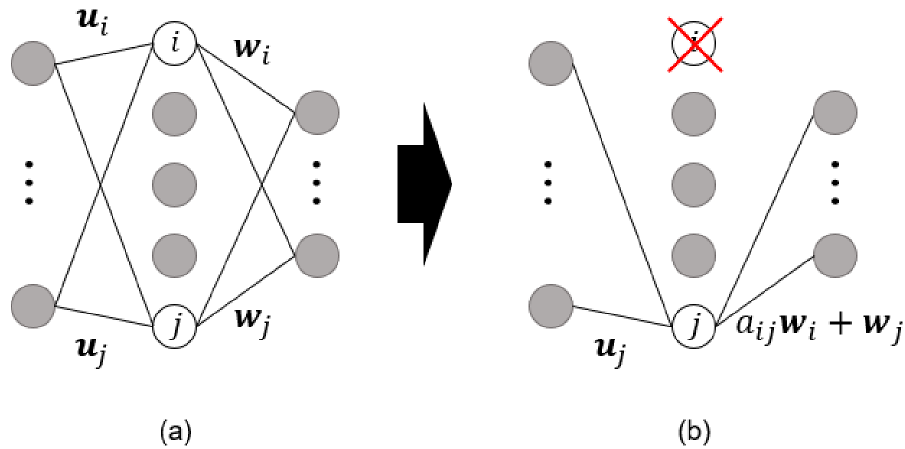


Figure III.4: The illustration of neuron unification. (a) The initial state. (b) After merging the $i$-th neuron into the $j$-th one. If their behavioral vectors are similar enough, we can unify them with small error.

4) If the number of neurons has become small enough, terminate iteration. Otherwise, go to 2).

In addition, we can optionally conduct extra reconstruction to preserve the accuracy even better. With extra reconstruction, we can have two or more neurons to emulate the pruned neuron's behavior, which results in making the error caused by pruning even smaller.

In this Section, we first explain the case of single reconstruction, and then explain the case we conduct extra reconstructions. We also show how to apply our method to the convolutional layers.

## 2.1 How to encode the neuron behaviors and unify the neurons having the same behavior

We first encode the neuron behavior by feeding several images into the model that we want to prune. Fig. III.2 is the conceptual drawing of neuron behavior encoding. For a single image, the $i$-th neuron outputs a scalar value. For $d$ input images, the output becomes a vector $\boldsymbol{x}_i \in \mathbb{R}^d$. We call it the *behavioral vector* of the $i$-th neuron.

If there are a pair of neurons with the same behaviors, we can unify them without error. Here, having the same behaviors means that their behavioral vectors are linearly dependent on each other, as shown in Fig. III.3 (a). We show an example

below.

Let $n$ and $n'$ denote the numbers of neurons in a layer and the next layer (intermediate layer and right one in Fig. III.4 (a), respectively), $\mathcal{I} = \{1, \cdots, n\}$ denote the set of neuron indices, $\boldsymbol{w}_i \in \mathbb{R}^{n'}$ denote the weights going from the $i$-th neuron to the ones in the next layer. The forward propagation is described by

$$Y = \boldsymbol{x}_i \boldsymbol{w}_i^\top + \boldsymbol{x}_j \boldsymbol{w}_j^\top + \sum_{k \in \mathcal{I} \setminus \{i,j\}} \boldsymbol{x}_k \boldsymbol{w}_k^\top, \tag{III.1}$$

where $Y \in \mathbb{R}^{d \times n'}$ denotes the inner activation levels in the next layer.

If $\boldsymbol{x}_i = a_{ij} \boldsymbol{x}_j$ holds for some $a_{ij}$, we can unify the $i$-th and the $j$-th neurons without error. As shown in Fig. III.4 (b), we prune the $i$-th neuron and update the $j$-th one's weights going to the next layer as

$$\boldsymbol{w}_j \leftarrow a_{ij} \boldsymbol{w}_i + \boldsymbol{w}_j. \tag{III.2}$$

Then, we can rewrite Eq. (III.1) as

$$Y = \boldsymbol{x}_j \left( a_{ij} \boldsymbol{w}_i^\top + \boldsymbol{w}_j^\top \right) + \sum_{k \in \mathcal{I} \setminus \{i,j\}} \boldsymbol{x}_k \boldsymbol{w}_k^\top. \tag{III.3}$$

Eq. (III.1) and Eq. (III.3) are equivalent because $\boldsymbol{x}_i = a_{ij} \boldsymbol{x}_j$ holds, which means the original $Y$ is preserved. This is how we reconstruct the outputs of the pruned neurons.

## 2.2 The case of the neurons having linearly independent behavioral vectors

As above, in the case of the neurons having linearly dependent behavioral vectors, they can be unified without error. Although, it rarely happens that those behavioral vectors are linearly dependent. In the case of unifying the neurons having linearly independent but similar behavioral vectors as shown in Fig. III.3 (b), we accept some error and make it as small as possible. In this case, we first approximate $\boldsymbol{x}_i$ by a vector which is linearly dependent on $\boldsymbol{x}_j$:

$$\boldsymbol{x}_i \simeq a_{ij} \boldsymbol{x}_j. \tag{III.4}$$

We regard that $a_{ij} \boldsymbol{x}_j$ is the behavioral vector of the $i$-th neuron so that we can conduct unification in the same manner with Eq. (III.2).

Here is a question. How to determine $a_{ij}$ in Eq. (III.4)? In order to minimize the error in the next layer, we have to minimize the error of $Y$. This can be formalized as

$$\begin{aligned} a_{ij}^* &= \underset{a_{ij}}{\operatorname{argmin}} \left\| (\boldsymbol{x}_i - a_{ij}\boldsymbol{x}_j)\, \boldsymbol{w}_i^\top \right\|_{\mathrm{F}}^2 \\ &= \underset{a_{ij}}{\operatorname{argmin}} \sum_{k=1}^{d} \sum_{l=1}^{n'} \left( \left( x_{i(k)} - a_{ij} x_{j(k)} \right) w_{i(l)} \right)^2 \\ &= \underset{a_{ij}}{\operatorname{argmin}} \sum_{l=1}^{n'} w_{i(l)}^2 \sum_{k=1}^{d} \left( x_{i(k)} - a_{ij} x_{j(k)} \right)^2 \\ &= \underset{a_{ij}}{\operatorname{argmin}} \left\| \boldsymbol{w}_i \right\|^2 \left\| \boldsymbol{x}_i - a_{ij}\boldsymbol{x}_j \right\|^2 , \end{aligned} \tag{III.5}$$

where $x_{i(k)}$ denotes the $k$-th entry of $\boldsymbol{x}_i$ and $w_{i(l)}$ denotes the $l$-th entry of $\boldsymbol{w}_i$. We can omit $\|\boldsymbol{w}_i\|^2$ in Eq. (III.5) as it is a constant. Then, Eq. (III.5) can be rewritten as

$$a_{ij}^* = \underset{a_{ij}}{\operatorname{argmin}} \left\| \boldsymbol{x}_i - a_{ij}\boldsymbol{x}_j \right\|^2 . \tag{III.6}$$

After all, we have to compute the orthogonal projection of $\boldsymbol{x}_i$ onto $\boldsymbol{x}_j$. Thus, we have

$$a_{ij}^* = \frac{\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle}{\|\boldsymbol{x}_j\|^2}. \tag{III.7}$$

If $\boldsymbol{x}_i$ and $a_{ij}^*\boldsymbol{x}_j$ are similar enough, it means the $j$-th neuron can emulate the behavior of the $i$-th one well, and the error caused by this unification will be small.

### 2.3 Criteria for selecting neurons to be unified

We know how to unify a given pair of neurons. Although, we have yet to know how to select a neuron pair to be unified out of many possible pairs.

As already discussed, we should minimize the error in the next layer. Therefore, we define the similarity of the $i$-th and the $j$-th neurons by the error caused by unifying them:

$$s(i,j) = \left\| \left( \boldsymbol{x}_i - a_{ij}^*\boldsymbol{x}_j \right) \boldsymbol{w}_i^\top \right\|_{\mathrm{F}}^2 . \tag{III.8}$$

Let $\mathcal{Q}$ denote the set composed of the tuples of the unified neurons' indices. For example, $(i,j) \in \mathcal{Q}$ means that the $i$-th neuron has been merged into the $j$-th one.

Then, we have to solve the following optimization problem:

$$\mathcal{Q}^* = \underset{\mathcal{Q}}{\operatorname{argmin}} \left\| \sum_{(i,j)\in\mathcal{Q}} \left(\boldsymbol{x}_i - a^*_{ij}\boldsymbol{x}_j\right) \boldsymbol{w}_i^\top \right\|_{\mathrm{F}}^2 \quad \text{subject to } |\mathcal{Q}| = q, \tag{III.9}$$

where $|\mathcal{Q}|$ denotes the number of elements in $\mathcal{Q}$ and $q$ denotes the desired number of the neuron pairs to be unified.

Because solving Eq. (III.9) directly is computationally intensive, we perform simplification by using the following theorem. Note that Fig. III.5 is helpful to understand this theorem.

**Theorem III.1** *Let $\Omega$ denote the set of indices and $\Phi_i \in \mathbb{R}^{\alpha\times\beta}$ for each $i \in \Omega$. Then,*

$$\left\| \sum_{i\in\Omega} \Phi_i \right\|_F^2 \leq |\Omega| \sum_{i\in\Omega} \|\Phi_i\|_F^2. \tag{III.10}$$

**Proof of Theorem III.1** *Let $\phi_{i(k,l)}$ denote the $(k,l)$ entry of $\Phi_i$. The LHS of Eq. (III.10) can be rewritten as*

$$\left\| \sum_{i\in\Omega} \Phi_i \right\|_F^2 = \sum_{k=1}^{\alpha}\sum_{l=1}^{\beta} \left( \sum_{i\in\Omega} \phi_{i(k,l)} \right)^2 \tag{III.11}$$

*Cauchy-Schwartz inequality is given by*

$$\left( \sum_{i\in\Psi} \eta_i\theta_i \right)^2 \leq \left( \sum_{i\in\Psi} \eta_i^2 \right) \left( \sum_{i\in\Psi} \theta_i^2 \right), \tag{III.12}$$

*where $\Psi$ denotes an arbitrary set of indices. By substituting $\Psi = \Omega$, $\eta_i = 1$, and $\theta_i = \phi_{i(k,l)}$ to Eq. (III.12), we get*

$$\left( \sum_{i\in\Omega} \phi_{i(k,l)} \right)^2 \leq |\Omega| \sum_{i\in\Omega} \phi_{i(k,l)}^2. \tag{III.13}$$

*Therefore, we have*

$$\begin{aligned} \left\| \sum_{i\in\Omega} \Phi_i \right\|_F^2 &= \sum_{k=1}^{\alpha}\sum_{l=1}^{\beta} \left( \sum_{i\in\Omega} \phi_{i(k,l)} \right)^2 \\ &\leq \sum_{k=1}^{\alpha}\sum_{l=1}^{\beta} |\Omega| \sum_{i\in\Omega} \phi_{i(k,l)}^2 = |\Omega| \sum_{i\in\Omega} \left( \sum_{k=1}^{\alpha}\sum_{l=1}^{\beta} \phi_{i(k,l)}^2 \right) = |\Omega| \sum_{i\in\Omega} \|\Phi_i\|_F^2. \end{aligned} \tag{III.14}$$

*(Q.E.D)*

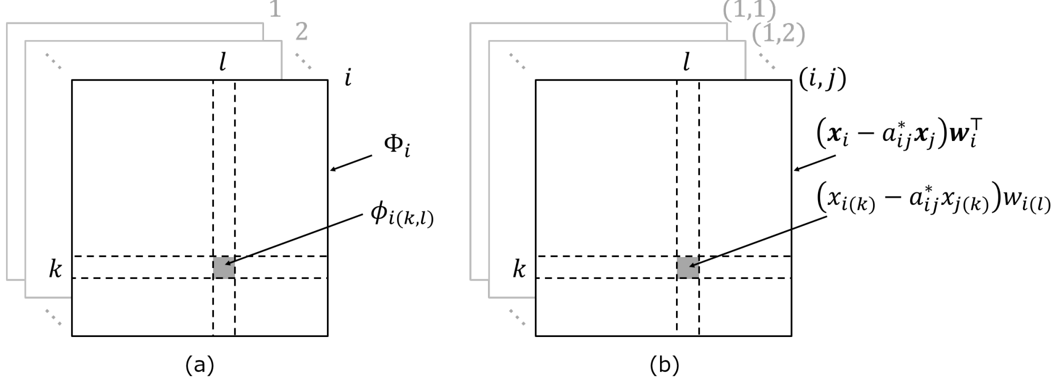


Figure III.5: (a) For each $(k, l)$, Eq. (III.13) holds true based on Cauchy-Schwartz inequality. Therefore, we have Eq. (III.14). (b) By substituting $\Phi_i = \left(\boldsymbol{x}_i - a^*_{ij}\boldsymbol{x}_j\right)\boldsymbol{w}_i^\top$ and $\Omega = \mathcal{Q}$, the same discussion holds true and we get Eq. (III.15).

By substituting $\Phi_i = \left(\boldsymbol{x}_i - a^*_{ij}\boldsymbol{x}_j\right)\boldsymbol{w}_i^\top$ and $\Omega = \mathcal{Q}$ in Eq. (III.10), we get

$$\begin{aligned}\left\|\sum_{(i,j)\in\mathcal{Q}}\left(\boldsymbol{x}_i - a^*_{ij}\boldsymbol{x}_j\right)\boldsymbol{w}_i^\top\right\|_{\mathrm{F}}^2 &\leq |\mathcal{Q}|\sum_{(i,j)\in\mathcal{Q}}\left\|\left(\boldsymbol{x}_i - a^*_{ij}\boldsymbol{x}_j\right)\boldsymbol{w}_i^\top\right\|_{\mathrm{F}}^2 \\ &= |\mathcal{Q}|\sum_{(i,j)\in\mathcal{Q}} s(i,j).\end{aligned} \tag{III.15}$$

We minimize this upper bound of Eq. (III.15). We then have the following problem.

$$\begin{aligned}&\mathcal{Q}'^* = |\mathcal{Q}|\operatorname*{argmin}_{\mathcal{Q}}\sum_{(i,j)\in\mathcal{Q}} s(i,j) = \operatorname*{argmin}_{\mathcal{Q}}\sum_{(i,j)\in\mathcal{Q}} s(i,j), \\ &\text{subject to } |\mathcal{Q}| = q.\end{aligned} \tag{III.16}$$

Note that we omitted $|\mathcal{Q}|$ in Eq. (III.16), because it is a constant as we have the constraint $|\mathcal{Q}| = q$.

We solve it in a greedy fashion. We select the neurons one by one based on the cost function $f$, where we have $\mathcal{Q}'^* = \operatorname{argmin}_{\mathcal{Q}} f(\mathcal{Q})$:

$$f(\mathcal{Q}) = \sum_{(i,j)\in\mathcal{Q}} s(i,j). \tag{III.17}$$

## 2.4 The case of unifying neurons that have already been unified

Assume that we have unified the $i$-th and the $j$-th neurons, and we no more have the $i$-th one, as shown in Fig. III.6 (a). As we have $\mathcal{Q} = \{(i,j)\}$, the cost function

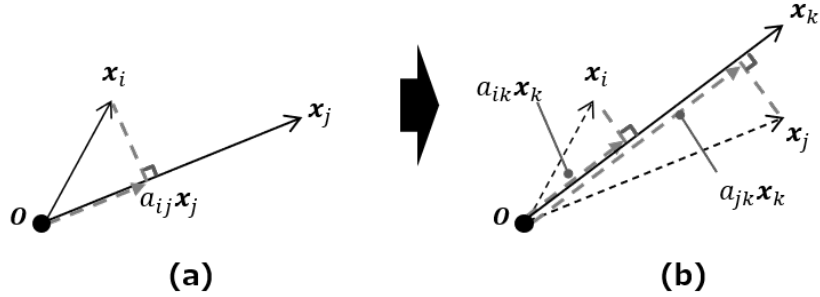


Figure III.6: The case of unifying the neurons that were already unified with other neurons. (a) The $i$-th and the $j$-th neurons have been unified, and the $j$-th one emulates the $i$-th one's behavior. (b) The $j$-th neuron is further merged into the $k$-th one. Then, the $k$-th one emulates both the $i$-th and the $j$-th ones' behavior.

$f$ is currently given by

$$f(\mathcal{Q}) = s(i, j). \tag{III.18}$$

What happens if we next merge the $j$-th neuron into the $k$-th one as shown in Fig. III.6 (b)? We then have $\mathcal{Q} = \{(i, j), (j, k)\}$. It means that the $j$-th neuron still emulates the $i$-th one, however, the $j$-th one has already been removed. In order to avoid this contradiction, we let the $k$-th neuron emulate both the $i$-th and the $j$-th ones. Therefore, after merging the $j$-th neuron into the $k$-th one, we will have $\mathcal{Q} = \{(i, k), (j, k)\}$ instead of $\mathcal{Q} = \{(i, j), (j, k)\}$, and the cost function will be

$$f(\mathcal{Q}) = s(i, k) + s(j, k). \tag{III.19}$$

Thus, $k$ should be selected so that this new error will be minimized.

## 2.5 Problem formalization based on graph theory

For better understanding, we formalize the problem of neuron selection based on graph theory, then we show the algorithm to solve it.

The problem of selecting the neurons to be unified is equivalent to the problem of creating a forest having minimum cost out of a complete symmetric digraph. Let $\mathcal{G}$ denote a graph defined by

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \tag{III.20}$$

where $\mathcal{V}$ and $\mathcal{E}$ denote the the sets composed of vertices and edges, respectively. The vertices and the edges correspond to neurons and possible unifications, respectively.

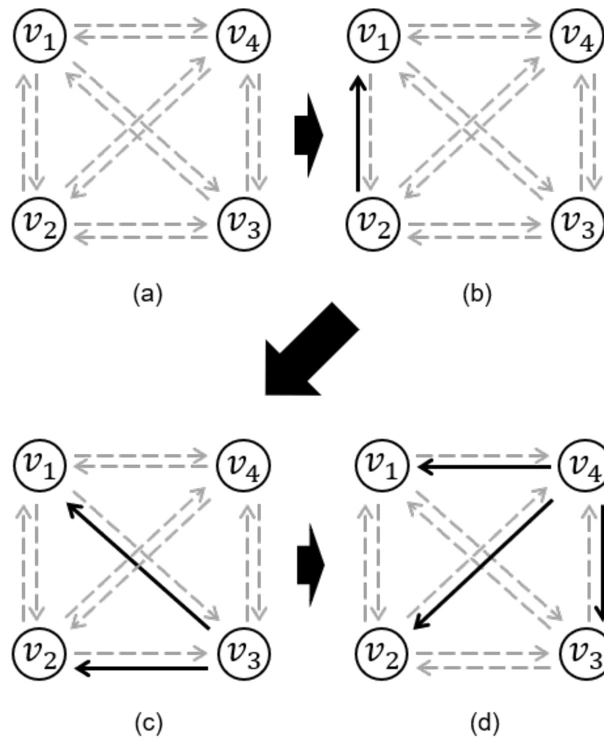


Figure III.7: Illustration of procedures of Neuro-Unification on a graph. (a) Initial state. (b) $(v_1, v_2)$ has been added to $\mathcal{E}'$ (c) $(v_2, v_3)$ has been added to $\mathcal{E}'$ and $(v_1, v_3)$ has replaced $(v_1, v_2)$. (d) $(v_3, v_4)$ has been added to $\mathcal{E}'$, $(v_1, v_4)$ and $(v_2, v_4)$ replaced $(v_1, v_3)$ and $(v_2, v_3)$, respectively.

Since the graph is a complete symmetric digraph, we have

$$\mathcal{E} = \mathcal{V} \times \mathcal{V} \setminus \{(v_i, v_i) | v_i \in \mathcal{V}\}. \tag{III.21}$$

Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ denote the forest we want to create, where $\mathcal{V}' \subset \mathcal{V}$ and $\mathcal{E}' \subset \mathcal{E}$. If $(v_i, v_j) \in \mathcal{E}'$, it means that the neuron represented by $v_i$ has been merged into the one represented by $v_j$. Then, we have the following optimization problem:

$$\underset{\mathcal{E}'}{\operatorname{argmin}} \sum_{(v_i, v_j) \in \mathcal{E}'} c(v_i, v_j) \quad \text{subject to} \quad |\mathcal{E}'| = q, \tag{III.22}$$

where $c(v_i, v_j)$ denotes the cost of $(v_i, v_j)$ and is given by Eq. (III.8).

Besides, we have a constraint that the height of trees composing $\mathcal{G}'$ must be 1. For example, assume that we have the following tree with height of 2:

$$\mathcal{V}' = \{v_1, v_2, v_3\}, \tag{III.23}$$

$$\mathcal{E}' = \{(v_1, v_2), (v_2, v_3)\}. \tag{III.24}$$

This means that $v_1$ has been merged into $v_2$, which has already been merged into $v_3$. This is an contradiction that we mentioned in Sec. 2.4. Therefore, the tree height must be 1.

All that is left is to solve the combinatorial optimization problem. For obtaining the solution efficiently, we use a greedy algorithm shown in Algorithm III.1.

### 2.6 Extra reconstruction

In Neuro-Unification, behavior of a pruned neuron is emulated by another one. However, it is possible to use 2 or more neurons for emulating the pruned one's behavior.

When merging the $i$-th neuron into the $j$-th one, $\boldsymbol{x}_i$ is reconstructed by using $a^*_{ij}\boldsymbol{x}_j$. The reconstruction residual is given by $\boldsymbol{r}_i = \boldsymbol{x}_i - a^*_{ij}\boldsymbol{x}_j$. In order to make this residual even smaller, we have the $k$-th neuron to reconstruct $\boldsymbol{r}_i$. This can be described by

$$b^*_{ik} = \underset{b_{ik}}{\operatorname{argmin}} \left\| \boldsymbol{r}_i - b_{ik}\boldsymbol{x}_k \right\|^2, \tag{III.25}$$

$$\boldsymbol{w}_k \leftarrow b^*_{ik}\boldsymbol{w}_i + \boldsymbol{w}_k. \tag{III.26}$$

In the same manner, we can pick yet another neuron for another extra reconstruction. By repeating this procedure, the residual of $\boldsymbol{x}_i$ gets even smaller, which results in reducing the errors of the inner activation levels in the next layer.

---

**Algorithm III.1**

---

**Input:** Complete symmetric digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, controlled parameter $q$.
**Output:** Forest $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$.
**Definition:** Initial and current edge cost $c(\cdot,\cdot)$ and $c'(\cdot,\cdot)$.
$\mathcal{V}' = \emptyset$, $\mathcal{E}' = \emptyset$.
**for each** $(v_i, v_j) \in \mathcal{E}$ **do**
  $c'(v_i, v_j) \leftarrow c(v_i, v_j)$
**end for**
**while** $|\mathcal{E}'| < q$ **do**
  $(i^*, j^*) = \text{argmin}_{(i,j)\in\mathcal{E}}\, c'(v_i, v_j)$
  $\mathcal{V}' \leftarrow \mathcal{V}' \cup \{v_{i^*}, v_{j^*}\}$, $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{(v_{i^*}, v_{j^*})\}$
  **for each** $(v_k, v_{i^*}) \in \mathcal{E}'$ **do**
    $\mathcal{E}' \leftarrow (\mathcal{E}' \setminus \{(v_k, v_{i^*})\}) \cup \{(v_k, v_{j^*})\}$
  **end for**
  **for each** $(v_k, v_{i^*}) \in \mathcal{E}$ **do**
    $\mathcal{E} \leftarrow \mathcal{E} \setminus \{(v_k, v_{i^*})\}$
  **end for**
  **for each** $(v_{i^*}, v_k) \in \mathcal{E}$ **do**
    $c'(v_{i^*}, v_k) \leftarrow c(v_{i^*}, v_k)$
  **end for**
  **for each** $(v_{j^*}, v_k) \in \mathcal{E}$ **do**
    $c'(v_{j^*}, v_k) \leftarrow c(v_{j^*}, v_k)$
    **for each** $l$ fulfilling $(v_l, v_{j^*}) \in \mathcal{E}'$ **do**
      $c'(v_{j^*}, v_k) \leftarrow c'(v_{j^*}, v_k) + c(v_l, v_k) - c(v_l, v_{j^*})$
    **end for**
  **end for**
**end while**

---

It should be noted that if we perform extra reconstruction infinite number of times, it would be equivalent to reconstructing $\boldsymbol{x}_i$ from all other $\boldsymbol{x}$-s with least squares method. This extension will be discussed in Part IV.

## 2.7 Applying Neuro-Unification to convolutional layers

Neuro-Unification can be applied to convolutional layers with minor modifications. By expanding the feature maps and the kernels into matrices, we can deal with the convolutional layers in the same manner with the fully connected layers.

Same with *im2col* function implemented in cuDNN [53], we can describe the sliding window operation in the convolutional layers by using a matrix multiplication, as shown in Fig. III.8. Let $n$ and $n'$ denote the numbers of input channels and output channels, $g$ denote the width and the height of the weight tensor, and $h$ denotes the width and the height of the feature maps (Although we assume squared shapes for feature maps and kernels here, they need not be squared.). The sliding window operations with a $d \times n \times h \times h$ tensor, which denotes the feature maps corresponding to $d$ input images, and a $n' \times n \times g \times g$ tensor, which denotes the kernels, can be alternatively written as

$$Y = \sum_{i=1}^{ng^2} \boldsymbol{x}_i \boldsymbol{w}_i^\top, \tag{III.27}$$

where $\boldsymbol{x}_i \in \mathbb{R}^{dh^2}$ denotes a part of the reshaped input feature map, and $\boldsymbol{w}_i \in \mathbb{R}^{n'}$ denotes a part of the reshaped weight tensor. Thus, we can regard that a convolutional layer that has $n$ input channels is equivalent to a fully connected layer that has $ng^2$ neurons. Pruning the $i$-th channel in the convolutional layer is equivalent to pruning the $(ig^2+1)$-th to the $((i+1)g^2)$-th neurons in this converted form.

## 2.8 Relevant methods

We show several methods that are relevant to NU and have some theoretical comparison.

### Data-free Parameter Pruning (DPP)

DPP [18] is a method for compressing fully connected layers. DPP unifies the neurons with similar incoming weights. Let $\boldsymbol{u}_i$ denote the vector composed of the weights coming to the $i$-th neuron from the ones in the previous layer. If $\boldsymbol{u}_i \simeq a_{ij}\boldsymbol{u}_j$, they prune the $i$-th neuron and update $\boldsymbol{w}_j$ in the same manner with Eq. (III.2).

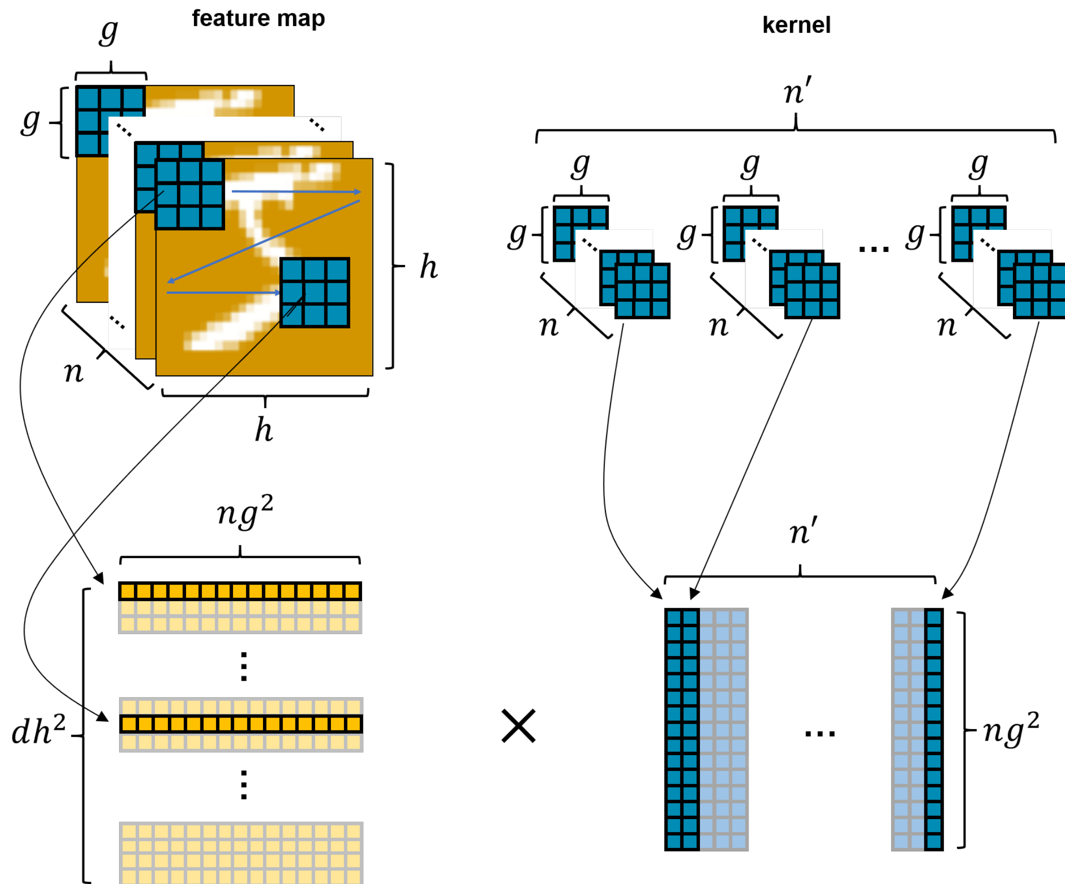


Figure III.8: The illustration of *im2col* function implemented in cuDNN library. The kernel filters (corresponding to the each single input channel) is reshaped into vertical vectors, and the sub-matrices of the feature maps are reshaped into horizontal vectors. After reshaping, we can describe the sliding window operation in the convolutional layers by a matrix multiplication.

we expect that our proposed method will perform better than DPP, because DPP does not evaluate the influence of activation functions while ours does. The assumption of DPP is that if two neurons have similar weights coming from the previous layer, their outputs should also be similar. However, the similarity of incoming weights does not guarantee that their outputs are also similar due to non-linear activation such as ReLU.

Besides, DPP can use only one neuron to emulate a removed neuron, while our NU can use as many neurons as we want for emulating the removed one.

**Oracle Pruning (OP)**

OP [22] conducts channel-level pruning for convolutional layers. OP computes the saliency of each channel based on first derivative information of the cost function, and prunes the least salient ones.

OP only prunes the neurons and do not conduct reconstruction. Therefore, pruning with OP may result in significant accuracy degradation.

Besides, differently from our method, OP's channel selection criteria is designed by heuristics, therefore, this criteria is not promising for preserving the model performances.

**ThiNet**

ThiNet [9] is a pruning method for convolutional layers. ThiNet prunes channels to be pruned in a greedy fashion so that the output error in that layer stay as small as possible, then reconstructs outputs by least squares method.

Fig. III.9 illustrates a part of weight tensor that goes from all the input channels to a single output channel. In reconstruction step, ThiNet multiplies the whole weights in each channel by a common coefficient such as $W_i \leftarrow \alpha_i W_i$, where $W_i$ denotes the $i$-th channel of the kernel. On the other hand, our method updates each weight independently such as $w_{i(j,k)} \leftarrow \alpha w_{1(1,1)} + \beta w_{1(1,2)} + \cdots + \gamma w_{n(g,g)}$, where $w_{i(j,k)}$ denotes the $(j, k)$ entry in the $i$-th channel. Therefore, NU should be able to compensate the error caused by pruning much better.

Another remarkable difference from NU is that ThiNet selects the channels to be pruned based on the error before reconstruction. Therefore, channel selection result of ThiNet may not be proper after reconstruction. On the other hand, NU selects the neuron pairs based on the error after reconstruction.

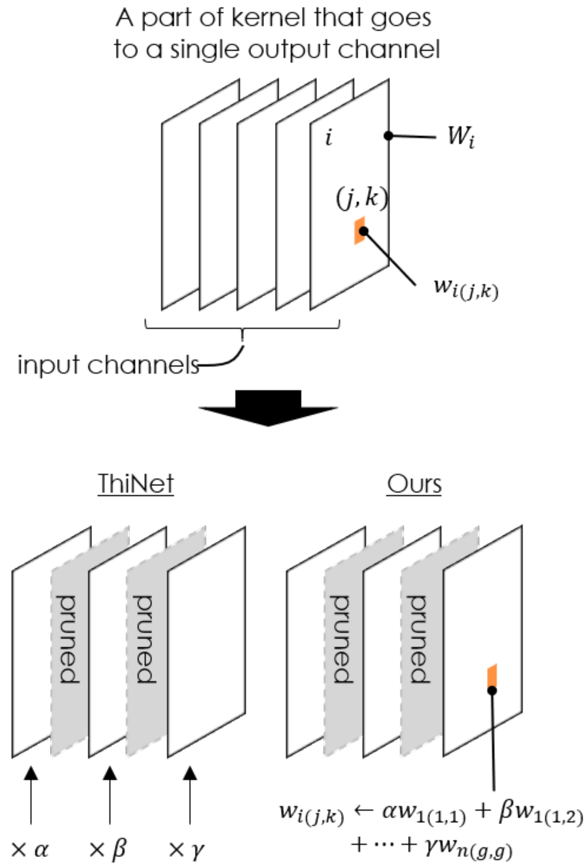


Figure III.9: Theoretical comparison of NU and ThiNet. This figure illustrates a part of weight tensor that goes from all the input channels to a single output channel. In the reconstruction step, ThiNet multiplies the whole weights in each channel by a common coefficient. On the other hand, NU can tune weights independently from other weights that belong to the same channel.

## 3 Experiments

We evaluate NU in experiments with VGG16 [2] on ImageNet [54] and ResNet-56 [11] on cifar-10 [55], and compare the results with some existing methods.

### 3.1 Datasets

**ImageNet**

ImageNet is a large scale dataset for 1,000 classes image classification [54]. It has approximately 1.2M images for training, 50K images for validation, and 100K images for testing. Following the former works, we used the validation images as the test dataset, and did not use the official test images in our experiments. As each image has different resolution, we resized them so that the shorter side would become 256 pixels. Then, $224 \times 224$ random crop was applied to the training images, and $224 \times 224$ center crop was applied to the test images. The random horizontal flip was applied to the training images. We randomly selected 5K training images, and used them for encoding neuron behavior.

**CIFAR-10**

CIFAR-10 is a dataset for 10 class image classification [55]. It has 50K images for training and 10K images for testing. All the images have $32 \times 32$ resolution. The training images were padded by 4 pixels at each side and $32 \times 32$ random crop was applied. Random horizontal flip was applied to the training images. The test images were used as they were. We randomly selected 5K images from training dataset for pruning.

### 3.2 Models

**VGG16**

VGG16 is a model that has 16 weight layers, including 13 convolutional layers and 3 fully connected layers. We used the original VGG16 model that was trained with ImageNet dataset. The convolutional layers are composed of 5 blocks that have 2 or 3 layers. For convenience, we call the $X$-th layer of the $Y$-th block *ConvY-X*. For fully connected layers, we call such as *FC1* and *FC2*. Architecture details are mentioned in Appendix A.1.

**ResNet-56**

ResNet-56 is a model having *identity shortcuts* that makes it possible to train a

Table III.1: The results of VGG16 on ImageNet, fully connected layers. In this Table, "($k$)" of "NU($k$)" denotes the times of extra reconstruction. The baseline top-5 accuracy is 0.895.

| Params# | NU(0) | NU(1) | NU(10) | DPP |
|---|---|---|---|---|
| ×1/2 | 0.854 | 0.874 | **0.879** | 0.856 |
| ×1/3 | 0.781 | 0.849 | **0.864** | 0.763 |

very deep models stably and effectively. ResNet-56 has 54 convolutional layers and 1 fully connected layer, and was trained with CIFAR-10 dataset. ResNet-56 has 3 blocks that have 18 convolutional layers. We call the $X$-th layer of the $Y$-th block is called *ConvY-X*. Architecture details are mentioned in Appendix A.2.

## 3.3 Experiments with VGG16 on ImageNet

### 3.3.1 Setups

We pruned *FC1* and *FC2* layers. The pruning ratios for the two layers were set evenly, and were tuned so that the total number of the weights would become 1/2 and 1/3. The rest of the setups were as below: the momentum was set to 0.9, the minibatch size was set to 128, the gradients were computed by SGD with cross entropy loss.

In the experiments, we did not conduct retraining after pruning. As one of our motivations is to save time for retraining, we wanted to see how well we could maintain accuracy even without retraining.

### 3.3.2 Results for fully connected layers

Table III.1 shows the results. We only see a marginal difference between NU without extra reconstructions and DPP at ×2 compression ratio. With extra reconstructions, our method easily outperforms DPP. At ×3 compression ratio, NU is obviously better even without extra reconstructions. After 10 extra reconstructions, we only suffer 0.031 accuracy drop, while DPP suffers 0.132.

We also briefly report the computational time of Neuro-Unification. When $n = 4,096$, where $n$ is the number of neurons, it took about 177 seconds computing with single thread of Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz. We believe this is fast and practical enough.

Table III.2: The results of VGG16 on ImageNet, convolutional layers. The baseline top-5 accuracy is 0.895.

| FLOPs | NU(0) | NU(1) | NU(10) | OP | ThiNet |
|---|---|---|---|---|---|
| $\times 1/2$ | 0.375 | 0.802 | **0.845** | 0.024 | 0.245 |
| $\times 1/3$ | 0.097 | 0.616 | **0.729** | 0.006 | 0.022 |

### 3.3.3 Results for convolutional layers

For convolutional layers, we set the pruning ratios in Conv1, Conv2, Conv3 and Conv4 to 6.5 : 6 : 6 : 5.5. We do not prune the layers in Conv5, because we found out that the layers in Conv5 are not redundant, and pruning those layers results in significant accuracy degradation.

Table III.2 shows the result. Our method, especially after 10 times of extra reconstructions, easily outperforms the other methods. At $\times 2$ speed-up, we only suffer 5.0% accuracy drop even though the pruned model has not been retrained. ThiNet is better than OP, however, much worse than NU even without extra reconstructions. This is because Neuro-Unification takes more sophisticated approach for reconstruction than that of ThiNet, as we mention in Fig. III.9.

### 3.3.4 Ablation study

For evaluating how important it is to encode neuron behavior accurately, we set the number of images used for neuron behavior encoding to 128, 1024, and 5000, and applied NU to fully connected layers of VGG16. We did not conduct extra reconstructions in this experiment.

See Table III.3. The trend is that the more images are used for encoding the neuron behaviors, the better the performance of NU becomes. It implies that when the number of images is not enough, the behavioral vectors cannot describe actual neuron behavior accurately, and performance of NU becomes poor. Thus, it is crucial to use many images enough to describe neuron behavior well.

## 3.4 Experiments with ResNet-56 on cifar-10

### 3.4.1 Setups

We conducted experiments with pretrained ResNet-56 on cifar-10 taken from [56]. In this experiment, we did not only pruned but also retrained the pruned models for 100 epochs at 0.01 learning rate and another 100 epochs at 0.001 learning rate, and

Table III.3: The results of ablation study, VGG16 on ImageNet, fully connected layers. Performances of NU with different number of images for neuron behavior encoding. The baseline top-5 accuracy is 0.895.

| Params | Images# for encoding | | |
|---|---|---|---|
| | 128 | 1024 | 5000 |
| ×1/2 | 0.704 | 0.769 | 0.854 |
| ×1/3 | 0.388 | 0.537 | 0.781 |

Table III.4: ResNet-56 on cifar-10. The baseline top-1 accuracy is 0.934.

| FLOPs | Retraining | NU(0) | NU(1) | NU(10) | ThiNet |
|---|---|---|---|---|---|
| ×1/2 | No | 0.655 | 0.810 | 0.834 | 0.827 |
| ×1/2 | Yes | 0.927 | 0.924 | 0.925 | 0.923 |

evaluate accuracy before and after retraining. The rest of the setups were as below: the momentum was set to 0.9, the minibatch size was set to 128, the gradients were computed by SGD with cross entropy loss.

We performed pruning on convolutional layers since ResNet has only one fully connected layer. ResNet-56 has 27 units that have 2 convolutional layers, such as the one shown in Fig. III.10. In each unit, input tensor is propagated forward as is in one path, and processed in convolutional layers in the other path, and both of them are added at the end of block. At this addition, two inputs must have the same dimension, which means we should not perform pruning in the layers having branched paths (This problem will be discussed in detail in Part VI). Therefore, we conducted pruning on the layers that do not have branched paths. We set pruning ratio in each layer constantly. As we think the results in Section 3.3.3 is enough to conclude that NU is better than OP, we compared NU and ThiNet.

### 3.4.2 Results

Table III.4 shows the result. Consistently with the other experiments, our method outperforms ThiNet. After retraining, the accuracy of the model pruned with NU was from 0.924 to 0.927. This is competitive with ResNet-32 in [11] (ResNet-32 accuracy is 0.925), while our pruned model has fewer FLOPs than ResNet-32 by approximately 10%. It should be noted that the performance difference of NU and ThiNet is not significant after retraining. However, we still have the room to improve NU, as we will discuss in Part. IV.

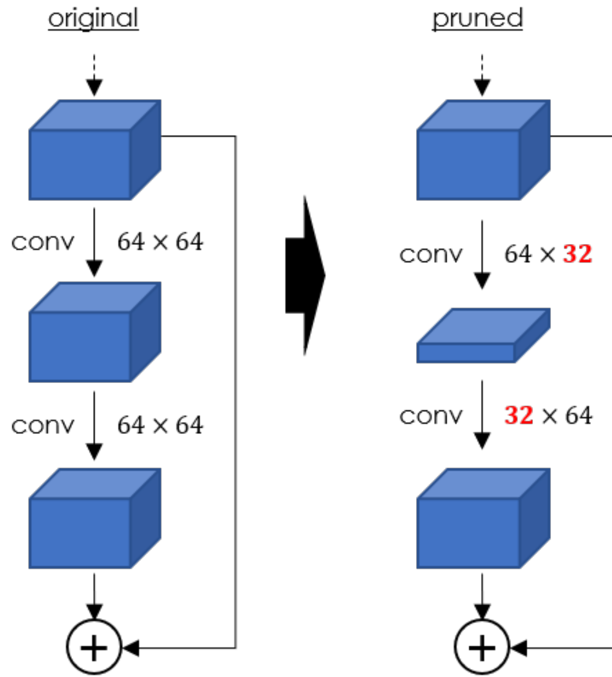


Figure III.10: Illustration of a part of ResNet architecture. Input tensor is propagated forward as is in one path, and processed in convolutional layers in the other path, and both of them are added at the end of block.

## 4 Summary of Part III

In Part III, we proposed Neuro-Unification (NU), a method for pruning DNN models. NU is designed to preserve the original performance of the model while reducing the weights and the FLOPs. It finds a pair of neurons having similar behaviors, prunes one of them, and updates the weights of the remaining neuron so as to compensate the damage of pruning. With extra reconstruction, the error becomes even smaller. In the experiments, the proposed methods outperformed several existing methods, and the effectiveness of NU was verified.

There is a weakness of NU. One is that the neuron pair to be unified is determined based on only their behavioral similarity. However, as we usually conduct extra reconstruction, it is better to select the neurons to be pruned based on error after extra reconstruction. In Part IV, this weakness of NU will be discussed and the method that overcomes it will be proposed.

# Part IV
# Reconstruction Error Aware Pruning

## 1 Introduction

In Part III, we presented Neuro-Unification (NU). The idea of NU is to unify a pair of neurons having similar behaviors, which makes the error small and preserves the model accuracy well. However, we have noticed there is the room to improve NU. For the purpose of minimizing the error caused by pruning, it is obviously better to use all the remaining neurons to reconstruct the outputs of the pruned one.

In Part IV, we present Reconstruction Error Aware Pruning (REAP), the updated version of NU. In REAP, when we prune a neuron, all the remaining neurons are used to reconstruct the pruned one's outputs by using least squares method. Accordingly, we select the neurons to be pruned based on the error after reconstruction.

However, our new approach requires a lot of computational cost with straightforward implementation. In order to select the neuron to be pruned, we should once prune each neuron and conduct reconstruction by using least squares method. When we have a lot of neurons (e.g. 1,024 neurons are already too many for us), it is not realistic to do such computation.

For efficient neuron selection, we developed a biorthogonal system-based algorithm with which the reconstruction errors for all the neurons can be computed in one-shot. This algorithm reduces the computational order of neuron selection from $O(n^4)$ to $O(n^3)$, where $n$ denotes the number of the neurons. Moreover, although we need to recalculate the re-construction errors each time we prune a neuron, this re-calculation can be further accelerated by using simple linear algebra tricks.

Fig. IV.1 shows the flow chart of REAP. For each layer, we conduct the following steps for each layer separately.

1) Encode the neuron behaviors by feeding images into a DNN model.

2) Compute the reconstruction errors for all the neurons by using the proposed biorthogonal system-based algorithm.

3) Prune the neuron(s) with the smallest reconstruction error.

4) If enough number of neurons have been pruned, finish iteration. Otherwise, go to 2).

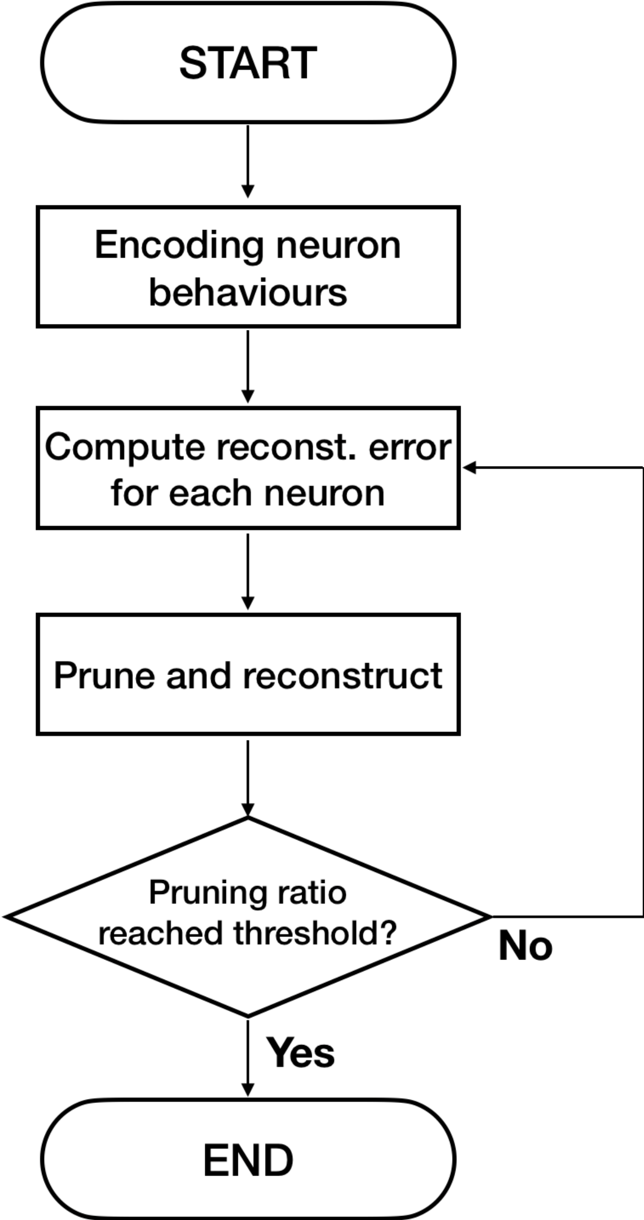


Figure IV.1: The flow chart of REAP. The procedures are similar with those of NU. Although, REAP reconstructs the behavior of the pruned neuron using all the remaining ones.

The rest of Part IV are structured as follows. In Sec. 2, we explain the idea of REAP and the efficient neuron selection algorithm that makes REAP feasible. Sec. 3 shows the experiments to verify the proposed method. Sec. 4 is the conclusion of Part IV.

# 2 From NU to REAP

In this section, we explain only the essence of NU, and show how we extend this method. Then, we show the algorithms to accelerate the neuron selection procedures of REAP.

## 2.1 Neuro-Unification (NU)

Let $n$ and $n'$ denote the numbers of neurons in a layer and the next layer, and $d$ denote the number of input images. The forward propagation is formulated as

$$Y = \sum_{i \in \mathcal{I}} \boldsymbol{x}_i \boldsymbol{w}_i^\top, \tag{IV.1}$$

where $\boldsymbol{x}_i \in \mathbb{R}^d$ denotes the outputs of the $i$-th neuron corresponding to $d$ input images, $\boldsymbol{w}_i \in \mathbb{R}^{n'}$ denotes the weights going from the $i$-th neuron to the ones in the next layer, $Y \in \mathbb{R}^{d \times n'}$ denotes the inner activation levels in the next layer, and $\mathcal{I} = \{1, \cdots, n\}$ is the set of neuron indices. The goal is to reduce the number of neurons to the desired number while keeping $Y$ as unchanged as possible.

**How to unify a given pair of neurons**

When we have a pair of neurons having similar outputs, we merge one of them to the other one without significant error. For instance, if $\boldsymbol{x}_i \simeq a_{ij}\boldsymbol{x}_j$ holds, we prune the $i$-th neuron and update the $j$-th neuron's weights as

$$\boldsymbol{w}'_j = a_{ij}\boldsymbol{w}_i + \boldsymbol{w}_j, \tag{IV.2}$$

where $a_{ij}$ is the coefficient for reconstruction. The error of $Y$ is given by

$$\begin{aligned} \Delta Y &= \boldsymbol{x}_i \boldsymbol{w}_i{}^\top + \boldsymbol{x}_j \boldsymbol{w}_j{}^\top - \boldsymbol{x}_j \boldsymbol{w}'_j{}^\top \\ &= (\boldsymbol{x}_i - a_{ij}\boldsymbol{x}_j)\boldsymbol{w}_i^\top. \end{aligned} \tag{IV.3}$$

If we have $\boldsymbol{x}_i \simeq a_{ij}\boldsymbol{x}_j$, $Y$ can be reconstructed well because the $j$-th neuron compensates the error caused by pruning the $i$-th one.

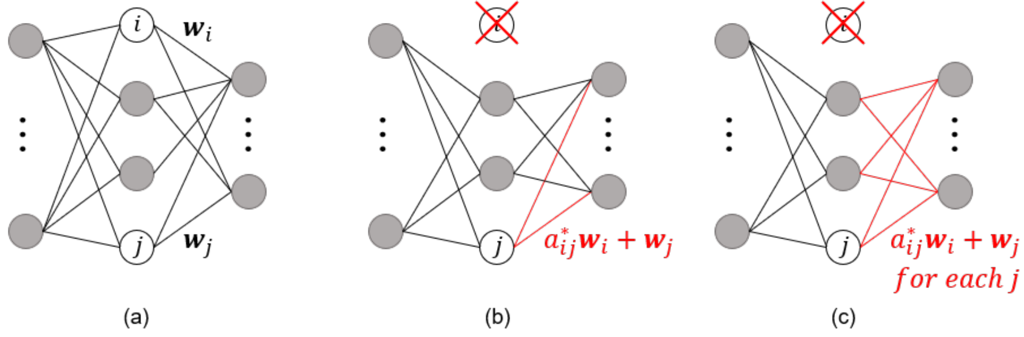


Figure IV.2: The concepts of NU and REAP. (a) The original model. (b) The model pruned with NU. Only one neuron is used for reconstructing the pruned one's behavior. (c) The model pruned with REAP. All the remaining neurons are used for reconstruction.

In order to determine $a_{ij}$, we need to minimize the error of $Y$. This can be formulated as follows.

$$a^*_{ij} = \operatorname*{argmin}_{a_{ij}} \left\| (a_{ij}\boldsymbol{x}_j - \boldsymbol{x}_i)\, \boldsymbol{w}_j^\top \right\|_\mathrm{F}^2. \tag{IV.4}$$

As we explain in Sec. 2.1 of Part III, this boils down to a problem of computing orthogonal projection of $\boldsymbol{x}_i$ onto $\boldsymbol{x}_j$.

## 2.2 Reconstruction Error Aware Pruning (REAP)

In NU, the output of the pruned neuron is reconstructed from another neuron. In REAP, we use all the remaining neurons for reconstruction, as shown in Fig. IV.2. This can be formulated by

$$\{a^*_{ij} | j \in \mathcal{I}\backslash\{i\}\} = \operatorname*{argmin}_{a_{ij}} \left\| \boldsymbol{x}_i - \sum_{j\in\mathcal{I}\backslash\{i\}} a_{ij}\boldsymbol{x}_j \right\|^2. \tag{IV.5}$$

Similarly with Eq. (IV.2), the weights of the remaining neurons are updated as follows for each $j \in \mathcal{I} \setminus \{i\}$.

$$\boldsymbol{w}'_j = a^*_{ij}\boldsymbol{w}_i + \boldsymbol{w}_j. \tag{IV.6}$$

**How to select the neuron to be pruned**

We should select the neuron to be pruned so as to minimize the reconstruction

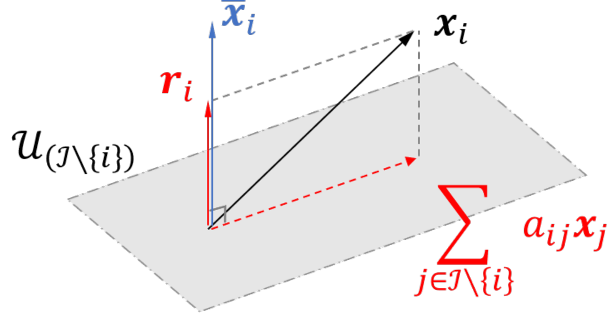


Figure IV.3: Illustration of the projection of $\boldsymbol{x}_i$ onto a subspace $\mathcal{U}_{(\mathcal{I}\setminus\{i\})}$ spanned by $\{\boldsymbol{x}_j | j \in \mathcal{I}\setminus\{i\}\}$. Computing the orthogonal projection of $\boldsymbol{x}_i$ onto $\mathcal{U}_{(\mathcal{I}\setminus\{i\})}$ is equivalent to solving the problem of reconstructing $\boldsymbol{x}_i$ from $\{\boldsymbol{x}_j | j \in \mathcal{I}\setminus\{i\}\}$ by least squares method. The residual $\boldsymbol{r}_i$ is linearly dependent on $\bar{\boldsymbol{x}}_i$, the dual basis for $\boldsymbol{x}_i$.

error of $Y$. This problem can be formulated as

$$\begin{aligned} i^* &= \operatorname*{argmin}_i \left\| Y - \sum_{j\in\mathcal{I}\setminus\{i\}} \boldsymbol{x}_j {\boldsymbol{w}'_j}^\top \right\|_\mathrm{F}^2 \\ &= \operatorname*{argmin}_i \left\| Y - \sum_{j\in\mathcal{I}\setminus\{i\}} \boldsymbol{x}_j \left( a^*_{ij}\boldsymbol{w}_i + \boldsymbol{w}_j \right)^\top \right\|_\mathrm{F}^2 . \end{aligned} \tag{IV.7}$$

In order to solve Eq. (IV.7), we first solve Eq. (IV.5) for each $i \in \mathcal{I}$ to compute the $a^*$-s. With straightforward solution using least squares method, the amount of computation would be tremendous if we have a lot of neurons (and we usually have a lot of neurons).

## 2.3 Neuron selection algorithm based on biorthogonal system

We have developed an efficient algorithm that can obtain the solution of Eq. (IV.5) for each $i \in \mathcal{I}$ in one-shot.

**How to solve Eq. (IV.5) for each $i$ in one-shot**

We solve Eq. (IV.5) for each $i$ in *one-shot* by using biorthogonal system. Let $\boldsymbol{r}_i$ denote the residual of $\boldsymbol{x}_i$ reconstructed from the other $\boldsymbol{x}$-s:

$$\boldsymbol{r}_i = \boldsymbol{x}_i - \sum_{j\in\mathcal{I}\setminus\{i\}} a^*_{ij}\boldsymbol{x}_j . \tag{IV.8}$$

As $\boldsymbol{r}_i$ is the residual of $\boldsymbol{x}_i$, $\boldsymbol{r}_i$ is orthogonal to all other $\boldsymbol{x}$-s. In other words, $\boldsymbol{r}_i$ is orthogonal to the subspace $\mathcal{U}_{(\mathcal{I}\setminus\{i\})}$ spanned by $\{\boldsymbol{x}_j | j \in \mathcal{I} \setminus \{i\}\}$, as shown in Fig. IV.3.

We compute the $\boldsymbol{r}$-s by using biorthogonal system. Let $\{\bar{\boldsymbol{x}}_j | j \in \mathcal{I}\}$ denote the dual bases of $\{\boldsymbol{x}_j | j \in \mathcal{I}\}$. The biorthogonal system is defined by

$$\langle \boldsymbol{x}_i, \bar{\boldsymbol{x}}_j \rangle = \begin{cases} 1 & (i = j) \\ 0 & (otherwise) \end{cases}. \tag{IV.9}$$

The biorthogonal expansion for $\boldsymbol{r}_i$ is given by

$$\boldsymbol{r}_i = \sum_{j \in \mathcal{I}} \langle \boldsymbol{r}_i, \boldsymbol{x}_j \rangle \bar{\boldsymbol{x}}_j. \tag{IV.10}$$

Obviously, $\langle \boldsymbol{r}_i, \boldsymbol{x}_j \rangle = 0$ holds for each $j \in \mathcal{I}\setminus\{i\}$, because $\boldsymbol{r}_i$ is orthogonal to $\mathcal{U}_{(\mathcal{I}\setminus\{i\})}$. Therefore, Eq. (IV.10) can be rewritten as

$$\boldsymbol{r}_i = \langle \boldsymbol{r}_i, \boldsymbol{x}_i \rangle \bar{\boldsymbol{x}}_i + \sum_{j \in \mathcal{I}\setminus\{i\}} \langle \boldsymbol{r}_i, \boldsymbol{x}_j \rangle \bar{\boldsymbol{x}}_j = \langle \boldsymbol{r}_i, \boldsymbol{x}_i \rangle \bar{\boldsymbol{x}}_i. \tag{IV.11}$$

Eq. (IV.11) means that $\boldsymbol{r}_i$ is linearly dependent on $\bar{\boldsymbol{x}}_i$. Therefore, we can obtain $\boldsymbol{r}_i$ by computing the orthogonal projection of $\boldsymbol{x}_i$ onto $\bar{\boldsymbol{x}}_i$, as shown in Fig. IV.3. Thus, the following holds:

$$\boldsymbol{r}_i = \frac{\langle \boldsymbol{x}_i, \bar{\boldsymbol{x}}_i \rangle}{\|\bar{\boldsymbol{x}}_i\|^2} \bar{\boldsymbol{x}}_i = \frac{\bar{\boldsymbol{x}}_i}{\|\bar{\boldsymbol{x}}_i\|^2}. \tag{IV.12}$$

By using Eq. (IV.12), we can compute $\boldsymbol{r}_i$ for each $i \in \mathcal{I}$ in one-shot. Let $X = [\boldsymbol{x}_1 \cdots \boldsymbol{x}_n]$ and $\bar{X} = [\bar{\boldsymbol{x}}_1 \cdots \bar{\boldsymbol{x}}_n]$. By definition of dual bases, $\bar{X}$ can be computed as

$$\bar{X} = (X^g)^\top, \tag{IV.13}$$

where $X^g$ denotes the generalized inverse of $X$. Then, we just compute $\boldsymbol{r}_i$ by using Eq. (IV.12) for each $i \in \mathcal{I}$.

We also need to compute the $a$-s for reconstruction. Let $R = [\boldsymbol{r}_1 \cdots \boldsymbol{r}_n]$ and $A^* \in \mathbb{R}^{n \times n}$ denote a matrix whose $(i, j)$ entry is $a^*_{ij}$. Because we have Eq. (IV.8), the following must hold:

$$R = X - XA^*. \tag{IV.14}$$

Then, we have

$$A^* = E - X^g R, \tag{IV.15}$$

where $E$ denotes an identity matrix.

**Substitute approach for biorthogonal system based algorithm**

We have developed another algorithm for computing the reconstruction error of each neuron by using Gram-Schmidt process. When the number of the neurons are very large (e.g. more than 5K neurons, although it depends on computational environments), the Gram-Schmidt process-based algorithm is faster than the biorthogonal system-based algorithm. The detail of this substitute approach is provided in Appendix B.

### 2.4 Even faster computation for selecting the second neuron to be pruned

Assume that we have pruned the $i$-th neuron. When we prune another neuron, we may simply repeat the same procedures mentioned in Sec. 2.3 with the remaining neurons. We have to solve the following problem for each $j$.

$$\{b^*_{jk} | k \in \mathcal{I}\backslash\{i,j\}\} = \underset{b_{jk}}{\operatorname{argmin}} \left\| \boldsymbol{x}_j - \sum_{k \in \mathcal{I}\backslash\{i,j\}} b_{jk} \boldsymbol{x}_k \right\|^2, \tag{IV.16}$$

Then, we solve the following problem for selecting the next neuron to be pruned:

$$k^* = \underset{k}{\operatorname{argmin}} \left\| Y - \sum_{k \in \mathcal{I}\backslash\{i,j\}} \boldsymbol{x}_k \left( b^*_{jk} \boldsymbol{w}'_j + \boldsymbol{w}'_k \right)^\top \right\|_\mathrm{F}^2. \tag{IV.17}$$

Note that we already have the $\boldsymbol{w}'$-s in Eq. (IV.6).

Although we may use the proposed biorthogonal system-based algorithm again for solving Eq. (IV.16) and Eq. (IV.17), we can solve them even faster by using the solution of Eq. (IV.5). We already have

$$\boldsymbol{x}_i = \boldsymbol{r}_i + a^*_{ij} \boldsymbol{x}_j + \sum_{k \in \mathcal{I}\backslash\{i,j\}} a^*_{ik} \boldsymbol{x}_k, \tag{IV.18}$$

$$\boldsymbol{x}_j = \boldsymbol{r}_j + a^*_{ji} \boldsymbol{x}_i + \sum_{k \in \mathcal{I}\backslash\{i,j\}} a^*_{jk} \boldsymbol{x}_k. \tag{IV.19}$$

After pruning both the $i$-th and the $j$-th neurons, we can no more use $\boldsymbol{x}_j$ for reconstructing $\boldsymbol{x}_i$. Thus, we substitute Eq. (IV.18) to Eq. (IV.19) and get

$$\boldsymbol{x}_j = \frac{\boldsymbol{r}_j + a^*_{ji} \boldsymbol{r}_i}{1 - a^*_{ji} a^*_{ij}} + \sum_{k \in \mathcal{I}\backslash\{i,j\}} \frac{a^*_{jk} + a^*_{ji} a^*_{ik}}{1 - a^*_{ji} a^*_{ij}} \boldsymbol{x}_k. \tag{IV.20}$$

We have $\langle \boldsymbol{x}_k, \boldsymbol{r}_i \rangle = 0$ and $\langle \boldsymbol{x}_k, \boldsymbol{r}_j \rangle = 0$ for each $k \in \mathcal{I}\backslash\{i,j\}$. Therefore, the first term on the RHS of Eq. (IV.20) denotes the residual of $\boldsymbol{x}_j$ reconstructed from

$\{\boldsymbol{x}_k | k \in \mathcal{I} \backslash \{i, j\}\}$ and the second term denotes the projection of $\boldsymbol{x}_j$ onto the subspace spanned by $\{\boldsymbol{x}_k | k \in \mathcal{I} \backslash \{i, j\}\}$. Thus, the coefficients of the $\boldsymbol{x}$-s in the second term is equivalent to the solution of Eq. (IV.16):

$$b_{jk}^* = \frac{a_{jk}^* + a_{ji}^* a_{ik}^*}{1 - a_{ji}^* a_{ij}^*}. \tag{IV.21}$$

After computing the $b^*$-s, we can solve Eq. (IV.17) easily.

### 2.5 Algorithm

To sum up, our neuron selection algorithm can be described as Algorithm IV.1.

**Algorithm IV.1**

**Input:** A set of neuron indices $\mathcal{I} = \{1, \cdots, n\}$, a set of neuron behavioral vectors $\{\boldsymbol{x}_i | i \in \mathcal{I}\}$ and the set of weight vectors of each neuron $\{\boldsymbol{w}_i | i \in \mathcal{I}\}$, an output matrix $Y$, desired number of remaining neurons $q$.

**Output:** A set $\mathcal{J} \subseteq \mathcal{I}$ composed of remaining neurons' indices and a matrix $A \in \mathbb{R}^{n \times n}$ whose $(i, j)$ entry is $a_{ij}$.

$\mathcal{J} \leftarrow \mathcal{I}$.

compute $R$ and $A$ by using Eq. (IV.13), (IV.12), (IV.14), and (IV.15).

**while** $|\mathcal{J}| > q$ **do**

  Select the neuron index $i$ to be pruned by solving Eq. (IV.7).

  $\mathcal{J} \leftarrow \mathcal{J} \setminus \{i\}$.

  $\boldsymbol{w}_j \leftarrow a_{ij} \boldsymbol{w}_i + \boldsymbol{w}_j$ for each $j \in \mathcal{J}$.

  $a_{jk} \leftarrow (a_{jk}^* + a_{ji}^* a_{ik}^*)/(1 - a_{ji}^* a_{ij}^*)$ for each $j, k \in \mathcal{J}, j \neq k$.

**end while**

We provide some tips for implementation of this algorithm. In order to solve Eq. (IV.7), we first compute the $\boldsymbol{w}'$-s, then we need to compute the following for each $i$:

$$f(\mathcal{I} \setminus \{i\}) = \left\| Y - \sum_{j \in \mathcal{I} \setminus \{i\}} \boldsymbol{x}_j \boldsymbol{w}_j'^\top \right\|_{\mathrm{F}}^2 \tag{IV.22}$$

This is also computationally expensive if implemented as is, because $Y$ and $\boldsymbol{x}_i \boldsymbol{w}_i'^\top$ are typically large matrices. In Appendix C, we provide tips for implementation to conduct this computation efficiently.

As we take a greedy approach in neuron selection of REAP, it naturally arises the question "How adequate is its solution?" Actually, REAP's algorithm does not

always give the global optimal solution. However, it gives us fairly good sub-optimal solution. We discuss this point in Appendix D.

### 2.6 Relation to CP

Channel Pruning (CP) [8] is also a layer-wise pruning method that conducts reconstruction with least squares method. Although, its strategy for neuron selection is different from ours. They select the neurons to be pruned by solving the following Lasso regression problem:

$$\begin{aligned} \boldsymbol{\beta}^* = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} & \left\| Y - \sum_{i \in \mathcal{I}} \beta_i \boldsymbol{x}_i \boldsymbol{w}_i^\top \right\|_\mathrm{F}^2 + \lambda \left\| \boldsymbol{\beta} \right\|_1 \\ \text{subject to } & \left\| \boldsymbol{\beta} \right\|_0 \leq q, \end{aligned} \tag{IV.23}$$

where $q$ denotes the desired number of neurons and $\boldsymbol{\beta} = (\beta_1, \cdots, \beta_n)^\top$ denotes a vector used for neuron selection. If $\beta_i = 0$, the $i$-th neuron can be pruned.

Then, reconstruction is performed with least squares method.

$$\left\{ \boldsymbol{w}_j^* | j \in \mathcal{J} \right\} = \underset{\boldsymbol{w}_j}{\operatorname{argmin}} \left\| Y - \sum_{j \in \mathcal{J}} \beta_j^* \boldsymbol{x}_j \boldsymbol{w}_j^\top \right\|_\mathrm{F}^2, \tag{IV.24}$$

where $\mathcal{J} = \{i | \beta_i^* \neq 0\}$ denotes the set composed of remaining neurons' indices.

The weakness of CP is that it selects the neurons to be pruned based on the error *before* reconstruction, which does not guarantee the minimal error *after* reconstruction. On the other hand, REAP selects the neurons to be pruned based on their reconstruction errors. Because of this difference, REAP performs better than CP, as we will show in the experiments.

## 3 Experiments

We conducted the experiments with several DNN models and datasets (VGG16 [2] on ImageNet [54], ResNet-56 [11] on CIFAR-10 [55], and DenseNet-121 [57] on Stanford Dogs [58]) to verify REAP.

Note that even though REAP is an extended method of NU, we mainly compare REAP with CP, because REAP is the most similar with CP in theory, and CP is one of the state-of-the-art methods recently.

## 3.1 Datasets

**ImageNet**

ImageNet is a large scale dataset for 1,000 classes image classification [54]. It has approximately 1.2M images for training, 50K images for validation, and 100K images for testing. Following the former works, we used the validation images as the test dataset, and did not use the official test images in our experiments. As each image has different resolution, we resized them so that the shorter side would become 256 pixels. Then, $224 \times 224$ random crop was applied to the training images, and $224 \times 224$ center crop was applied to the test images. The random horizontal flip was applied to the training images. We randomly selected 5K training images, and used them for encoding neuron behavior.

**CIFAR-10**

CIFAR-10 is a dataset for 10 class image classification [55]. It has 50K images for training and 10K images for testing. All the images have $32 \times 32$ resolution. The training images were padded by 4 pixels at each side and $32 \times 32$ random crop was applied. Random horizontal flip was applied to the training images. The test images were used as they were. We randomly selected 5K images from training dataset for pruning.

**Stanford Dogs**

Stanford Dogs is a dataset for 120 classes fine-grained image classification [58]. It has approximately 12k images for training and 8.58K images for testing. All the images are of dog species. All images were resized so that the shorter side would become 256 pixels. Then, $224 \times 224$ random crop was applied to the training images, and $224 \times 224$ center crop was applied to the test images. The random horizontal flip was applied to the training images. We randomly selected 1.2K training images for pruning

## 3.2 Models

**VGG16**

VGG16 is a model that has 16 weight layers, including 13 convolutional layers and 3 fully connected layers. We used the original VGG16 model that was trained with ImageNet dataset. The convolutional layers are composed of 5 blocks that have 2 or 3 layers. For convenience, we call the $X$-th layer of the $Y$-th block *ConvY-X*.

Table IV.1: The pruning ratio (the ratio of the pruned neurons in each layer) setting for VGG16.

| Layer | FLOPs after pruning | |
|---|---|---|
| | ×2 | ×5 |
| Conv1-1 | 35% | 63% |
| Conv1-2 | 36% | 66% |
| Conv2-1 | 37% | 68% |
| Conv2-2 | 33% | 60% |
| Conv3-1 | 32% | 58% |
| Conv3-2 | 36% | 65% |
| Conv3-3 | 31% | 57% |
| Conv4-1 | 35% | 64% |
| Conv4-2 | 25% | 46% |
| Conv4-3 | 30% | 55% |

For fully connected layers, we call such as *FC1* and *FC2*. Architecture details are mentioned in Appendix A.1.

**ResNet-56**

ResNet-56 is a model having *identity shortcuts* that makes it possible to train a very deep models stably and effectively. ResNet-56 has 54 convolutional layers and 1 fully connected layer, and was trained with CIFAR-10 dataset. ResNet-56 has 3 blocks that have 18 convolutional layers. We call the $X$-th layer of the $Y$-th block is called *Conv$Y$-$X$*. Architecture details are mentioned in Appendix A.2.

**DenseNet-121**

DenseNet-121 is a model that has a lot of skip connections in its convolutional layers. The feature of DenseNet is the dense connection between the layers. Thus, the feature map in a layer are computed with the feature maps of all the shallower layers, not just the previous one. Architecture details are mentioned in Appendix A.3.

## 3.3 VGG16 on ImageNet

We conducted the experiments with VGG16 on ImageNet. We pruned the convolutional layers until the FLOPs became ×0.5 and ×0.2. The pruned models were

Table IV.2: VGG16 on ImageNet. The changes of top-5 accuracy from the baseline (89.5%) are reported (The greater, the better.). In this table, “rt” stands for “retraining”. *our implementation.

| FLOPs | Method | Acc. before rt | Acc. after rt | epochs# |
|---|---|---|---|---|
| ×0.5 | REAP | **-2.0**% | **+0.2**% | 10 |
| | NU [10] | -5.0% | - | - |
| | CP [8] | -2.7% | 0.0% | 10 |
| | *ThiNet [9] | -65.0% | -1.0% | 10 |
| | SPP [17] | - | 0.0% | - |
| ×0.2 | REAP | **-9.4**% | **-1.3**% | 10 |
| | CP [8] | -22.0% | -1.7% | 10 |
| | *ThiNet [9] | -88.8% | -3.4% | 10 |
| | SPP [17] | - | -2.0% | - |

Table IV.3: The statistics of the channels selected by REAP and CP. “P” and “R” stand for “Pruned” and “Remaining”, respectively.

| Layer | CP | REAP P | REAP R |
|---|---|---|---|
| *Conv1-1* | P | 45 | 3 |
| | R | 3 | 13 |
| *Conv1-2* | P | 41 | 7 |
| | R | 7 | 9 |
| *Conv2-1* | P | 88 | 8 |
| | R | 8 | 24 |
| *Conv2-2* | P | 86 | 10 |
| | R | 10 | 22 |
| *Conv3-1* | P | 174 | 18 |
| | R | 18 | 46 |
| *Conv3-2* | P | 182 | 10 |
| | R | 10 | 54 |
| *Conv4-1* | P | 350 | 34 |
| | R | 34 | 94 |
| *Conv4-2* | P | 361 | 23 |
| | R | 23 | 105 |

retrained for 10 epochs at $10^{-5}$ learning rate. The momentum was set to 0.9, the minibatch size was set to 128, and the dropout rate for fully connected layers was set to 0.5. For the pruning ratio setting in each layer, we followed the pruned model provided by [8]'s authors in their Github repository [59]. The detail is shown in Table IV.1. The rest of the setups were set to the same values with [8].

The results are shown in Table IV.2. REAP performs consistently better than the existing methods. After retraining, we marginally outperform the other methods at ×0.5 FLOPs. At ×0.2 FLOPs, the existing methods suffer even larger accuracy drop than we do.

An important observation is that we only suffer 9.4% accuracy drop at ×0.2 FLOPs before retraining. On the other hand, CP suffers 22.0% drop and ThiNet

Table IV.4: Time (sec.) spent for channel selection per layer (channels# in the parentheses), at the pruning ratios of 0.25, 0.5, 0.75.

| Method | *Conv1-1* (64) | | | *Conv1-2* (64) | | | *Conv2-1* (128) | | | *Conv2-2* (128) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0.25 | 0.50 | 0.75 | 0.25 | 0.50 | 0.75 | 0.25 | 0.50 | 0.75 | 0.25 | 0.50 | 0.75 |
| REAP | 2.2 | 2.6 | 3.0 | 2.3 | 2.7 | 3.1 | 8.4 | 11.2 | 14.0 | 8.6 | 11.5 | 14.3 |
| CP | 1.8 | 1.7 | 1.6 | 1.1 | 1.4 | 1.3 | 2.1 | 2.6 | 2.4 | 3.2 | 3.3 | 3.3 |
| Method | *Conv3-1* (256) | | | *Conv3-2* (256) | | | *Conv4-1* (512) | | | *Conv4-2* (512) | | |
| | 0.25 | 0.50 | 0.75 | 0.25 | 0.50 | 0.75 | 0.25 | 0.50 | 0.75 | 0.25 | 0.50 | 0.75 |
| REAP | 41.0 | 60.1 | 79.9 | 41.0 | 60.3 | 79.9 | 363.6 | 612.8 | 868.7 | 361.1 | 603.4 | 848.9 |
| CP | 6.6 | 7.6 | 6.7 | 5.4 | 6.0 | 5.8 | 12.8 | 15.1 | 12.0 | 13.4 | 14.1 | 12.0 |

spoiled the model performance. This is because we use the consistent strategy for channel selection and reconstruction to preserve the performances of the pruned models. As we show better performances before retraining, we can achieve higher accuracy after retraining as well. To put this observation differently, REAP enables us to achieve a certain accuracy with fewer epochs of retraining, which means that we can save time and labors for retraining.

It is also worth noting that the model pruned by REAP, at $\times 0.5$ FLOPs, after retraining, is better than the original VGG16 model. This is most likely because we removed the redundant weights, the remaining weights had smaller chance of being trapped in the local minima during training.

**Close analysis on performance difference between CP and REAP**

As already mentioned, REAP is the most similar with CP in theory. The only difference of them is the channel selection criteria. REAP selects the channels to be pruned in a greedy fashion, which probably raises the following questions:

1) Do the channels selected by REAP actually cause smaller reconstruction error than the channels selected by Lasso Regression in CP?

2) Although REAP takes an efficient algorithm for channel selection, does the computation finishes within reasonable time?

In order to answer these questions, we conducted additional experiments with VGG16 [2]. We pruned *Conv1-1*, *Conv1-2*, *Conv2-1*, *Conv2-2*, *Conv3-1*, *Conv3-2*, *Conv4-1*, and *Conv4-2* with several pruning ratios, and observe the layer-wise reconstruction error and measure the computational time spent on channel selection. For tuning the hyper-parameter in CP (the coefficient for Lasso regression), we use binary search

algorithm as we found out it was the fastest. All the methods are implemented with python3.6 and tested on Intel(R) Core(TM) i9-9900K CPU.

As shown in Fig. IV.4, REAP suffers smaller error than CP. Besides, the trend is that the higher the pruning ratios are, the larger the error gaps between REAP and CP are. Because REAP selects channels based on the reconstruction error, we suffer consistently smaller reconstruction error than CP, despite we use greedy algorithm for channel selection.

Table IV.3 shows the matrices that shows how many neurons were selected for pruning by REAP and CP at 0.75 pruning ratio (similarly with confusion matrices). The fact that a neuron was selected by CP but not by REAP indicates that the outputs of that neuron is easy to reconstruct using the outputs of other neurons. In fact, many neurons were selected in common by REAP and CP, and a few neurons were selected by only one of them. Thus, REAP is better than CP, because REAP can select these neurons whose outputs can be reconstructed.

**Computational time**

Table IV.4 shows the results of computational time measurements. Even though CP is much faster than REAP, we can say that REAP is fast enough. It can finish computation within minutes even in *Conv4-1* and *Conv4-2* that are the largest layers of VGG16. We believe that up to 848 seconds for *Conv4-1* and *Conv4-2* is acceptable enough in practice, considering that REAP saves us time for retraining the pruned model and that the training typically takes much more time (e.g. 1 epoch takes over 8 hours on NVIDIA Geforce GTX 1080 Ti).

**Notes about retraining**

As we followed the experimental setups in [8], we retrained the pruned models for only 10 epochs. Fig. IV.5 shows the learning curves of the pruned models for training dataset. The training loss looks to be going down if we train more. However, extra training will not significantly improve the performance for the test dataset. We mention this in detail in relevant experiments in Part V.

## 3.4 ResNet-56 on CIFAR-10.

### 3.4.1 Setups

In this experiment, we conducted pruning, retrained the pruned models, and evaluated the performances before and after the retraining.

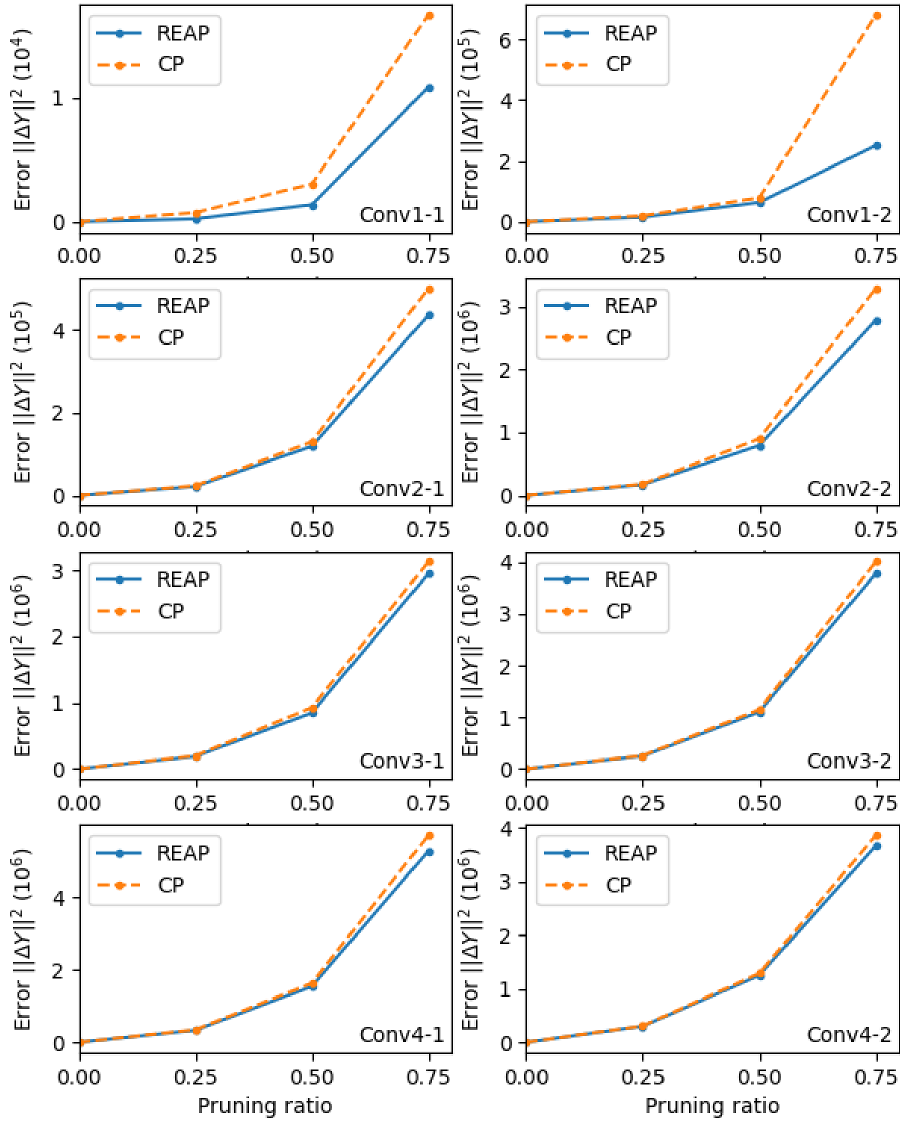


Figure IV.4: Layer-wise analysis for VGG16 on ImageNet. The channels selected by REAP cause consistently smaller reconstruction error than the channels selected by CP.

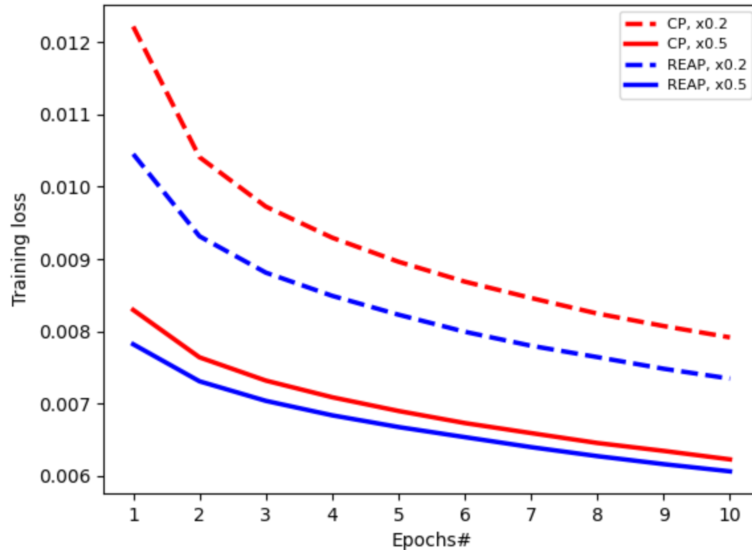


Figure IV.5: The learning curves of the pruned VGG16 models for training dataset.

ResNet-56 has 54 convolutional layers, including 27 layers that have the identity paths. The pruning ratios in the first 18 units, the second 18 ones and the rest were set to $3:2:1$. The pruned models were retrained for 100 epochs, beginning with the learning rate $10^{-2}$ and dividing it by 10 every 25 epochs. The rest of the training setups were aligned with [11]. Since we wanted to compare REAP with CP in the same conditions, we tried to evaluate CP on our own and put the results reported in [8] just for reference (the pretrained model used in [8] is not available).

### 3.4.2 Results

The results are shown in Table IV.5. Before retraining, we could easily outperform the existing methods. We suffer only 1.9% accuracy drop without retraining, while CP suffers 3.7%. After retraining, we are slightly worse than, although competitive with another state-of-the-art pruning method DCP [16]. However, while DCP needs 400 epochs of retraining to achieve this result, we only need 100 epochs to achieve the competitive result. In this way, the fact that we can save efforts on retraining is a strength of REAP.

Table IV.5: ResNet-56 on CIFAR-10. The changes of top-1 accuracy (baseline: 93.4%) are reported (The greater, the better.). *our implementation. **results taken from [8].

| FLOPs | Method | Acc. before rt | Acc. after rt | Epochs# |
|---|---|---|---|---|
| ×0.5 | REAP | **-1.9%** | -0.5% | 100 |
| | NU [8] | -13.3% | -1.9% | 10 |
| | *CP [8] | -3.7% (**-2.0%) | -0.9% (**-1.0%) | 100 |
| | *ThiNet [9] | -56.9% | -1.9% | 100 |
| | DCP [16] | - | **-0.3%** | 400 |

Table IV.6: DenseNet-121 on Stanford Dogs. The changes of top-1 accuracy (baseline: 84.6%) are reported (The greater, the better.). *our implementation.

| FLOPs | Method | Acc. before rt | Acc. after rt | Epochs# |
|---|---|---|---|---|
| ×0.5 | REAP | **-3.1**% | **-3.3**% | 20 |
| | *CP [8] | -4.7% | -3.5% | 20 |
| | *ThiNet [9] | -63.7% | -4.9% | 20 |

## 3.5 DenseNet-121 on Stanford Dogs.

### 3.5.1 Setups

Finally, we conducted the experiments on DenseNet-121 fine-tuned with Stanford-Dogs dataset. For transfer learning, we set the learning rate to $10^{-2}$ for the first 30 epochs and set it to $10^{-3}$ for 20 more epochs. For retraining after pruning, we set the learning rate as $10^{-3}$ and trained the models for 10 epochs, then repeated another 10 epochs with the learning rate $10^{-4}$. We set the pruning ratios in Block1, Block2, Block3 and Block4 to $5:5:4:3$. The rest of the setups were aligned with Section 3.3.

### 3.5.2 Results

The results are shown in Table IV.6. Similarly with other experiments, REAP could preserve the model accuracy better than the other methods. It is also remarkable that the model pruned by REAP without retraining is as accurate as the model after retraining. REAP preserves the model performances so well that we sometimes do not even need to retrain the pruned models.

## 4 Summary of Part IV

In Part IV, we proposed REAP, a pruning method that is the extended version of NU. REAP reconstructs the pruned neuron's behavior using all the remaining neurons by least squares method. REAP can reduce the computational complexity of the DNN models while maintaining their performances, which not only makes it possible to produce a fast, compact, and accurate model but also saves time and labors required for retraining. On the experiments with several well-known models and benchmark datasets, we could confirm these strengths of REAP.

REAP is the best pruning method in terms of minimizing layer-wise error. Although, as a nature of layer-wise pruning method, pruning has to be performed in each layer separately, and pruning ratio in each layer has to be determined by human hands. In Part V, we will present a method that can be combined with REAP for optimizing pruning ratios.

# Part V

# Pruning Ratio Optimizer

## 1 Introduction

The common problem of the layer-wise pruning methods including REAP is that there is not a proper way of determining the pruning ratio (the ratio of neurons to be pruned) in each layer. Intuitively, it is reasonable to say that just a small error in a certain layer caused by pruning may not have a significant impact on the model accuracy, and vice versa. However, there may be a sensitive layer where pruning just a few neurons will significantly affect the model performance. Conventionally, the way of optimizing the pruning ratios was to actually try to perform pruning with various settings for the pruning ratios, which is inefficient.

In Part V, we present Pruning Ratio Optimizer (PRO), a method for optimizing the pruning ratio in each layer based on the error in the final layer of the model. In PRO, we repeat the following steps until the pruned model becomes fast (and/or small) enough:

1) Select the the most redundant layer.

2) Prune a small number of neurons in the selected layer.

For evaluating the redundancy, we try to perform pruning in each layer with several pruning ratios, and observe the error in the final layer of the pruned model, as shown in Fig. V.1. The layer where pruning will have the smallest impact on the outputs in the final layer will be selected, and some neurons are pruned in that layer. After some iterations, the pruning ratio in each layer will be properly tuned.

It is worth noting that PRO has to be combined with REAP, even though other layer-wise pruning methods that conduct reconstruction, such as CP [8] and ThiNet [9], can also be used. This is because REAP is the best method for preventing the error. As far as the pruned model retains close to its original accuracy, we can say that more neurons can still be pruned. On the other hand, with other pruning methods, the model being pruned easily suffers significant degradation. After significant degradation, we cannot judge if more neurons can be pruned. Therefore, we can optimize the pruning ratios more properly if we use REAP.

The rest of Part V are structured as follows. The related works are summarized in Sec. 2, the proposed method is explained in Sec. 3, the experiments are in Sec.

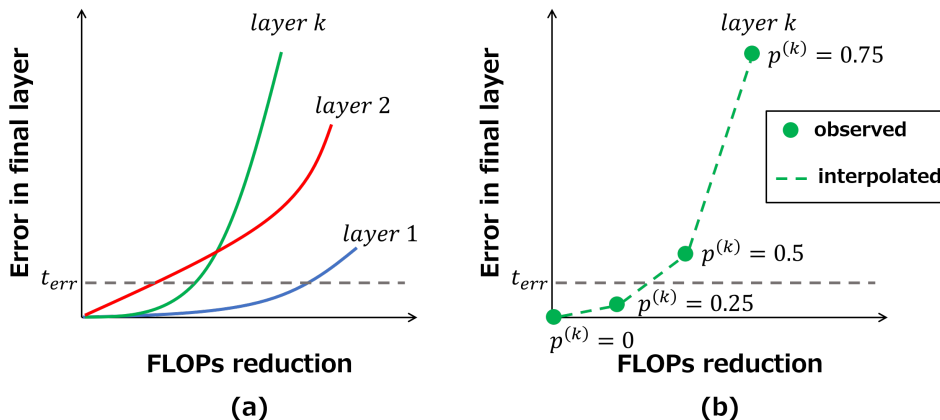


Figure V.1: (a) Illustration of the idea of PRO. In each layer, we try pruning with several pruning ratios and observe the error in the final layer. Then, we set the error threshold $t_{err}$, select the layer where the most FLOPs can be reduced at the cost of error of $t_{err}$ (In this case, $layer1$ will be selected.), and perform pruning in the selected layer. We repeat these procedures several times until the inference with the pruned model becomes fast enough. (b) The strategy for efficient layer selection. Drawing precise curves is computationally intensive, as it requires us to conduct pruning and error observation repeatedly. Therefore, we set $p^{(k)}$, the pruning ratio in the $k$-th layer, to a few values (In this example, $p^{(k)} = 0, 0.25, 0.5, 0.75$.), conduct pruning, and observe the error in the final layer. We perform linear interpolation between the observed points.

4, and we conclude the discussions in Sec. 5.

## 2 Related works

For pruning ratio optimization with a layer-wise method, He et al. proposed AutoML Model Compression (AMC), a method to optimize pruning ratio based on reinforcement learning [60]. They show that the accuracy of the pruned model can be preserved better if they optimize pruning ratios with AMC than if they do by human hands. Although, AMC has some weaknesses:

- Reinforcement learning itself is computationally expensive, because it requires us to perform pruning quite a lot of times with various pruning ratio settings.
- One still needs to tune lots of hyper-parameters related to reinforcement learn-

ing by human hands.

Therefore, it is desired to develop a novel method which is easier to use and less time-consuming.

The holistic pruning methods can be used for optimizing the pruning ratios as well. However, because most existing holistic methods do not perform reconstruction, the pruned models suffer significant accuracy degradation. Exceptionally, Optimal Brain Surgeon (OBS) [7] is a holistic method that performs reconstruction. However, as we already mentioned in Part II Sec. 2.1, it is not realistic to apply OBS to large DNN models due to heavy computational cost. Structured Probabilistic Pruning (SPP) [17] conducts pruning by using dropout. The idea of SPP is to conduct extra training on a pretrained model with some neurons dropped out (in other word, the weights connected to those neurons are temporarily set to 0), and if it ends up in high accuracy, the dropped neurons are not important and can be eventually pruned. As SPP requires a lot of training, it is computationally expensive.

# 3 How to optimize the pruning ratio with a layer-wise pruning method

In this section, we explain Pruning Ratio Optimizer (PRO). We first re-formulate REAP in order to make it easier to explain PRO. Then, we show the details of PRO.

## 3.1 Formulation of REAP

Let $n^{(k)}$ denote the number of neurons in the layer where pruning is performed, $\mathcal{I}^{(k)} = \{1, \cdots, n^{(k)}\}$ denote the set of neuron indices, $\boldsymbol{x}_i^{(k)}$ denote the $i$-th neuron's behavioral vector, $\boldsymbol{w}_i^{(k)}$ denote the weights going from the $i$-the neuron to the ones in the next layer, $Y^{(k)} = \sum_{i \in \mathcal{I}} \boldsymbol{x}_i^{(k)} {\boldsymbol{w}_i^{(k)}}^\top$ denote the layer-wise outputs. REAP's neuron selection can be formulated as below.

$$\begin{aligned} {\mathcal{J}^{(k)}}^* = \underset{\mathcal{J}^{(k)}}{\operatorname{argmin}} \min_{\boldsymbol{w}_i^{(k)}} & \left\| Y^{(k)} - \sum_{i \in \mathcal{J}^{(k)}} \boldsymbol{x}_i^{(k)} {\boldsymbol{w}_i^{(k)}}^\top \right\|_\mathrm{F}^2, \\ \text{subject to} & \left| \mathcal{J}^{(k)} \right| \leq (1 - p^{(k)}) \left| \mathcal{I}^{(k)} \right|, \end{aligned} \tag{V.1}$$

where $\mathcal{J}^{(k)}$ denotes the set of the remaining neurons' indices and $p^{(k)}$ denotes the pruning ratio. Note that we obtain the solution of Eq. (V.1) by solving Eq. (IV.7)

sequentially. REAP's neuron selection algorithm presented in Part IV can find a better solution of this problem than other layer-wise pruning methods, such as [8].

Although REAP is good at preserving the original layer-wise outputs, it is not obvious how much this layer-wise error will have an impact on the model accuracy. Some amount of error in a layer may not affect model performance, although the same amount of error in another layer may lead to significant degradation. Moreover, pruning in a layer will change the outputs of the subsequent layers, which makes it difficult to optimize the pruning ratios in several layers simultaneously.

### 3.2 Pruning Ratio Optimizer (PRO)

We propose Pruning Ratio Optimizer (PRO) that can be combined with REAP (or other layer-wise pruning methods) for optimizing the pruning ratios. In PRO, we optimize the pruning ratio in each layer so as to minimize the error in the final layer of the model. Because it is difficult to solve this optimization problem analytically, we solve it in a greedy fashion. The idea of PRO is to select the most redundant layer and prune some neurons in the selected layer, repeatedly.

The procedures of PRO can be described as follows.

**Step 1)** Draw a curve of FLOPs reduction and the error in the final layer that is caused by performing pruning in each layer, as shown in Fig. V.1 (a). In order to do this, we try to set the pruning ratio to various values, apply REAP, and observe the errors in the final layer. Note that pruning is conducted separately in each layer, and the pruned neurons and the updated weights have to be restored in this step.

**Step 2)** Set the threshold $t_{err}$ to the error in the final layer. By using the curves drawn in Step 1), select the layer where the most FLOPs can be reduced at the cost of error of $t_{err}$. How to determine $t_{err}$ properly will be explained later.

**Step 3)** Perform pruning in the selected layer until the error in the final layer reaches $t_{err}$.

**Step 4)** If enough amount of FLOPs have been reduced, terminate computation. Otherwise, go to Step 1). At the end, the pruning ratio in each layer will be properly tuned.

In order to avoid ambiguity, we provide more detailed descriptions for Step 1). Let $\mathcal{M}$ denote the model that have $k^+$ layers and $\mathcal{D}$ denote the dataset used for

pruning. The original outputs in the final layer are given by

$$Y^{(k^+)} = \mathcal{M}(\mathcal{D}). \tag{V.2}$$

Then, we prune $l = p^{(k)} \left|\mathcal{I}^{(k)}\right|$ neurons in the $k$-th layer with REAP. Let $\mathcal{M}_{k,l}$ denote the model after pruning $l$ neurons in the $k$-th layer. Then, the error in the final layer becomes

$$\Delta Y_{k,l}^{(k^+)} = Y^{(k^+)} - \mathcal{M}_{k,l}(\mathcal{D}). \tag{V.3}$$

We also need to compute FLOPs reduction achieved by pruning. By pruning $l$ neuron in the $k$-th layer, the amount of reduced FLOPs is given by

$$\Delta o^{(k)} = l\left(n^{(k-1)} + n^{(k+1)}\right). \tag{V.4}$$

We draw a curve of $\left\|\Delta Y_{k,l}^{(k^+)}\right\|_{\mathrm{F}}^2$ and $\Delta o^{(k)}$. This has to be repeated for each $k \in \{1, \cdots, k^+ - 1\}$.

The remaining question is how to determine the threshold $t_{err}$ in Step 2). With extremely small $t_{err}$, FLOPs reduction in each layer corresponding to the error of $t_{err}$ will be close to zero, and we will not be able to prune any neuron in any layer. With too large $t_{err}$, all the neurons in the selected layer will be pruned. In order to avoid these situations, we induce a threshold $t_{flops}$ for FLOPs to be reduced at each iteration. We first set a very small value to $t_{err}$, and select a layer that has the largest $\Delta o^{(k)}$ at the cost of error of $t_{err}$ in the final layer. If $\Delta o^{(k)}$ in the selected layer is no smaller than $t_{flops}$, we go to Step 3). Otherwise, we increase $t_{err}$ a little and repeat Step 2).

It is worth noting that because of REAP's high ability of preserving the original layer-wise outputs, pruning a small number of neurons in a layer barely changes the layer-wise outputs significantly, and thus, the final layer's outputs are not affected significantly as well. Therefore, in Step 2), we normally select several layers where we conduct pruning in Step 3). Then, we can reduce FLOPs more efficiently at each iteration, which saves the computational cost for pruning ratio optimization with PRO.

### 3.3 Strategy for more efficient optimization

We still have a problem with PRO, which is the large computational cost for Step 1). In order to draw precise curves of error and FLOPs reduction such as Fig. V.1 (a), we need to compute Eq. (V.3) each time we prune a neuron, which is

computationally intensive. Thus, we draw rough curves such as Fig. V.1 (b) in the following scheme.

a) Set the pruning ratio in the $k$-th layer $p^{(k)}$ (Then, $l = p^{(k)} \left|\mathcal{I}^{(k)}\right|$ neurons will be pruned.) to some value, compute corresponding $\left\|\Delta Y_{k,l}^{(k+)}\right\|_{\mathrm{F}}^{2}$ and $\Delta o^{(k)}$ by using Eq. (V.3) and Eq. (V.4). This step has to be repeated a few times, with several values of $p^{(k)}$ (e.g. $p^{(k)} = 0, 0.25, 0.5$).

b) Plot $\Delta o^{(k)}$ and $\left\|\Delta Y_{k,l}^{(k+)}\right\|_{\mathrm{F}}^{2}$, as shown in Fig. V.1. As we have only a few dots on the plot, we perform linear interpolation between the dots so that the error ($\left\|\Delta Y_{k,l}^{(k+)}\right\|_{\mathrm{F}}^{2}$) corresponding to an arbitrary value of $\Delta o^{(k)}$ can be estimated.

### 3.4 Algorithm

The procedures of PRO are summed up in Algorithm V.1. Here, we assume that REAP is employed for pruning.

## 4 Experiments

We evaluated PRO with several benchmark datasets and several models. We implemented PRO with Python 3.6.9 and Pytorch 1.0.0. All the experiments were done on Intel Core-i9 9900K CPU and a single board of NVIDIA Titan RTX GPU.

### 4.1 Datasets

**ImageNet**

ImageNet is a large scale dataset for 1,000 classes image classification [54]. It has approximately 1.2M images for training, 50K images for validation, and 100K images for testing. Following the former works, we used the validation images as the test dataset, and did not use the official test images in our experiments. As each image has different resolution, we resized them so that the shorter side would become 256 pixels. Then, $224 \times 224$ random crop was applied to the training images, and $224 \times 224$ center crop was applied to the test images. The random horizontal flip was applied to the training images. We randomly selected 5K training images, and used them for encoding neuron behavior.

**CIFAR-10**

CIFAR-10 is a dataset for 10 class image classification [55]. It has 50K images for

**Algorithm V.1**

**Input:** Model $\mathcal{M}$, the number of selected layers for pruning in each iteration $m$, threshold for FLOPs reduction $t_{flops}$, threshold of error in the final layer $t_{err}$, a set $\mathcal{P}$ whose elements denote pruning ratios, a dataset used for pruning $\mathcal{D}$.

**while** The number of the FLOPs is not small enough **do**

  **for** $k = 1, \cdots, k^+ - 1$ **do**

    Feed $\mathcal{D}$ into $\mathcal{M}$ to compute $\{\boldsymbol{x}_i^{(k)} | i \in \mathcal{I}^{(k)}\}$ for each $k = \{1, \cdots, k^+\}$.

    **for** $p^{(k)} \in \mathcal{P}$ **do**

      $\mathcal{M}' \leftarrow \mathcal{M}$.

      Set pruning ratio to $p^{(k)}$ and perform pruning with REAP on the $k$-th layer of $\mathcal{M}'$.

      Compute corresponding $\left\| \Delta Y_{k,l}^{(k^+)} \right\|_{\mathrm{F}}^2$ and $\Delta o^{(k)}$ by using Eq. (V.3) and Eq. (V.4), where $l = p^{(k)} \left| \mathcal{I}^{(k)} \right|$.

    **end for**

    Make a plot of $\left\| \Delta Y_{k,l}^{(k^+)} \right\|_{\mathrm{F}}^2$ and $\Delta o^{(k)}$, as shown in Fig. V.1. Perform linear interpolation between the plots.

  **end for**

  $t'_{err} \leftarrow t_{err}$.

  **while** $\Delta o^{(k)}$ in the selected layer(s) is smaller than $t_{flops}$ **do**

    Select $m$ layer(s) with the largest $\Delta o^{(k)}$ at $\left\| \Delta Y_{k,l'}^{(k^+)} \right\|_{\mathrm{F}}^2 = t'_{err}$, where $l' = p'^{(k)} \left| \mathcal{I}^{(k)} \right|$.

    Compute corresponding pruning ratio $p'^{(k)}$ in the selected layer(s).

    $t'_{err} \leftarrow z t'_{err}$, where $z$ is arbitrary value larger than 1.

  **end while**

  Perform pruning on the selected layers of $\mathcal{M}$, with $p'^{(k)}$ pruning ratio for the $k$-th layer.

**end while**

training and 10K images for testing. All the images have $32 \times 32$ resolution. The training images were padded by 4 pixels at each side and $32 \times 32$ random crop was applied. Random horizontal flip was applied to the training images. The test images were used as they were. We randomly selected 5K images from training dataset for pruning.

## 4.2 Models

**VGG16**

VGG16 is a model that has 16 weight layers, including 13 convolutional layers and 3 fully connected layers. We used the original VGG16 model that was trained with ImageNet dataset. The convolutional layers are composed of 5 blocks that have 2 or 3 layers. For convenience, we call the $X$-th layer of the $Y$-th block *ConvY-X*. For fully connected layers, we call such as *FC1* and *FC2*. Architecture details are mentioned in Appendix A.1.

**ResNet-56**

ResNet-56 is a model having *identity shortcuts* that makes it possible to train a very deep models stably and effectively. ResNet-56 has 54 convolutional layers and 1 fully connected layer, and was trained with CIFAR-10 dataset. ResNet-56 has 3 blocks that have 18 convolutional layers. We call the $X$-th layer of the $Y$-th block is called *ConvY-X*. Architecture details are mentioned in Appendix A.2.

## 4.3 VGG16 on ImageNet

### 4.3.1 Setups

We performed pruning until the FLOPs would become approximately $\times 0.2$ of the original VGG16 model.

The baseline method is AMC [60]. AMC is a reinforcement learning-based method for pruning ratio optimization. Basically, PRO is combined with REAP, and AMC is combined with a layer-wise pruning method named CP [8]. For fair comparison of PRO and AMC, we also evaluated the combination of PRO and CP. In addition, we applied REAP with uniform pruning ratio settings in all the layers.

As shown in Algorithm V.1, the hyper-parameters in PRO are as follows. $m$ is the number of layers to be selected in each iteration, $t_{err}$ is the threshold of the error in the final layer, $t_{flops}$ is the amount of FLOPs that should be reduced at each iteration, and $\mathcal{P}$ is the set whose elements are the pruning ratios and are substituted

to $p^{(k)}$. We set $m = 3$, $t_{err} = 10^{-10}$, $t_{flops} = 2 \times 10^8$ (For reference, the original VGG16 model has $1.547 \times 10^{10}$ FLOPs.), and $\mathcal{P} = \{0, 0.125, 0.25, 0.375, 0.5\}$.

Regarding to AMC, we could not find some important experimental information in [60]. In order to be fair, we evaluated AMC by ourselves using the source code provided by [60]'s authors[1].

The pruned models were fine-tuned for 10 epochs with $10^{-5}$ learning rate. The momentum was set to 0.9, the mini-batch size was set to 128, and the dropout rate in the fully connected layers was set to 0.5. For the rest of training setups, we followed [2].

### 4.3.2 Results

We performed pruning with the pruning ratio optimization. The results are summarized in Table V.1, and the discussions are as follows.

**Comparison to the case of uniform pruning ratio**

Compared to the the case of uniform pruning ratios in all the layers, we could make the accuracy degradation much smaller. Especially, the accuracy degradation was smaller by over 23% by using PRO, at approximately $\times 0.2$ FLOPs ratio, before retraining.

The accuracy of the pruned model after retraining was better when using PRO. This is because 1) By using PRO, we can preserve the accuracy of the pruned model well, which means that we can start retraining with the models that have been less damaged; 2) The pruning ratio for each layer has been optimized even without retraining.

**Comparison to AMC**

We then discuss the comparison of PRO & CP and AMC & CP. As shown in Table V.1, PRO could outperform AMC significantly. PRO suffers 15.9% accuracy degradation at $\times 0.203$ FLOPs ratio without retraining, while AMC suffers 39.8% degradation at $\times 0.219$ FLOPs rate. After retraining, PRO still suffers smaller degradation than AMC by 2.0%.

One thing that should be noted is the implementation difference of PRO and AMC. In PRO, each time we perform pruning in a layer, we encode the neuron behaviors in all the layers again. After pruning in a layer, it affects the neuron behaviors in other layer, and we cannot perform pruning properly without re-encoding

[1]https://github.com/mit-han-lab/amc

Table V.1: VGG16 on ImageNet. The top-5 accuracy are reported (The greater, the better.). In this table, “rt” stands for “retraining”, ”uniform” means that the pruning ratio was set to the same value for all the layers. The baseline accuracy of the original VGG16 model is 89.5%.

| Method | FLOPs | Acc. before rt | Acc. after rt | Time for optim. |
|---|---|---|---|---|
| PRO & REAP | ×0.200 | **80.5**% | **88.2**% | 78,026 sec |
| PRO & CP | ×0.203 | 73.6% | 87.8% | 71,840 sec |
| AMC & CP | ×0.219 | 49.7% | 85.8% | 35,181 sec |
| uniform & REAP | ×0.212 | 56.2% | 87.1% | - |

them. Even though the optimization schemes of PRO and AMC are totally different, re-encoding of neuron behaviors is important in AMC as well for reconstruction. However, in their implementation, they encode the neuron behaviors in all the layers only in the beginning, and keep using those initial neuron behaviors to the end in order to shorten time for pruning ratio optimization.

Then, what if we conduct re-encoding for neuron behaviors in AMC? We tried to apply AMC while re-encoding the neuron behaviors. It took 1.7M sec (approximately 20 days) for pruning ratio optimization. However, the accuracy before retraining improved only 0.6% (49.7% to 50.3%), and the accuracy after retraining dropped by 0.2% (86.8% to 86.6%). After all, re-encoding the neuron behaviors did not work for improving the performance of AMC.

Then, why the performance of AMC was worse than PRO? We discuss it in detail in the following paragraphs.

#### Analyses on optimized pruning ratio in each layer

Fig. V.2 shows the pruning ratio in each layer of the VGG16 model. The trend of both PRO and AMC is that they set higher pruning ratios to the layers closer to the input side and lower pruning ratios to the layers closer to the output side.

A remarkable observation is that PRO does not prune a lot in *Conv5-1* and *Conv5-2* layers, while AMC does. Actually, it is known that these layers are not redundant and pruning them leads to significant degradation [8, 9]. PRO could successfully find out that these layers should not be pruned and eventually set zero or very low pruning ratios to them. On the other hand, AMC pruned a lot in these layers, which ended up in significant degradation.

We also investigated how the error in the final layer responses to the pruning

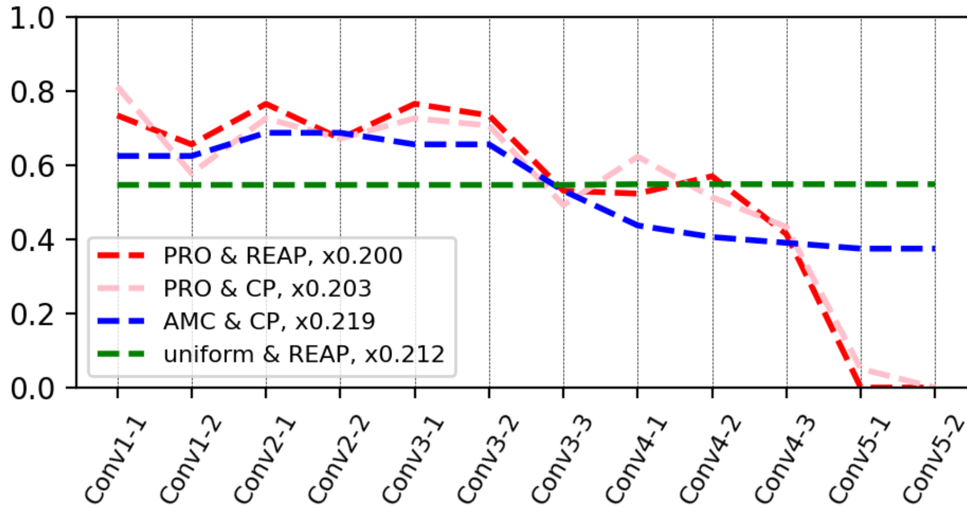


Figure V.2: Results of pruning ratio optimization for VGG16. Both PRO and AMC tend to set higher pruning ratio to the layers on the input side and lower pruning ratio to the layers on the output side. The difference is that PRO does not prune Conv5-1 and Conv5-2 layers a lot, while AMC does. As reported in several literatures, such as [8, 9], pruning these layers leads to significant degradation. And our PRO successfully avoids pruning these layers.

ratio in each layer. We used REAP to prune *Conv1-1*, *Conv2-1*, *Conv3-1*, *Conv4-1*, and *Conv5-1* layers, with various pruning ratios, and observed the error in the final layer. The result is shown in Fig. V.3.

Fig. V.3 (a) shows the relationship of the error in the final layer and the pruning ratio in each layer, and Fig. V.3 (b) shows a similar graph with FLOPs reduction in the horizontal axis. We can see clearly different trends between the layers. In the *Conv5-1* layer, the error increases more rapidly than the other layers. Thus, by observing the relationship of pruning ratio (FLOPs reduction) and the error directly, we can get the insight that we should not perform pruning a lot in *Conv5-1*.

Why did AMC set higher pruning ratios to the *Conv5-1* and *Conv5-2* layers? We suppose that AMC's reinforcement learning-based algorithm was simply not capable of evaluating the redundancy of the layers. As it performs pruning in all the layers simultaneously, it cannot evaluate the impact of the pruning ratio in each layer on the accuracy directly.

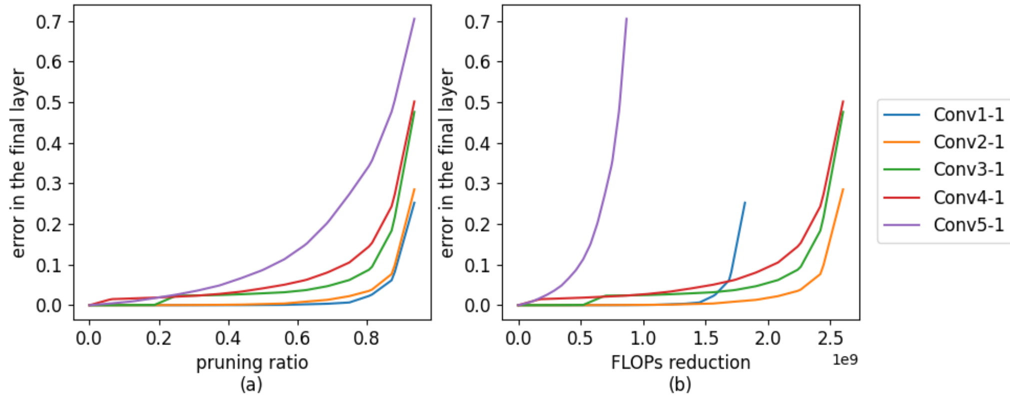


Figure V.3: (a) Relationship of the error in the final layer and pruning ratio in each layer. (b) Relationship of the error in the final layer and FLOPs reduction in each layer, which we actually use for selecting the layer to be pruned.

Table V.2: The results with the ResNet-56 model on the CIFAR-10 dataset. The top-1 accuracy are reported (The greater, the better.). The baseline accuracy is 92.8%.

| Method | FLOPs | Acc. before rt | Acc. after rt | Time for optim. |
|---|---|---|---|---|
| PRO & REAP | ×0.500 | **90.6**% | **92.1**% | 4,237 sec |
| PRO & CP | ×0.498 | 90.0% | 92.0% | 3,800 sec |
| AMC & CP | ×0.501 | 79.0% | 91.4% | 6,885 sec |
| uniform & REAP | ×0.510 | 86.3% | 91.2% | - |

**Learning curves of the pruned models**

Following [60], we retrained the pruned models for only 10 epochs. Fig. V.4 shows the learning curves of the pruned models for training dataset. Based on the curves, the training loss will still be going down if we train the pruned models for some more epochs. However, more training will not, at least, make AMC as good as PRO for the test dataset. We performed extra training for the model pruned with AMC for 10 more epochs (thus, 20 epochs in total), however, it achieved only 86.8%, which is still worse than PRO.

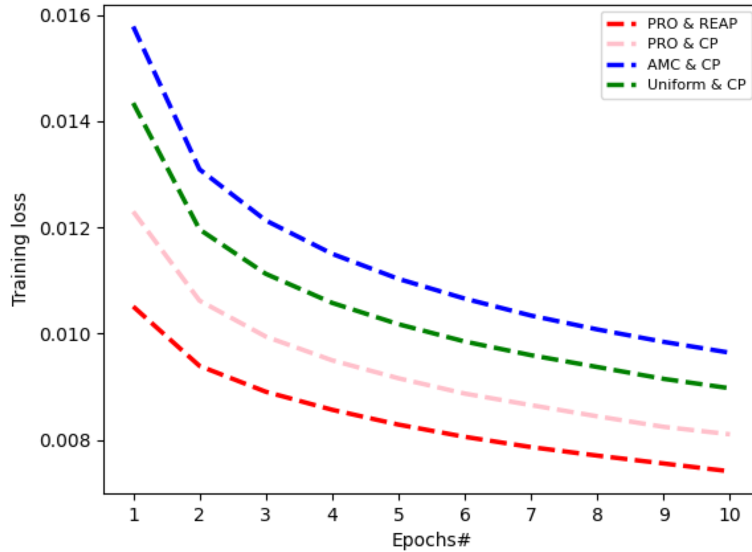


Figure V.4: The learning curves of the pruned VGG16 models for training dataset.

## 4.4 ResNet-56 on CIFAR-10

### 4.4.1 Setups

We conducted pruning on ResNet-56 so that the FLOPs would become half. The pruned models were retrained for 200 epochs, beginning with the learning rate $10^{-2}$ and dividing it by 10 at 100 epochs.

### 4.4.2 Results

The results are summarized in Table V.2. Similarly with the experiments with the VGG16 model and ImageNet dataset, we could outperform AMC in the accuracy of the pruned model. Compared with the case of using REAP with the uniform pruning ratios, using PRO significantly improved the accuracy of the pruned model. AMC & CP suffers 6.5% accuracy degradation while PRO & CP suffers only 2.8%. PRO & CP still outperforms AMC & CP by 0.6% after retraining. By using REAP instead of CP with PRO, the results improved even more.

## 5 Summary of Part V

In Part V, we presented PRO, a method for optimizing the pruning ratio in each layer of a DNN model. Some layer-wise pruning methods are theoretically sound and better than the conventional holistic pruning methods. However, if we perform pruning on several layers of the model simultaneously, we need to be able to tune the pruning ratio in each layer properly. With PRO, we can determine the pruning ratios so as to minimize the error in the final layer of the model. We assume that PRO is combined with REAP, even though other pruning methods could be the options. REAP can preserve the original layer-wise outputs well even without retraining. Therefore, by combining PRO and REAP, we can search proper pruning ratio without time-consuming retraining. The experimental results verify the effectiveness of PRO.

# Part VI
# Serialized Residual Network

## 1 Introduction

With REAP and PRO that we presented in Part IV and V, we can conduct pruning on pretrained large DNN models, so as to make them more efficient and preserve their performances simultaneously. On the other hand, there is an important point that we have not discussed so far, which is the limitation of structured pruning for ResNet. In Part VI, we discuss this limitation and present its solution.

In the recent developments of Computer Vision, the contribution of Residual Network (ResNet) [11] has been remarkable. In the competition of large scale image recognition [54], ResNet significantly outperformed the models that had been developed before ResNet, such as VGG [2]. It is widely believed that the key of ResNet is the architecture with *identity shortcuts*. ResNet architecture is composed of the stacked blocks that are called *ResNet blocks* With identity shortcuts, the convolutional layers are trained so that the optimal residual of the feature maps is learned. This architecture makes it possible to train a very deep model effectively and stably. This is why ResNet could show a record-breaking performance at that time [11].

Although, similarly with other neural network models, the ResNet models are computationally expensive and may not be deployed on the edge devices as they are. One of the effective approaches for saving the computational cost is to conduct pruning for reducing the number of channels in the convolutional layers.

However, the structured pruning methods including REAP have a limitation when we prune ResNet. The architecture of ResNet consists of the blocks with identity shortcuts, as shown in Fig. VI.1 (a). The feature maps go through convolutional layers and are added to the ones coming through the identity shortcut. At this addition, the dimensions of two inputs must match, which means that we cannot prune the layers connected to the identity shortcuts. This limitation is crucial, because ResNet architecture is employed in various models for various tasks, such as object detection, segmentation, and so on.

Therefore, we propose a technique to transform ResNet into a serial network which we refer to by *Serialized Residual Network* (SRN). In Fig. VI.1 (a) and (b), we show a ResNet block and an equivalent SRN block. By building the kernels in the SRN block by concatenating the kernels taken from ResNet and the ones that

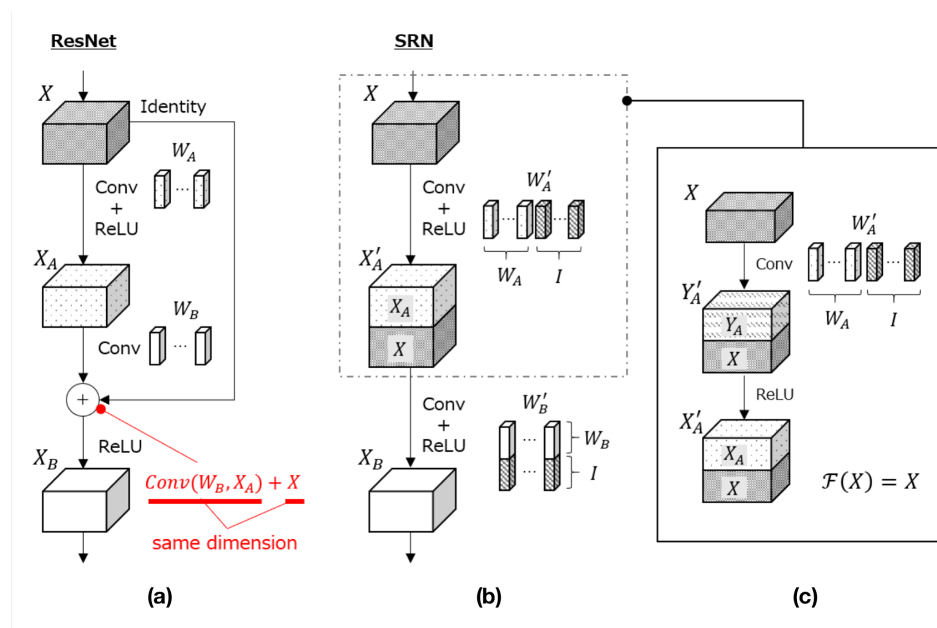


Figure VI.1: This figure illustrates the concept of SRN. (a) The conventional ResNet block. (b) The SRN block that emulates ResNet. (c) The detailed illustration of operations in first convolution and ReLU activation of the SRN block.

conduct identity mapping, identity shortcut can be emulated by the SRN block. In this way, the ResNet model is equivalent to the SRN model whose weights are partially fixed to conduct identity mapping.

Although SRN model has more FLOPs than the ResNet model, it is much easier to be accelerated by pruning. Since the SRN model has a serial architecture, we can prune any layers and reduce the computational cost drastically at the cost of relatively small degradation.

Other than the purpose of facilitating pruning, SRN can be used for enhancing the pretrained ResNet models, especially when the accuracy is more important than the complexity. Because the ResNet model is equivalent to the SRN model with the constraint that the weights are partially fixed for performing identity mapping, the SRN model can outperform the ResNet model if we unfix the fixed weights and optimize them by training. Although the basic strategy of ResNet for gaining accuracy is simply stacking the layers, our serialization strategy can be a better option to achieve better trade-off between accuracy and inference time.

The problem is that training the SRN model in the naïve way often ends up in no improvement or even degradation. The SRN model suffers some optimization

problems caused by having both the optimized weights and the unoptimized weights. In order to avoid this problem, we also propose the training scheme dedicated for SRN.

It is also worth noting that our contribution is not limited to ResNet. Other types of the DNNs that have branched architectures, such as GoogLeNet [61] and so on, can be emulated by the serial networks, and thus, the discussions in this paper are applicable to those networks.

The rest of Part VI are as follows. The related works are overviewed in Sec. 2. The details of SRN are explained in Sec. 3. The experiments are reported and some analyses are done in Sec. 4. We conclude the discussions in Sec. 5.

## 2 Related works

In [8, 19], they try to avoid this issue by adding the layer only for sampling the outputs of the remaining channels after the pruned layer, as shown in Fig. VI.2, instead of removing the pruned channels. However, we found out that this approach is less practical. Indeed, by adding the sampling layer the FLOPs can be reduced. However, at the same time, the sampling layer brings the computational overhead that makes the inference slower and may cancel the advantage of the FLOPs reduction. In addition, such a non-standard implementation is not supported by the major DNN frameworks, such as Pytorch and Tensorflow. Implementing it on one's own is costly. Therefore, the sampling layer will not be the standard solution for ResNet pruning for now.

## 3 Serialized Residual Network (SRN)

In this section, we show how to build the SRN block that emulates the ResNet block, and explain our training strategy for the SRN models.

### 3.1 How to build SRN that emulates ResNet

Fig. VI.1 illustrates the ResNet block and the equivalent SRN block, where we omit the batch normalization layers for simplicity.

Let $\otimes$ denote convolutional operation, $z$ denote ReLU function and $X \in \mathbb{R}^{d \times n \times h_w \times h_h}$ denote the feature map, where $d$ denotes the batch size, $n$, $h_w$ and $h_h$ denote the number of channels, the width and the height of $X$, respectively. The operations in

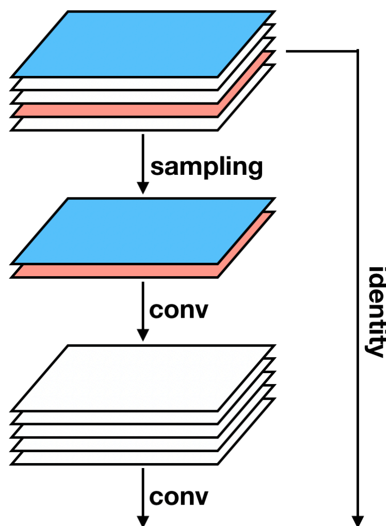


Figure VI.2: The illustration of a sampling layer. Instead of pruning the layer connected to identity shortcut, a sampling layer is inserted. The sampling layer samples the channels that are used for convolution calculation in the subsequent layer.

the ResNet block can be written as follows:

$$Y_A = W_A \otimes X, \tag{VI.1}$$

$$X_A = z(Y_A), \tag{VI.2}$$

$$Y_B = W_B \otimes X_A + X, \tag{VI.3}$$

$$X_B = z(Y_B), \tag{VI.4}$$

where $W_A, W_B \in \mathbb{R}^{n \times n \times g_w \times g_h}$ denote the kernel weights, $g_w$ and $g_h$ are the width and the height of the kernel. The feature maps $X_A, X_B, Y_A$, and $Y_B$ are $d \times n \times h_w \times h_h$ tensors.

We reproduce these operations with the SRN block. In the SRN block, the operations are as follows.

$$Y'_A = W'_A \otimes X, \tag{VI.5}$$

$$X'_A = z(Y'_A), \tag{VI.6}$$

$$Y_B = W'_B \otimes X'_A, \tag{VI.7}$$

$$X_B = z(Y_B). \tag{VI.8}$$

In Eq. (VI.5), $W'_A \in \mathbb{R}^{n\times 2n\times g_w\times g_h}$ consists of 2 sub-tensors, $W_A$ and $I \in \mathbb{R}^{n\times n\times g_w\times g_h}$, where $I$ is the kernel that conducts identity mapping ($I \otimes X = X$). Then, the output $Y'_A$ is composed of 2 sub-tensors that are identical to $Y_A$ and $X$, as shown in Fig. VI.1.

In Eq. (VI.6), $Y'_A$ is fed into $z$, and the output $X'_A$ is obtained. Assuming that $X$ is already the output of ReLU in the previous block and that every entry of $X$ is no less than 0 (This assumption basically holds true because ResNet usually has ReLU at the end of each block.), $X'_A$ still contains the sub-tensor that is identical to $X$.

The kernel $W'_B \in \mathbb{R}^{2n\times n\times g_w\times g_h}$ in Eq. (VI.7) is built by concatenating $W_B$ and $I$ so that the convolution and the addition in Eq. (VI.3) are reproduced with a single convolution. Then, the output will be identical to $Y_B$, and the final output of this block will be identical to $X_B$.

In this way, we can build the SRN block that precisely reproduces the operations of the ResNet block.

**Limitation**

It should be noted that the nonlinear function $z$ must be ReLU for the ResNet block to be emulated by the SRN block. Thus, the discussions in this paper may not be valid for some modified ResNet models with other types of activation, for example, Sigmoid, Tangent Hyperbolic, and so on. However, this limitation is not very important, because ReLU is used as standard for the modern DNNs.

### 3.2 Training strategy

The scheme for producing the SRN model is as follows.

1) Transform the pretrained ResNet model into an equivalent SRN model that has the fixed weights to reproduce the identity shortcuts.

2) Unfix the fixed weights and conduct training.

In Step 2), what we want to do is to train the whole weights of the SRN model, including the previously fixed ones. Although, we observe that the naïve training often ends up in no accuracy improvement or even degradation. Regarding to this observation, we hypothesize two problems and suggest the corresponding countermeasures. One problem is the degradation caused by training the SRN model having the well-optimized weights taken over from the ResNet model and the unoptimized

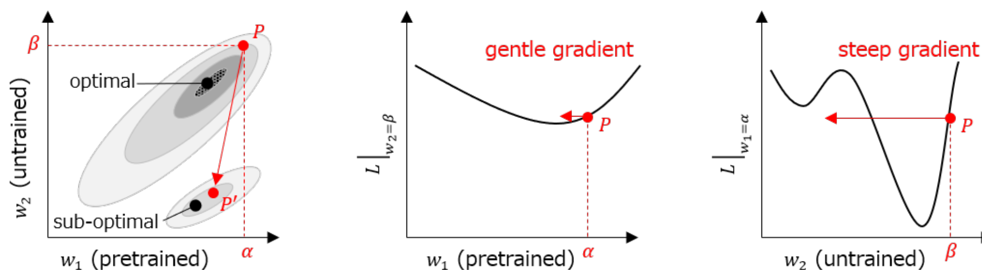


Figure VI.3: This figure illustrates the problem caused by training SRN model having a pretrained weight and an untrained weight. Left: The contour map of the loss $f$ with respect to the pretrained weight $w_1$ and the untrained weight $w_2$. Center: The graph of $f|_{w_2=\beta}$ with respect to $w_1$. Right: The graph of $f|_{w_1=\alpha}$ with respect to $w_2$. The untrained weight $w_2$ may have a steep gradient and be updated drastically by training, while the pretrained weight $w_1$ is likely to have a gentle gradient. Then, we may move from the current point $P$, which we assume is close to the optimal point, to a far point $P'$. Then, $w_1$ and $w_2$ may converge toward the sub-optimal point.

weights that are initially fixed for identity mapping. Another problem is the *side effect* of L2 regularization.

### 3.2.1 Problem caused by having both pretrained weights and untrained weights

Assume that $w_1$ is the pretrained weight taken over from ResNet, and $w_2$ is the untrained weight that was previously fixed for identity mapping. Fig. VI.3 illustrates the cost function $f$ in the weight space spanned by $w_1$ and $w_2$, and the sketches of $f$ over $w_1$ and $w_2$ around the point $P(\alpha, \beta)$ that represents the current weight values. We assume that $P$ is already near from the optimal point, since it is the result of the pre-training of the ResNet model. If we train these weights, $w_2$ may have a steep gradient and be updated significantly, because $w_2$ has not been optimized yet, while $w_1$ is an optimized weight and is likely to have a gentle gradient. Then, we may move from the current point $P$ to a far point $P'$, and $w_1$ and $w_2$ may start to converge toward the sub-optimal point that is near from $P'$, which means the training fails.

The naïve solution for this problem would be reducing the learning rate, although it would require quite a lot of iterations to converge and is computationally

inefficient.

**Alternately Unfixing Weights and Training (AUWT)**

We propose AUWT standing for *Alternately Unfixing Weights and Training.* Assuming that this problem is more likely to happen when we have too many untrained weights that may have steep gradients, we repeatedly unfix the weights partially and conduct training, in order to limit the number of the untrained weights to be trained at the same time.

For instance, we conduct AUWT in the following steps.

1) Unfix the fixed weights in the first SRN block and train the model for 1 epochs.

2) Go to the second block and do the same. It will be repeated till the final SRN block.

### 3.2.2 Side effect of L2 regularization

In many cases, we use L2 regularization to stabilize the training on the neural networks. However, L2 regularization can cause a *side effect* when we train the SRN model.

We explain the *side effect* of L2 regularization with a fully connected layer, as the same discussion is valid for convolutional layers. In the fully connected layer, the weights for identity mapping is represented by an identity matrix $E$. Let $e_{ij}$ denote the $(i, j)$ entry of $E$, $f$ denote the loss function, $a$ denote the learning rate, and $b$ denote the weight decay (the coefficient on regularization term). By feeding some training samples into the model, $e_{ij}$ is updated by $e_{ij} + \delta e_{ij}$, where $\delta e_{ij}$ is given by

$$\begin{aligned} \delta e_{ij} &= -a \frac{\partial}{\partial e_{ij}} \left( f + \frac{b}{2} \sum_{k,l} e_{kl}^2 \right) \\ &= -a \left( \frac{\partial f}{\partial e_{ij}} + b e_{ij} \right). \end{aligned} \tag{VI.9}$$

Therefore, the diagonals of $E$ tend to be strongly affected by the L2 regularization term due to their large initial values ($e_{ii} = 1$), while the rest of the weights are initially equal to 0 ($e_{ij} = 0 | i \neq j$) and are not significantly affected by L2 regularization at least in the beginning of training.

When we train the SRN model, we need to optimize the weights initialized to either 0 or 1 at the same time. In such a case, the weights initialized to 1 will be updated drastically due to the L2 regularization. Then, similarly with the problem

illustrated in Fig. VI.3, we may move away from the optimal point in the weight space, and the weights may converge toward the sub-optimal point.

**Elastic Weight Regularization**

Inspired by [62], we suggest *Elastic Weight Regularization* (EWR) to prevent the *side effect* of L2 regularization. Instead of penalizing the L2 norm of the weights, we penalize the L2 norm of the difference from the initial weight values. This is formalized as follows.

$$\begin{aligned} \delta e_{ij} &= -a\frac{\partial}{\partial e_{ij}}\left(f + \frac{b}{2}\sum_{k,l}\left(e_{kl} - e'_{kl}\right)^2\right) \\ &= -a\left(\frac{\partial f}{\partial e_{ij}} + b\left(e_{ij} - e'_{ij}\right)\right), \end{aligned} \tag{VI.10}$$

where $e'_{ij}$ denotes the initial value of $e_{ij}$. EWR prevents the weights from being too different from the initial values. With EWR, the weights initialized to 1 are not affected by the regularization term too strongly, and thus the *side effect* of L2 regularization can be avoided.

The possible drawback of EWR is the initial value dependency. As the regularized weights cannot be so different from their original values, the training result strongly depends on the initial weight values. Although, we suppose that this is not a problem when training the SRN model converted from ResNet counterpart. If the ResNet model was trained successfully, then it is intuitively reasonable to assume that its trained weights are not bad initial values. As we show in the experiments in Sec. 4.3, EWR improves SRN training compared to the normal L2 regularization.

# 4 Experiments

We conducted several experiments to verify SRN and some ablation studies to test the hypotheses mentioned in Sec. 3.2.1. We implemented the proposed method with Python 3.6 and Pytorch 1.0 [63].

## 4.1 Experiments to facilitate pruning

We evaluated SRN's ability of facilitating pruning, with the CenterNet [64] model that has ResNet-18 backbone. We transformed this backbone to SRN-18, perform pruning with REAP, and evaluated the performance of the pruned models.

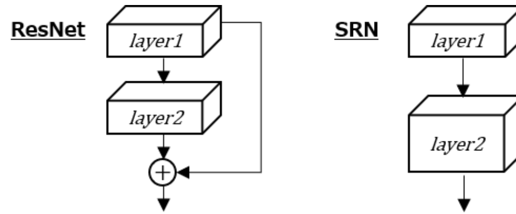


Figure VI.4: The illustration of the ResNet block and the SRN block. Due to serialization, *layer2* of SRN has an increased number of channels.

Table VI.1: The results on CenterNet.

| Backbone | mAP | FLOPs | Inf. time (msec) |
|---|---|---|---|
| ResNet-18 (baseline) | 0.274 | ×1 | 131 |
| ResNet-18-pruned (A) | 0.261 | ×0.75 | 94 |
| SRN-18-pruned (A) | **0.272** | ×0.75 | **91** |
| ResNet-18-pruned (B) | 0.248 | ×0.5 | 82 |
| SRN-18-pruned (B) | **0.262** | ×0.5 | **81** |
| ResNet-18-pruned (C) | 0.183 | ×0.25 | 67 |
| SRN-18-pruned (C) | **0.239** | ×0.25 | **57** |

We also measured the inference time per image of each model deployed on NVIDIA Jetson Nano [65], using camera demo mode of the TensorRT implementation provided in [66]. Jetson Nano is a device designed for neural network inference, and it is widely recognized/used in the industry and research.

### 4.1.1 Dataset

**MS-COCO**

MS-COCO is a popular large dataset for object detection [67]. It contains approximately 82K training images and 40K test images and 80 object classes. All the images were Following augmentation settings in [64], training and evaluation were performed on $512 \times 512$ resolution. We applied random scaling (scaling factor was 0.6 to 1.3), and random horizontal flip to the training images. We used randomly selected 5K images for pruning.

### 4.1.2 Models

We converted ResNet-18 backbone to SRN-18 that has the fixed weights, and then unfix them from the shallower side. As we unfixed the weight in a block, we trained the model for 10 epochs at $1.25 \times 10^{-5}$ learning rate, and performed pruning. We set the ratio of the pruned channels so that the FLOPs would become (A) 75%, (B) 50%, and (C) 25% of the original backbone. In SRN architecture, we can prune both *Layer1* and *Layer2* shown in Fig. VI.4. The ratio of *Layer1* and *Layer2* was tuned so that the number of remaining channels would become the same after pruning. After serializing and pruning all the blocks, we further trained the model for 20 more epochs at $1.25 \times 10^{-5}$ learning rate which was divided by 10 at 10 epochs. The rest of the training setups were the same with [64].

For ResNet, we pruned only the layers without branched paths (*Layer2* in Fig. VI.4), since we cannot prune the layers connected to identity shortcuts. The training setups are the same with the SRN models.

Just for fair comparison with the original model (with ResNet-18 backbone), we further trained the pretrained original model, which results in no apparent improvement nor degradation.

### 4.1.3 Results

The results are reported in Table VI.1. As shown in Table VI.1, the pruned the SRN models could outperform the pruned ResNet models at the same FLOPs. For instance, At $\times 0.75$ FLOPs rate, the SRN model shows a very small degradation, while the pruned ResNet model suffered more than 1% degradation in mAP. At larger FLOPs reduction, the performance gap of the ResNet model and the SRN model became even more significant. For reducing lots of FLOPs of the ResNet model, only the layers without identity shortcuts needed to be pruned, and the pruned layers with few remaining channels could not preserve the original performance. On the other hand, as any layer of the SRN model could be pruned, the model accuracy could be preserved better.

Even though the model with our *SRN-18-pruned (A)* backbone was competitive to the original model in mAP, we could achieve $\times 1.43$ speed up. In this way, even though the SRN model has more FLOPs than the ResNet model, we can effectively make the SRN model faster by performing pruning.

## 4.2 Experiments to improve accuracy

We conducted the experiments to verify the second benefit of SRN, improving the accuracy of a pretrained ResNet model. The goal of these experiments is to show that our serialization strategy can be a better option for gaining accuracy than simply stacking the layers.

### 4.2.1 Datasets

We used CIFAR-10 [55], CUB-200 [68], and STL-10 [69] datasets.

**CIFAR-10**

CIFAR-10 is a dataset for 10 class image classification [55]. It has 50K images for training and 10K images for testing. All the images have $32 \times 32$ resolution. The training images were padded by 4 pixels at each side and $32 \times 32$ random crop was applied. Random horizontal flip was applied to the training images. The test images were used as they were. We randomly selected 5K images from training dataset for pruning.

**CUB-200**

CUB-200 is a dataset for 200 classes fine-grained image classification [68]. It has approximately 6K images for training and 6K images for test. As all the images are of birds, the feature differences between some classes are slight, which makes the classification more difficult. As each image has different resolution, they were resized so that the shorter side would become 256 pixels. Then, $224 \times 224$ random crop was applied to the training images, and $224 \times 224$ center crop was applied to the test images. The random horizontal flip was applied to the training images. We randomly selected 1K images from training dataset used for pruning.

**STL-10**

STL-10 is a dataset for 10 class image classification [69]. It has 5K images for training and 8K images for test. All the images have $96 \times 96$ resolution. The training images were padded by 12 pixels at each side and $96 \times 96$ random crop was applied. The random horizontal flip was applied to the training images. The test images were used without preprocessing. We randomly selected 5K images from the training dataset and used them for pruning.

#### 4.2.2 Models

We used ResNet-20/32/44/56 for CIFAR-10 and ResNet-18/34 for CUB-200 and STL-10.

**Models for CIFAR-10:** We transformed the pretrained ResNet models to the SRN models with the partially fixed weights. As we unfix the weights in each block, we conducted training for 10 epochs at $10^{-2}$ learning rate (This is AUWT step.). Finally, we conducted training for 200 epochs at $10^{-2}$ learning rate which was divided by 10 at 100 epochs. For other experimental setups, we followed [11].

We also trained the SRN models from scratch with the naïve training scheme other than the training strategy mentioned in Sec. 3.2 for comparison. The training setups were the same with [11]. These models are referred as "SRN-x-naïve".

**Models for CUB-200 and STL-10:** For CUB-200 and STL-10, we prepared the baseline models in two different ways: 1) Training from scratch and 2) Fine-tuning the ResNet model pretrained with ImageNet dataset [54]. The experimental setups were the same with the experiments of CIFAR-10 except that the weight decay for regularization was set to $1.25 \times 10^{-3}$ for STL-10 and $2 \times 10^{-4}$ for CUB-200.

#### 4.2.3 Results on CIFAR-10

The results are shown in Table VI.2. The analyses and the discussions are as follows.

**Can SRN outperform ResNet?**

The SRN models consistently suffer lower error than the ResNet models. For instance, SRN-32 suffers 6.32% test error, while ResNet-32 suffers 7.40%. This is not only because SRN-32 have more trainable weights than ResNet-32 but also because we have trained the weights of the SRN models in a proper way mentioned in Sec. 3.2.

A remarkable observation is that SRN-32 outperforms even ResNet-56, and is much faster than ResNet-56. Although stacking the layer is ResNet's basic strategy for improving the performance, this result implies that our serialization approach can be the better option than stacking the layers.

**Comparison with the SRN models trained from scratch in the naïve way**

In order to verify our training strategy mentioned in Sec. 3.2, we compared the

Table VI.2: The results on CIFAR-10. The SRN models consistently outperform the ResNet models. Unexpectedly, the SRN models run faster than the ResNet models in some cases, despite the doubled FLOPs. *We did not measure the inference time of SRN-x-naïve as it must be the same with SRN-X.

| Model | Test error (%) | FLOPs | Inf. time (msec) at batch size | | |
|---|---|---|---|---|---|
| | | | 1 | 4 | 16 |
| ResNet-20 | 8.46 | **40.5M** | 191.1 | 55.0 | 23.5 |
| SRN-20 | **7.19** | 80.6M | **173.7** | 50.9 | 25.3 |
| SRN-20-naïve | 7.76 | 80.6M | -* | -* | -* |
| SRN-20-pruned | 8.17 | **40.5M** | 174.7 | **49.8** | **20.4** |
| ResNet-32 | 7.40 | **68.8M** | 268.7 | 73.2 | 29.2 |
| SRN-32 | **6.32** | 137.2M | 247.0 | **67.9** | 36.5 |
| SRN-32-naïve | 8.66 | 137.2M | -* | -* | -* |
| SRN-32-pruned | 7.12 | **68.8M** | **245.6** | 68.0 | **28.4** |
| ResNet-44 | 7.18 | **97.1M** | 347.6 | 91.0 | 36.6 |
| SRN-44 | **6.03** | 193.9M | 321.8 | 85.1 | 46.6 |
| SRN-44-naïve | 9.58 | 193.9M | -* | -* | -* |
| SRN-44-pruned | 6.98 | **97.1M** | **310.5** | **84.1** | **35.0** |
| ResNet-56 | 6.63 | **125.4M** | 432.6 | 111.9 | 44.9 |
| SRN-56 | **5.62** | 250.5M | 397.9 | 102.2 | 56.9 |
| SRN-56-naïve | 11.52 | 250.5M | -* | -* | -* |
| SRN-56-pruned | 6.57 | **125.4M** | **388.7** | **102.1** | **42.1** |

SRN models trained in the proposed scheme and the ones trained from scratch in the naïve way. The trend was that the deeper the architecture was, the worse the performance of SRN-x-naïve model became. On the other hand, SRN trained in the proposed way was robust and gained accuracy as the depth increases. This result supports that the proposed training scheme is effective for stabilizing and improving the training for the SRN models.

**FLOPs and measured inference time**

We measured the inference time on NVIDIA Jetson Nano. We report the average inference time per image in Table VI.2.

Unexpectedly, when the batch size is 1 and 4, SRN models are faster than the ResNet models at the same depth, even though SRN models have approximately twice as many FLOPs as the ResNet models. For example, SRN-20 needs 17.3 msec per image while ResNet-20 needs 19.1 msec. This observation is counter-intuitive,

however, it can be explained with 2 factors.

One is that the SRN models have fewer computational steps than the ResNet models. The ResNet models have the step of addition at the end of each identity shortcut. On the other hand, the SRN models do not have it. Even though the operation of addition is a less expensive operation, it still requires some computational overheads.

The other one is that the resolution of CIFAR-10 images is only $32 \times 32$. The FLOPs required for convolutional operations on the feature maps are relatively few, and such operations can be fully parallelized and significantly accelerated thanks to the recent developments of hardware and libraries. Thus, in this case, the increase of the FLOPs caused by serialization did not lead to the increase of the inference time. In fact, at larger batch size, the ResNet models are faster than the SRN models, because the FLOPs required for each batch increased and became dominant for the inference time.

On the other hand, the depth of the architecture always affects the inference time. For instance, SRN-32 runs much faster than ResNet-56, even though SRN-32 has more FLOPs than ResNet-56. This is because the operations in the different layers must be conducted sequentially, while the operations within each layer can be parallelized. Therefore, unless the FLOPs is critically dominant for inference time, the shallower and more accurate SRN model could be a better option than the deeper and and less accurate ResNet model.

**Applying pruning method to SRN models**

We conducted an extra experiment to further analyze the counter-intuitive observation that the SRN models are faster than the ResNet models despite the significantly increased FLOPs. We performed pruning on the layers of SRN having the doubled channels due to serialization (*layer2* in Fig. VI.4) so that the number of channels in these layers would become halved. REAP [6] was used for pruning. The results are summarized in Table VI.2.

As shown in Table VI.2, the pruned SRN models are faster than the ResNet models. Note that the pruned SRN models have the same architecture with the ResNet models, except that the pruned SRN models do not have identity shortcuts. This result confirms that the the presence of identity shortcuts somewhat affects the inference time.

It is also worth noting that the pruned SRN models were still better than the ResNet models in accuracy. By serializing and pruning the ResNet models, we can

Table VI.3: The results on CUB-200 and STL-10. Here, "fine-tuned" means that the baseline model (ResNet) was trained by fine-tuning the ResNet model pretrained with ImageNet dataset.

| Dataset | Model | Test error (%) |
|---|---|---|
| CUB-200 | ResNet-18 | 46.83 |
| | SRN-18 | **43.86** |
| | ResNet-34 | 46.17 |
| | SRN-34 | **42.27** |
| | ResNet-18 (fine-tuned) | 23.06 |
| | SRN-18 (fine-tuned) | **22.29** |
| | ResNet-34 (fine-tuned) | 20.75 |
| | SRN-34 (fine-tuned) | **19.70** |
| STL-10 | ResNet-18 | 23.62 |
| | SRN-18 | **21.79** |
| | ResNet-34 | 21.89 |
| | SRN-34 | **20.87** |
| | ResNet-18 (fine-tuned) | 25.45 |
| | SRN-18 (fine-tuned) | **23.65** |
| | ResNet-34 (fine-tuned) | 24.77 |
| | SRN-34 (fine-tuned) | **22.88** |

take the benefits in both the computational complexity and the accuracy.

### 4.2.4 Results on CUB-200

As shown in Table VI.3, the SRN models consistently outperforms the ResNet models. Especially, in the case of training from scratch, we could significantly outperform ResNet. SRN-18 outperforms ResNet-18 by approximately 3% and even ResNet-34 by 2.3% in test error. In this way, our serialization strategy can be a better option than simply stacking the layers.

Although we succeeded to outperform the ResNet models in the case of fine-tuning, the improvement in accuracy was less significant than that in the case of training from scratch. We suppose that the fine-tuned models had already been trained very well and had smaller room for improvement.

Table VI.4: Ablation studies with CIFAR-10, CUB-200, and STL-10.

| Dataset | Model | Test error (%) | | |
|---|---|---|---|---|
| | | Ours | w/o EWR | w/oAUWT |
| CIFAR-10 | SRN-20 | **7.19** | 7.65 | 7.57 |
| | SRN-32 | **6.32** | 6.61 | 6.57 |
| | SRN-44 | **6.03** | 6.57 | 6.54 |
| | SRN-56 | **5.62** | 6.13 | 6.81 |
| CUB-200 | SRN-18 | **43.86** | 44.52 | 44.10 |
| | SRN-34 | **42.27** | 44.93 | 42.91 |
| STL-10 | SRN-18 | **21.79** | 22.73 | 21.87 |
| | SRN-34 | **20.87** | 80.05 | 21.29 |

### 4.2.5 Results on STL-10

Consistently with the results on CUB-200, the SRN models outperform the ResNet models. SRN-18 model is better than even the deeper SRN-34 model. This is also a case example that converting ResNet to SRN can be a better option than simply stacking the layers of ResNet.

In this experiment, the fine-tuned ResNet models were worse than the ResNet models trained from scratch. This is probably because of the gap of the resolution of the input images. The pretrained models had been optimized for $224 \times 224$ resolution in the pre-training with ImageNet dataset, and could not be properly re-optimized for $96 \times 96$ resolution of the target task. It is well known that utilizing the weights of the models pretrained with the larger task is a good strategy for producing the accurate models for the smaller tasks, however, this strategy did not work well in this experiment. In any case, our serialization method is a good option for gaining accuracy.

## 4.3 Ablation studies and analyses

We conducted the ablation studies with CIFAR-10, CUB-200, and STL-10 to test the hypotheses that we mention in Sec. 3.2.1. We trained the SRN models in 3 different ways:

1) Our proposed scheme presented in Sec. 3.2. (referred as "Ours").

2) With normal L2 regularization instead of EWR (referred as "w/o EWR").

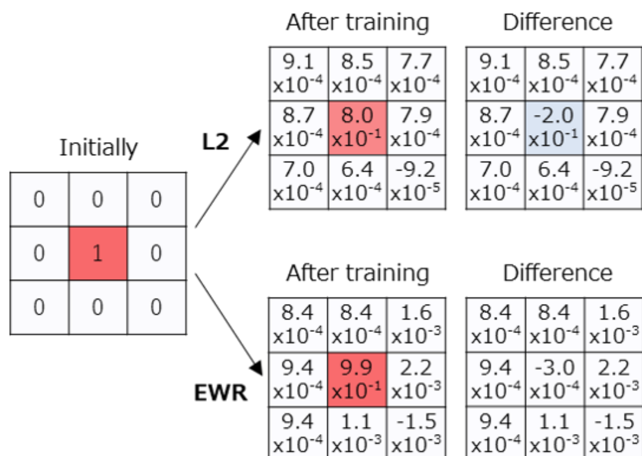


Figure VI.5: The visualized kernel weights of SRN-34 on STL-10. Left: The initial weights; Center: The weights after training with L2 regularization and EWR; Right: The difference of the trained weights and the initial weights. Red is positive, blue is negative, and white is close to 0. With the conventional L2 regularization, the weight initialized to 1 was updated drastically.

3) Unfix all the weights in all the SRN blocks at once and conduct training instead of AUWT (referred as "w/o AUWT").

The setups, results, and discussions are as follows.

#### 4.3.1 What if we use conventional L2 regularization instead of EWR?

We trained the SRN models without EWR. The rest of the setups are the same with the above experiments in this section.

As shown in Table VI.4, we suffer higher error with L2 regularization instead of EWR. Especially, a catastrophic degradation was observed when we trained SRN-34 on STL-10 with L2 regularization. Fig. VI.5 shows some kernel weights of this model that had been initialized for performing identity mapping and the same weights after training with L2 regularization and EWR. The weights initialized to 1 was changed much more drastically than the weights initialized to 0, with L2 regularization. On the other hand, this problem could be avoided with EWR.

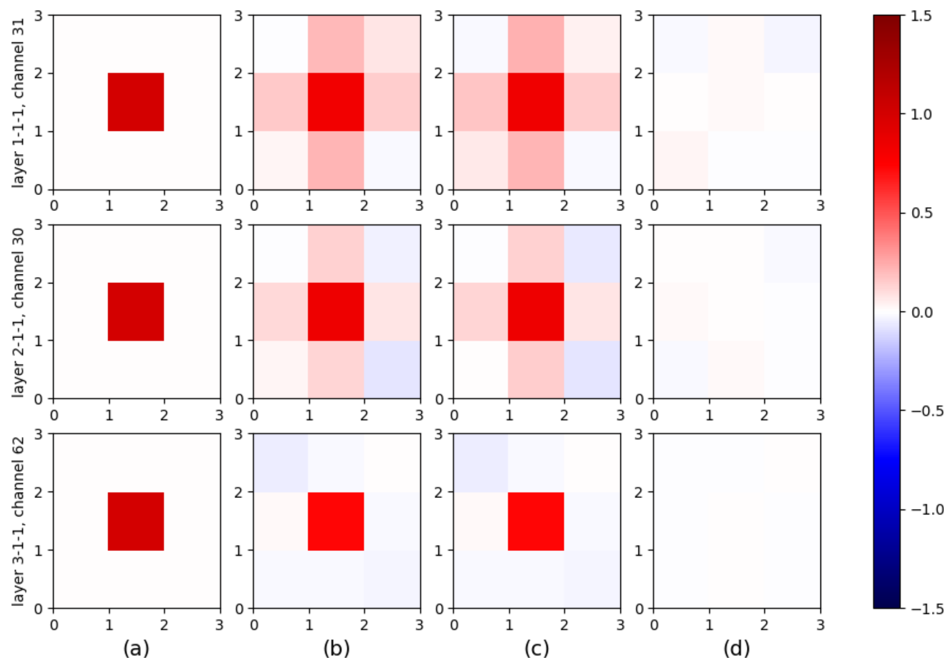


Figure VI.6: Drawings of the kernels that are initially for identity mapping. (a) Weights right after serialization. (b) Weights after SRN training. (c) Weights after reconstruction. (d) Difference of weights before and after reconstruction.

### 4.3.2 What if we unfix all the fixed weights at once and start training instead of AUWT?

We trained the SRN models without AUWT. Concretely, we unfixed the fixed weights in the whole SRN model at once and conducted training for 200 epochs. For each model, we set pruning ratio to $10^{-2}$ and divided it by 10 at 100 epochs. The rest of the setups stay the same.

As a result, we observe some degradation without AUWT. This experimental results implies that our AUWT is effective for training the SRN models. We suppose this is because lots of the weights that had not been optimized yet were updated drastically when we started training, and the whole weights have converged toward the worse sub-optimal point, as shown in Fig. VI.3.

#### 4.3.3 What the identity mapping portion of kernels will be like after reconstruction?

We checked how the kernels that are initially for identity mapping will change after applying REAP. In Fig. VI.6, we show 3 channels of kernels from 3 different layers of ResNet-20 model trained with CIFAR-10. We pruned 50% of the channels in those 3 layers, respectively, and observed how these weights got changed. Fig. VI.6 (b) shows the kernels right after SRN training, (c) shows the weights after pruning and reconstruction, and (d) shows the difference of (b) and (c). We do not see drastic change in those weights.

## 5 Summary of Part VI

We presented a technique to convert a ResNet model into a serial model, which we call Serialized Residual Network (SRN), in order to facilitate pruning on the pretrained ResNet models. The ResNet model is not easy to accelerated by pruning due to its architecture with identity shortcuts. Although, it can be emulated by the SRN model. Even though the SRN model has more FLOPs than the equivalent ResNet model, it is easier to be accelerated by pruning, because the SRN model has a serial architecture and any layer of it can be pruned. We can finally produce the faster and more accurate model by once serializing the ResNet model and then performing pruning. Another benefit of SRN is to further improve the accuracy of the pretrained ResNet model. By optimizing the weights that are added by serialization, we can improve the accuracy. As the naïve end-to-end training often fails due to several optimization problems, we suggested a specific training scheme for SRN. The experimental results support the effectiveness of our SRN.

# Part VII
# Summary

In this thesis, we presented the methods for pruning the pretrained DNN models effectively. The proposed methods include two pruning methods, Neuro-Unification (NU) and Reconstruction Error Aware Pruning (REAP), and two facilitation methods for pruning, Pruning Ratio Optimizer (PRO), and Serialized Residual Network (SRN). These methods offer a practical solution for those who want to use large DNN models in resource-limited environments, such as smartphones, drones, and so on.

The biggest highlight of this thesis is REAP. REAP is theoretically well designed for preserving the accuracy of the model while reducing the model's redundancy. Among the methods that perform pruning based on layer-wise error, no other method is as good as REAP in terms of minimizing the error, to our best knowledge. Moreover, since REAP requires significant amount of computation for selecting the neurons to be pruned, we presented an efficient algorithm based on biorthogonal system. This algorithm is a novel usage of biorthogonal system.

PRO is a method to optimize the pruning ratio in each layer efficiently. The idea of PRO is to repeat selecting the layer where pruning will have the least impact on the outputs in the final layer of the model, and pruning some neurons in the selected layer. With REAP and PRO, we can conduct compression and optimize the architecture of the pruned model simultaneously.

SRN is a method to facilitate pruning on ResNet. The limitation of structured pruning on ResNet is that the layers with branched paths cannot be pruned. We noticed that the ResNet architecture is equivalent to a specific case of a serial architecture. Therefore, ResNet can be converted to an serial form which we call SRN. Once converted, we can reduce its redundancy drastically by pruning, as SRN has a serial architecture and its any layer can be pruned. SRN improves the practicality of pruning.

In the future, we plan to put the proposed method to practical use. By inputting the target model, some data, and overall compression ratio, the system identifies the layers to be pruned by performing structural analysis, serializes the ResNet architecture (if exists), optimizes the pruning rate with PRO, and conducts pruning with REAP. We aim to design the system in such a way that user does not need any special knowledge, and we expect the system to be widely used in the industry.

# Acknowledgements

I would like to thank all the professors in Faculty of Systems Engineering, Wakayama University for their guidance in writing this thesis. Especially, I appreciate Prof. Wada, Prof. Amano for daily meetings and discussions, Prof. Kazama, Prof. Kurita (Hiroshima University), and Lecturer Hachiya for having constructive discussions in degree examination. I also thank the students in laboratory for their cooperation in discussions, Ms. Fujiwara and other administrative officers for supporting my laboratory life, and my family for their support in my daily life.

# References


[1] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

[2] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

[3] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *CoRR*, 2015.

[4] X. Wang, X. Wang, and S. Mao. Rf sensing in the internet of things: A general deep learning framework. *IEEE Communications Magazine*, Vol. 56, No. 9, pp. 62–67, 2018.

[5] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 598–605, 1990.

[6] K. Kamma and T. Wada. Reconstruction error aware pruning for accelerating neural networks. In *Proceedings of International Symposium on Visual Computing (ISVC)*, pp. 59–72, 2019.

[7] B. Hassibi, D. G. Stork, and G. J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pp. 293–299, 1993.

[8] 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)*, pp. 1389–1397, 2017.

[9] J. 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)*, pp. 5058–5066, 2017.

[10] K. Kamma, Y. Isoda, S. Inoue, and T. Wada. Behavior-based compression for convolutional neural networks. In *Proceedings of International Conference on Image Analysis and Recognition (ICIAR)*, pp. 427–439, 2019.

[11] 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)*, pp. 770–778, 2016.

[12] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440, 2015.

[13] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *CoRR*, 2015.

[14] X. Dong, S. Chen, and S. Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 4857–4867, 2017.

[15] A. Aghasi, A. Abdi, N. Nguyen, and J. Romberg. Net-trim: Convex pruning of deep neural networks with performance guarantee. In *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 3177–3186, 2017.

[16] 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 Conference on Neural Information Processing Systems (NIPS)*, pp. 881–892, 2018.

[17] H. Wang, Q. Zhang, Y. Wang, and H. Hu. Structured probabilistic pruning for convolutional neural network acceleration. In *Proceedings of British Machine Vision Conference (BMVC)*, 2018.

[18] V. Babu S. Srinivas. Data-free parameter pruning for deep neural networks. In *Proceedings of British Machine Vision Conference (BMVC)*, 2018.

[19] 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)*, pp. 2736–2744, 2017.

[20] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 1135–1143, 2015.

[21] M. Zhu and S. Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *CoRR*, 2017.

[22] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient transfer learning. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

[23] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu. Reshaping deep neural network for fast decoding by node-pruning. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 245–249, 2014.

[24] S. Park, J. Lee, S. Mo, and J. Shin. Lookahead: A far-sighted alternative of magnitude-based pruning. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

[25] R. Yu, A. Li, C. Chen, J. Lai, V. I. Morariu, X. Han, M. Gao, C. 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)*, pp. 9194–9203, 2018.

[26] . Liu, . Wang, H. Foroosh, M. Tappen, and M. Penksy. Sparse convolutional neural networks. In *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814, 2015.

[27] G. Xie, J. Wang, T. Zhang, J. Lai, R. Hong, and G. Qi. Interleaved structured sparse convolutional neural networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8847–8856, 2018.

[28] L. Zhao, Q. Hu, and W. Wang. Heterogeneous feature selection with multimodal deep neural networks and sparse group lasso. *IEEE Transactions on Multimedia*, Vol. 17, No. 11, pp. 1936–1948, 2015.

[29] J. Park, S. R. Li, W. Wen, H. Li, Y. Chen, and P. Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *CoRR*, 2016.

[30] J. Xue, J. Li, and Y. Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 2365–2369, 2013.

[31] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of British Machine Vision Conference (BMVC)*, 2014.

[32] J. Kossaifi, A. Toisoul, A. Bulat, Y. Panagakis, T. Hospedales, and M. Pantic. Factorized higher-order cnns with an application to spatio-temporal emotion estimation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6060–6069, 2020.

[33] W. Wen, C. Xu, C. Wu, Y. Wang, Y. Chen, and H. Li. Coordinating filters for faster deep neural networks. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 658–666, 2017.

[34] J. Ye, L. Wang, G. Li, D. Chen, S. Zhe, X. Chu, and Z. Xu. Learning compact recurrent neural networks with block-term tensor decomposition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9378–9387, 2018.

[35] X. Yu, T. Liu, X. Wang, and D. Tao. On compressing deep models by low rank and sparse decomposition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7370–7379, 2017.

[36] M. Courbariaux, Y. Bengio, and J. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 3123–3131, 2015.

[37] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of European Conference on Computer Vision (ECCV)*, pp. 525–542, 2016.

[38] Z. Li, B. Ni, W. Zhang, X. Yang, and W. Gao. Performance guaranteed network acceleration via high-order residual quantization. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 2584–2592, 2017.

[39] F. Tung and G. Mori. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7873–7882, 2018.

[40] X. Xu, Q. Lu, L. Yang, S. Hu, D. Chen, Y. Hu, and Y. Shi. Quantization of fully convolutional networks for accurate biomedical image segmentation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8300–8308, 2018.

[41] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *CoRR*, 2015.

[42] S. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, pp. 5191–5198, 2020.

[43] J. Yim, D. Joo, J. Bae, and J. Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7130–7138, 2017.

[44] T. Elsken, J. H. Metzen, and F. Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[45] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2018.

[46] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing. Neural architecture search with bayesian optimisation and optimal transport. *CoRR*, 2018.

[47] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *CoRR*, 2016.

[48] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *Proceedings of International Conference on Machine Learning (ICML)*, pp. 4095–4104, 2018.

[49] C. Hsu, S. Chang, D. Juan, J. Pan, Y. Chen, W. Wei, and S. Chang. Monas: Multi-objective neural architecture search using reinforcement learning. *CoRR*, 2018.

[50] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of European Conference on Computer Vision (ECCV)*, pp. 19–34, 2018.

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

[52] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. V. Gool. Ai benchmark: All about deep learning on smartphones in 2019. *CoRR*, 2019.

[53] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, 2014.

[54] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, 2009.

[55] Cifar-10 (canadian institute for advanced research). `https://academictorrents.com/details/463ba7ec7f37ed414c12fbb71ebf6431eada2d7a`. (accessed on 08/01/2021).

[56] Proper implementation of resnet-s for cifar10/100 in pytorch that matches description of the original paper. `https://github.com/akamaster/pytorch_resnet_cifar10`. (accessed on 08/01/2021).

[57] G. Huang, Z. Liu, v. d. M. Laurens, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4700–4708, 2017.

[58] A. Khosla, N. Jayadevaprakash, B. Yao, and L. Fei-Fei. Novel dataset for fine-grained image categorization. In *First Workshop on Fine-Grained Visual Categorization*, 2011.

[59] Channel pruning for accelerating very deep neural networks. `https://github.com/yihui-he/channel-pruning`. (accessed on 08/01/2021).

[60] Y. He, J. Lin, Z. Liu, H. Wang, L. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of European Conference on Computer Vision (ECCV)*, pp. 784–800, 2018.

[61] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[62] W. Hu, Z. Lin, B. Liu, C. Tao, Z. Tao, J. Ma, D. Zhao, and R. Yan. Overcoming catastrophic forgetting via model adaptation. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.

[63] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[64] K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, and Q. Tian. Centernet: Keypoint triplets for object detection. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 6569–6578, 2019.

[65] Nvidia jetpack sdk. `https://developer.nvidia.com/embedded-computing`. (accessed on 08/01/2021).

[66] Tensorrt-centernet. `https://github.com/CaoWGG/TensorRT-CenterNet`. (accessed on 08/01/2021).

[67] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *Proceedings of European Conference on Computer Vision (ECCV)*, pp. 740–755, 2014.

[68] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. The caltech-ucsd birds-200-2011 dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[69] A. Coates, A. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of International Conference on Artificial Intelligence and Statistics (ICAIS)*, Vol. 15, pp. 215–223, 2011.

# Appendix

## A DNN model architectures

This section explains the architectures of the DNN models used in our experiments.

### A.1 VGG16

VGG16 [2] has a typical convolutional network architecture, as shown in Fig. A.1. It has 13 convolutional layers, and 3 fully connected layers. The convolutional layers are divided into 5 blocks with $2 \times 2$ max pooling layers at the ends. The whole architecture of VGG16 is shown in Table A.1.

### A.2 ResNet-18 and ResNet-56

ResNet [11] is a series of the DNN models that have the identity shortcuts at every second (or third) layers, as shown in Fig. A.1. ResNet-18 and ResNet-56 that we used have been trained with ImageNet and CIFAR-10 datasets, respectively. The whole architecture of ResNet-18 is shown in Table A.2.

### A.3 DenseNet

DenseNet [57] is a series of the DNN models whose convolutional layers have a lot of skip connections. DenseNet-121 has 4 "Dense Blocks", and in each block, every second layer has the skip connections to each other, as shown in Fig. A.1. The whole architecture is shown in Table A.3

## B Gram-schmidt process-based algorithm for neuron selection in REAP

In this section, we explain the substitute algorithm for the biorthogonal system based algorithm for computing the reconstruction errors. In this substitute algorithm, the Gram-Schmidt process is used iteratively so that the vector initialized to $\boldsymbol{x}_i$ will converge into $\boldsymbol{r}_i$. When the number of the neurons are very large, this Gram-Schmidt process based algorithm is faster than the biorthogonal based one. We also provide the proof that this algorithm will correctly converges and show some case studies.

Table A.1: The architecture of VGG16. The output shape are $Width \times Height \times Channels\#$.

| Layer | Type | Output shape |
|---|---|---|
| *Input* | - | $224 \times 224 \times 3$ |
| *Conv1-1* | $3 \times 3$ conv | $224 \times 224 \times 64$ |
| *Conv1-2* | $3 \times 3$ conv | $224 \times 224 \times 64$ |
| *Pool* | $2 \times 2$ maxpool, $2 \times 2$ stride | $112 \times 112 \times 64$ |
| *Conv2-1* | $3 \times 3$ conv | $112 \times 112 \times 128$ |
| *Conv2-2* | $3 \times 3$ conv | $112 \times 112 \times 128$ |
| *Pool* | $2 \times 2$ maxpool, $2 \times 2$ stride | $56 \times 56 \times 128$ |
| *Conv3-1* | $3 \times 3$ conv | $56 \times 56 \times 256$ |
| *Conv3-2* | $3 \times 3$ conv | $56 \times 56 \times 256$ |
| *Conv3-3* | $3 \times 3$ conv | $56 \times 56 \times 256$ |
| *Pool* | $2 \times 2$ maxpool, $2 \times 2$ stride | $28 \times 28 \times 256$ |
| *Conv4-1* | $3 \times 3$ conv | $28 \times 28 \times 512$ |
| *Conv4-2* | $3 \times 3$ conv | $28 \times 28 \times 512$ |
| *Conv4-3* | $3 \times 3$ conv | $28 \times 28 \times 512$ |
| *Pool* | $2 \times 2$ maxpool, $2 \times 2$ stride | $14 \times 14 \times 512$ |
| *Conv5-1* | $3 \times 3$ conv | $14 \times 14 \times 512$ |
| *Conv5-2* | $3 \times 3$ conv | $14 \times 14 \times 512$ |
| *Conv5-3* | $3 \times 3$ conv | $14 \times 14 \times 512$ |
| *Pool* | $2 \times 2$ maxpool, $2 \times 2$ stride | $7 \times 7 \times 512$ |
| *FC1* | fully connected | 4096 |
| *FC2* | fully connected | 4096 |
| *Output* | fully connected | 1000 |

Table A.2: The architecture of ResNet-18. The output shape are $Width \times Height \times Channels\#$.

| Layer | Type | Output shape |
|---|---|---|
| *Input* | - | $224 \times 224 \times 3$ |
| *Conv* | $7 \times 7$ conv, $2 \times 2$ stride | $112 \times 112 \times 64$ |
| *Pool* | $2 \times 2$ maxpool, $2 \times 2$ stride | $56 \times 56 \times 64$ |
| *Conv1-1* | $3 \times 3$ conv | $56 \times 56 \times 64$ |
| *Conv1-2* | $3 \times 3$ conv | $56 \times 56 \times 64$ |
| *Shortcut* | add | $56 \times 56 \times 64$ |
| *Conv1-3* | $3 \times 3$ conv | $56 \times 56 \times 64$ |
| *Conv1-4* | $3 \times 3$ conv | $56 \times 56 \times 64$ |
| *Shortcut* | add | $56 \times 56 \times 64$ |
| *Conv2-1* | $3 \times 3$ conv, $2 \times 2$ stride | $28 \times 28 \times 128$ |
| *Conv2-2* | $3 \times 3$ conv | $28 \times 28 \times 128$ |
| *Shortcut* | add | $28 \times 28 \times 128$ |
| *Conv2-3* | $3 \times 3$ conv | $28 \times 28 \times 128$ |
| *Conv2-4* | $3 \times 3$ conv | $28 \times 28 \times 128$ |
| *Shortcut* | add | $28 \times 28 \times 128$ |
| *Conv3-1* | $3 \times 3$ conv, $2 \times 2$ stride | $14 \times 14 \times 256$ |
| *Conv3-2* | $3 \times 3$ conv | $14 \times 14 \times 256$ |
| *Shortcut* | add | $14 \times 14 \times 256$ |
| *Conv3-3* | $3 \times 3$ conv | $14 \times 14 \times 256$ |
| *Conv3-4* | $3 \times 3$ conv | $14 \times 14 \times 256$ |
| *Shortcut* | add | $14 \times 14 \times 256$ |
| *Conv4-1* | $3 \times 3$ conv, $2 \times 2$ stride | $7 \times 7 \times 512$ |
| *Conv4-2* | $3 \times 3$ conv | $7 \times 7 \times 512$ |
| *Shortcut* | add | $7 \times 7 \times 512$ |
| *Conv4-3* | $3 \times 3$ conv | $7 \times 7 \times 512$ |
| *Conv4-4* | $3 \times 3$ conv | $7 \times 7 \times 512$ |
| *Shortcut* | add | $7 \times 7 \times 512$ |
| GAP | global avgopool | $1 \times 1 \times 512$ |
| *FC* | fully connected | 1000 |

Table A.3: The architecture of DenseNet-121. The output shape are $Width \times Height \times Channels\#$.

| Layer | Type | Output shape |
|---|---|---|
| *Input* | - | $224 \times 224 \times 3$ |
| *Conv*<br>*Pool* | $7 \times 7$ conv, $2 \times 2$ stride<br>$2 \times 2$ maxpool, $2 \times 2$ stride | $112 \times 112 \times 64$<br>$56 \times 56 \times 64$ |
| Dense Block (1) | $1 \times 1$ conv, $3 \times 3$ conv (repeat 6 times) | $56 \times 56 \times 64$ |
| Transition (1) | $1 \times 1$ conv<br>$2 \times 2$ avgpool, $2 \times 2$ stride | $56 \times 56 \times 128$<br>$28 \times 28 \times 128$ |
| Dense Block (2) | $1 \times 1$ conv, $3 \times 3$ conv (repeat 12 times) | $28 \times 28 \times 128$ |
| Transition (2) | $1 \times 1$ conv<br>$2 \times 2$ avgpool, $2 \times 2$ stride | $28 \times 28 \times 256$<br>$14 \times 14 \times 256$ |
| Dense Block (3) | $1 \times 1$ conv, $3 \times 3$ conv (repeat 24 times) | $14 \times 14 \times 256$ |
| Transition (3) | $1 \times 1$ conv<br>$2 \times 2$ avgpool, $2 \times 2$ stride | $14 \times 14 \times 512$<br>$7 \times 7 \times 512$ |
| Dense Block (4) | $1 \times 1$ conv, $3 \times 3$ conv (repeat 16 times) | $7 \times 7 \times 512$ |
| GAP | global avgpool | $1 \times 1 \times 512$ |
| *FC* | fully connected | 1000 |

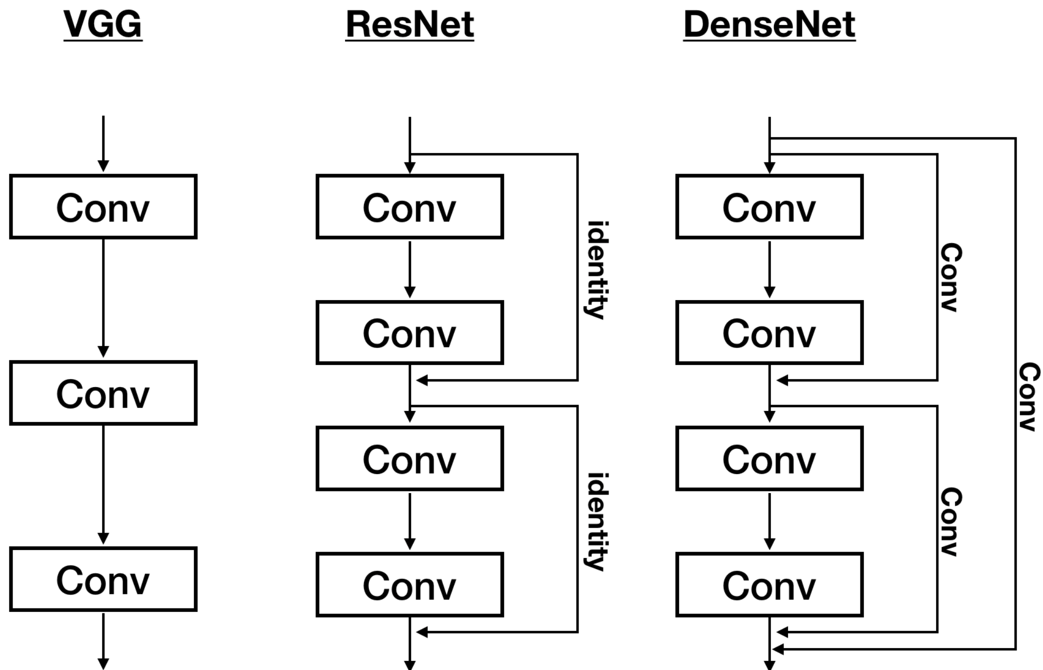


Figure A.1: The partial architectures of the DNNs used in our experiments.

### B.1 Gram-Schmidt process based algorithm

We try to accelerate the computation for the $\boldsymbol{r}$-s by parallel computing. However, it is not smart to implement (IV.5) as is, because the naive solution of the least squares problem is not memory-efficient, and the degree of parallelism would be limited because of the heavy memory requirement. Therefore, we have developed a memory efficient algorithm that uses Gram-Schmidt process.

Solving (IV.5) is equivalent to computing the orthogonal projection of $\boldsymbol{x}_i$ onto the subspace $\mathcal{U}_{(\mathcal{I}\backslash\{i\})}$ spanned by $\{\boldsymbol{x}_j | j \in \mathcal{I}\backslash\{i\}\}$, as shown in Fig. IV.3. Then, we have

$$\begin{aligned} \boldsymbol{r}_i &= \boldsymbol{x}_i - \sum_{j\in\mathcal{I}\backslash\{i\}} a^*_{ij}\boldsymbol{x}_i \\ &= \boldsymbol{x}_i - M_{(\mathcal{I}\backslash\{i\})}M^T_{(\mathcal{I}\backslash\{i\})}\boldsymbol{x}_i, \end{aligned} \tag{B.1}$$

where $M_{(\mathcal{I}\backslash\{i\})}$ represents the orthogonal basis of $\mathcal{U}_{(\mathcal{I}\backslash\{i\})}$. However, as $M_{(\mathcal{I}\backslash\{i\})}$ is typically a large matrix, computing $M_{(\mathcal{I}\backslash\{i\})}$ for each $i \in \mathcal{I}$ is not memory-efficient, therefore, this computation cannot be highly parallelized.

We obtain the approximate solution of the $\boldsymbol{r}$-s and the $a$-s by applying Gram-

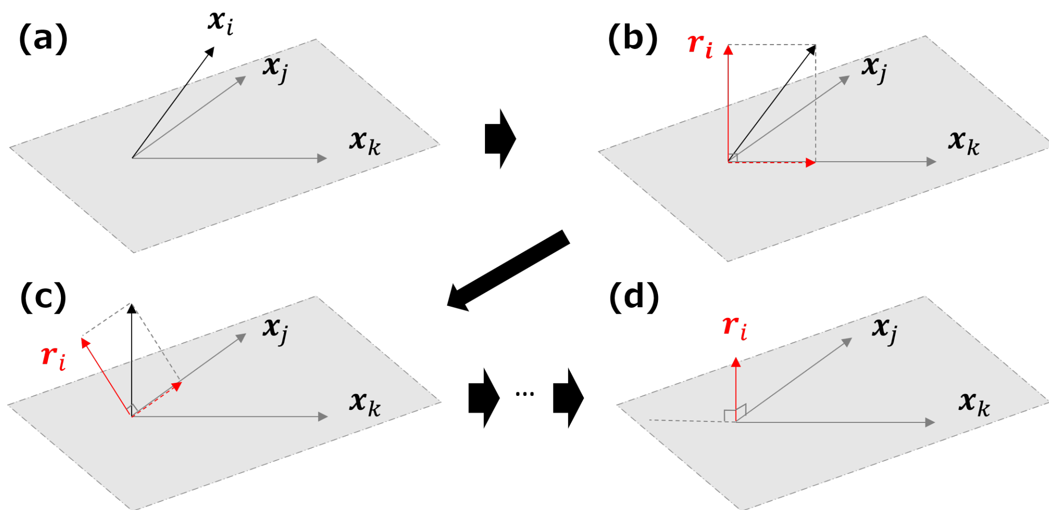


Figure B.1: An example of the Gram-Schmidt process based algorithm. We want to obtain $\boldsymbol{r}_i$, the residual of the projection of $\boldsymbol{x}_i$ onto the subspace spanned by $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$. (a) $\boldsymbol{r}_i$ is initialized to $\boldsymbol{x}_i$. (b) $\boldsymbol{r}_i$ is projected onto $\boldsymbol{x}_k$ and is replaced by the residual. (c) $\boldsymbol{r}_i$ is projected onto $\boldsymbol{x}_j$ and is replaced by the residual. Note that $\boldsymbol{r}_i$ becomes NOT orthogonal to $\boldsymbol{x}_k$ again, as $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$ are not orthogonal. (d) After repeating above procedures lots of times, $\boldsymbol{r}_i$ converges into a vector that is orthogonal to both of $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$.

Schmidt process iteratively. Let $p(\cdot,\cdot)$ denote the function to compute the coefficient of orthogonal projection:

$$p(\boldsymbol{r},\boldsymbol{x}) = \frac{\langle \boldsymbol{r},\boldsymbol{x}\rangle}{\|\boldsymbol{x}\|^2}. \tag{B.2}$$

Then, the idea of this algorithm is illustrated in Fig. B.1 and can be described as follows.

- We initialize: $\boldsymbol{r}_i \leftarrow \boldsymbol{x}_i$, $a_{ij} \leftarrow 0$ for each $i, j \in \mathcal{I}$.
- We apply Gram-Schmidt process to make $\boldsymbol{r}_i$ orthogonal to $\boldsymbol{x}_j$ for each $j \in \mathcal{I} \setminus \{i\}$ alternately, such that $\boldsymbol{r}_i \leftarrow \boldsymbol{r}_i - p(\boldsymbol{r}_i, \boldsymbol{x}_j)\boldsymbol{x}_j$. We simultaneously update the coefficient $a_{ij}$ such that $a_{ij} \leftarrow a_{ij} + p(\boldsymbol{r}_i, \boldsymbol{x}_j)$.
- After repeating this sequential orthogonalizations many times, $\boldsymbol{r}_i$ converges into a vector which is approximately orthogonal to each $\{\boldsymbol{x}_j | j \in \mathcal{I} \setminus \{i\}$. In other words, $\boldsymbol{r}_i$ becomes approximately orthogonal to $\mathcal{U}_{(\mathcal{I}\setminus\{i\})}$. Then, $\{a_{ij} | j \in \mathcal{I} \setminus \{i\}\}$ is the approximate solution of Eq. (IV.5).

The detailed procedures are summarized in Algorithm B.1.

**Algorithm B.1**

**Input:** $\boldsymbol{x}_i | i \in \{1, \cdots, cn\}$ and iterations# $k_{max}$
**Output:** $\boldsymbol{r}_i | i \in \mathcal{I}$ and $a_{ij} | i, j \in \mathcal{I}$
**Initialize**: $\boldsymbol{r}_i \leftarrow \boldsymbol{x}_i$ and $a_{ij} \leftarrow 0$ for each $i, j \in \mathcal{I}$
**for** $k = 1, \cdots, k_{max}$ **do**
  **for** $i \in \mathcal{I}$ **do**
    **for** $j \in \mathcal{I} \setminus \{i\}$ **do**
      $a_{ij} \leftarrow a_{ij} + p(\boldsymbol{r}_i, \boldsymbol{x}_j)$
      $\boldsymbol{r}_i \leftarrow \boldsymbol{r}_i - p(\boldsymbol{r}_i, \boldsymbol{x}_j)\boldsymbol{x}_i$
    **end for**
  **end for**
**end for**

In this algorithm, we do not need any large tensors except for the $\boldsymbol{x}$-s, the $\boldsymbol{r}$-s, and the $a$-s. In CUDA, they can be stored in the shared memory and passed to each thread by reference. Therefore, each thread suffers little memory consumption, and $\boldsymbol{r}_i$ and $a_{ij}$ for $i, j \in \mathcal{I}$ can be computed parallely. The theoretical computational order of this algorithm is $O(n^3)$, where $n$ denotes the number of neurons. However, it substantially reduces to $O(n^2)$ by parallel computing.

## B.2 Formalization and proof

The idea behind Algorithm B.1 can be formalized into the following theorem.

**Theorem B.1** *Let $\boldsymbol{a}$ and $\boldsymbol{b}_1, \cdots \boldsymbol{b}_m \in \mathbb{R}^d$, $\boldsymbol{r}$ denote the vector that is orthogonal to all the $\boldsymbol{b}$-s, $o$ denote the function of Gram-Schmidt orthogonalization:*

$$o(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{a} - \frac{\langle \boldsymbol{a}, \boldsymbol{b} \rangle}{\|\boldsymbol{b}\|^2} \boldsymbol{b}. \tag{B.3}$$

*Furthermore, let $m^p$ denote the function that satisfies $m^{p+1}(\boldsymbol{a}) = m^1(m^p(\boldsymbol{a}))$ and $m^1(\boldsymbol{a}) = o(o(\cdots o(o(\boldsymbol{a}, \boldsymbol{b}_1), \boldsymbol{b}_2) \cdots)$. Then, we have*

$$\boldsymbol{r} = \lim_{p \to \infty} m^p(\boldsymbol{a}). \tag{B.4}$$

**Proof of Theorem B.1** *As $m^1$ is the function of sequential Gram-Schmidt orthogonalization, $m^1$ should have an attractive fixed point $\boldsymbol{c}$ such that*

$$\boldsymbol{c} = \lim_{p \to \infty} m^p(\boldsymbol{a}). \tag{B.5}$$

*By the definition of the attractive fixed point, we have*

$$\boldsymbol{c} = m^1(\boldsymbol{c}) = o(o(\cdots o(o(\boldsymbol{c}, \boldsymbol{b}_1), \boldsymbol{b}_2)\cdots), \boldsymbol{b}_m). \tag{B.6}$$

*Since $o$ conducts contractive mapping, the necessary and sufficient condition for (B.6) to hold is*

$$o(\boldsymbol{c}, \boldsymbol{b}_1) = o(\boldsymbol{c}, \boldsymbol{b}_2) = \cdots = o(\boldsymbol{c}, \boldsymbol{b}_m) = \boldsymbol{c} \tag{B.7}$$

*This means that $\boldsymbol{c}$ is orthogonal to all of $\boldsymbol{b}_1, \cdots, \boldsymbol{b}_m$. Therefore, we have $\boldsymbol{c} \perp U$, which means that $\boldsymbol{c}$ is the projection residual of $\boldsymbol{a}$ onto $U$. Therefore, we have*

$$\boldsymbol{r} = \boldsymbol{c} = \lim_{p\to\infty} m^p(\boldsymbol{a}) \tag{B.8}$$

## B.3 Case studies with VGG16

We evaluated Algorithm B.1 with VGG16 model. VGG16's *Conv1-1* layer has 64 channels and $3\times3$ kernel resolution. By using *im2col* function [53], it can be turned to an equivalent fully connected layer that has $64 \times 3 \times 3 = 576$ neurons. We fed images into the model and encode neuron behavior (the $\boldsymbol{x}$-s), and applied Gram-schmidt process-based algorithm to compute reconstruction error (the $\boldsymbol{r}$-s).

We picked up some neurons and show how the L2 norm of their residual (For better view, we normalized it with its original value, such that $\|\boldsymbol{r}_i\|/\|\boldsymbol{x}_i\|$.) changed over iterations in Fig. B.2. When the curve becomes flat, it means that values of the $\boldsymbol{r}$-s will not change a lot anymore, and that $\boldsymbol{r}_i$ has become almost orthogonal to $\boldsymbol{x}_j$ for each $j \in \mathcal{I} \setminus \{i\}$. In Figure B.2, we observe that the curves become almost flat at 10 to 20 iterations.

**Computational cost and scalability**

We have implemented our algorithm with CUDA 8.0, and run it on GeForce GTX 1080. When the number of neurons $n = 4608, 9216, 18432$ and the number of iterations $p = 100$, it took 115, 651, and 5117 seconds. This Gram-Schmidt process-based algorithm can obtain reconstruction error of neurons on such large layers within reasonable computational time.

# C Tips for implementation of REAP

In this section, we provide some tips for implementation that are crucial for the computational efficiency of REAP significantly.

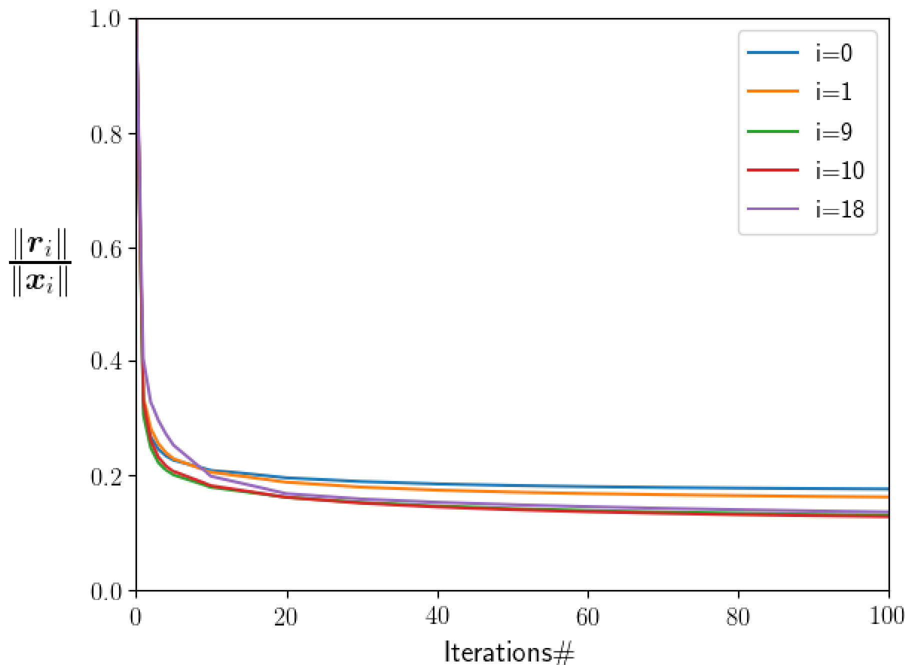


Figure B.2: This figure shows how the $\boldsymbol{r}$-s converge with the Gram-Schmidt process-based algorithm.

In order to solve Eq. (IV.7), we first compute the $v$-s by using the biorthogonal system based algorithm that we proposed, then we need to compute the following for each $i$:

$$f(\mathcal{I} \setminus \{i\}) = \left\| Y - \sum_{j \in \mathcal{I} \setminus \{i\}} \boldsymbol{x}_j \left( a_{ij}^* \boldsymbol{w}_i + \boldsymbol{w}_j \right)^\top \right\|_\mathrm{F}^2 . \tag{C.1}$$

This is also computationally expensive if implemented as is, because $Y$ and $\boldsymbol{x}_i \boldsymbol{w}_i^\top$ are typically large matrices. Because $Y = \sum_{i \in \mathcal{I}} \boldsymbol{x}_i \boldsymbol{w}_i^\top$ by definition, Eq. (C.1) can be simplified to

$$f(\mathcal{I} \setminus \{i\}) = \left\| \left( \boldsymbol{x}_i - \sum_{j \in \sum_{j \in \mathcal{I} \setminus \{i\}}} a_{ij}^* \boldsymbol{x}_j \right) \boldsymbol{w}_j^\top \right\|_\mathrm{F}^2 = \left\| \boldsymbol{r}_i \boldsymbol{w}_i^\top \right\|_\mathrm{F}^2 . \tag{C.2}$$

Note that $\boldsymbol{r}_i$ is the residual of $\boldsymbol{x}_i$ reconstructed from all other $\boldsymbol{x}$-s, as defined in Eq. (IV.8).

Let $tr(\cdot)$ denote trace. Because $tr(AA^\top) = \|A\|_\mathrm{F}^2$ holds true for an arbitrary

matrix $A$, Eq. (C.2) can be rewritten as

$$\begin{aligned} f(\mathcal{I} \setminus \{i\}) &= tr\left(\boldsymbol{r}_i \boldsymbol{w}_i^\top \left(\boldsymbol{r}_i \boldsymbol{w}_i^\top\right)^\top\right) \\ &= tr\left(\boldsymbol{r}_i \boldsymbol{w}_i^\top \boldsymbol{w}_i \boldsymbol{r}_i^\top\right). \end{aligned} \tag{C.3}$$

Because trace is invariant under cyclic permutation, $tr(AB^\top) = tr(B^\top A)$ holds true as far as $A$ and $B$ have the same dimensions. Therefore, we can further rewrite Eq. (C.3) as

$$\begin{aligned} f(\mathcal{I} \setminus \{i\}) &= tr\left(\boldsymbol{r}_i^\top \boldsymbol{r}_i \boldsymbol{w}_i^\top \boldsymbol{w}_i\right) \\ &= \|\boldsymbol{r}_i\|^2 \|\boldsymbol{w}_i\|^2. \end{aligned} \tag{C.4}$$

Computing Eq. (C.4) is more efficient, as we do not need to once compute the large matrix $\boldsymbol{r}_i \boldsymbol{w}_i^\top$ and put it on memory. Therefore, we can save the memory consumption for those large matrices by using Eq. (C.4).

**When we prune more than one neuron**

When we the second and more neurons, we need to compute $f(\mathcal{J})$, where $\mathcal{J}$ denotes the set of remaining neurons' indices. In this case, we first compute

$$B = R^\top R \odot W^\top W, \tag{C.5}$$

where $\odot$ denotes Hadamard product, $R = [\boldsymbol{r}_1 \cdots \boldsymbol{r}_n]$, $W = [\boldsymbol{w}_1 \cdots \boldsymbol{w}_n]$ and $n$ is the number of neurons. Let $b_{ij}$ denote the $(i, j)$ entry of $B$. Then, we have

$$f(\mathcal{J}) = \left\| \sum_{i \in \mathcal{I} \setminus \mathcal{J}} \boldsymbol{r}_i \boldsymbol{w}_i^\top \right\|_{\mathrm{F}}^2 = \sum_{i,j \in \mathcal{I} \setminus \mathcal{J}} b_{ij}. \tag{C.6}$$

Eq. (C.6) can be proven by substituting arbitrary vectors to the $\boldsymbol{r}$-s and the $\boldsymbol{w}$-s. Therefore, once we compute $B$, we can compute $f(\mathcal{J})$ for any $\mathcal{J}$ by only adding corresponding elements of $B$, which results in significant reduction of computational time.

# D How adequate is REAP's solution?

As we take a greedy approach in neuron selection of REAP, it naturally arises the question "How adequate is its solution?" we demonstrate it in an experiment with simulation data.

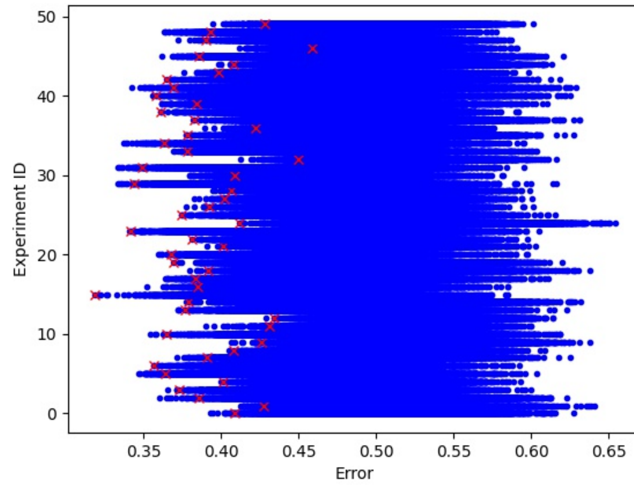


Figure D.1: The error caused by pruning. The blue dots are of exhaustive search, and the red crosses are of REAP. The vertical axis is experiment ID, and the horizontal axis is reconstruction error.

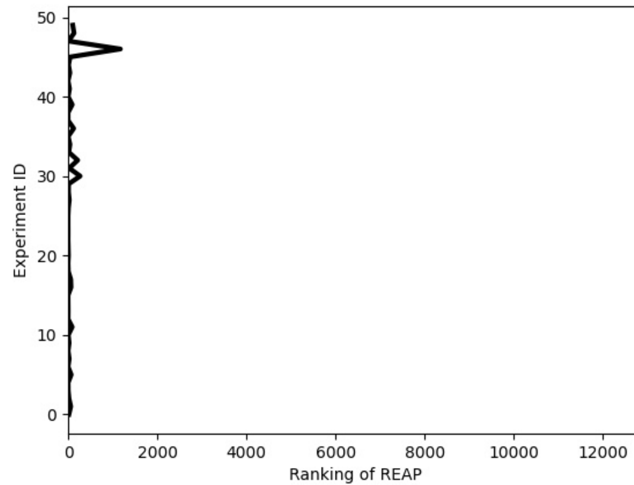


Figure D.2: The rank of REAP's solution out of 12,870 possible solutions. The vertical axis is experiment ID, and the horizontal axis is the rank.

In the experiments, we have 16 neurons, prune 8 of them, and perform reconstruction with the remaining 8. The behavioral vectors, the $\boldsymbol{x}$-s, are randomly (normal random) generated 128-dimension vectors. The next layer has only 1 neuron, and the weight going from each neuron to the next layer is set to 1, which is without loss of generality. On one hand, we used REAP's algorithm to select 8 neurons, and on the other hand, we conducted exhaustive search whose solutions always include REAP's one. In order to be statistically fair, we repeated experiments 50 times with different (randomly generated) data.

Fig. D.1 shows the reconstruction errors in each experiment. In the distribution of the exhaustive search solutions, REAP's solutions tend to be located to the left, which implies that they are fairly good solutions. Fig. D.2 shows the rank of REAP's solution out of 12,870 possible solutions in exhaustive search. REAP gave the global optimal solution 5 times out of 50 trials, and is ranked within the top 1% in most cases. Therefore, we can say that even though REAP does not always give the global optimal solution, it gives fairly good sub-optimal solution.

# List of Publications

- Koji Kamma, Yuki Isoda, Sarimu Inoue, and Toshikazu Wada. Neural Behavior-Based Approach for Neural Network Pruning. IEICE Transactions on Information and Systems, Vol. E103-D, No. 05, pp. 1135-1143, 2020.
- Koji Kamma and Toshikazu Wada. REAP: A Method for Pruning Convolutional Neural Networks with Performance Preservation. IEICE Transactions on Information and Systems, Vol. E104-D, No. 01, pp. 194-202, 2021.
- Koji Kamma, Yuki Isoda, Sarimu Inoue, and Toshikazu Wada. Behavior-Based Compression for Convolutional Neural Networks. Image Analysis and Recognition, Vol. 1, p. 427-439, 2019.
- Koji Kamma and Toshikazu Wada. Reconstruction Error Aware Pruning for Accelerating Neural Networks, Advances in Visual Computing, Vol. 1, p. 59-72, 2019.
- 和田俊和，菅間幸司，磯田雄基. ニューラルネットワーク処理装置、コンピュータプログラム、ニューラルネットワーク製造方法、ニューラルネットワークデータの製造方法、ニューラルネットワーク利用装置、及びニューラルネットワーク小規模化方法. 特願 2019-059091.
- 和田俊和，菅間幸司. ニューラルネットワークの圧縮方法、ニューラルネットワーク圧縮装置、コンピュータプログラム、及び圧縮されたニューラルネットワークデータの製造方法. 特願 2019-137019.
- Koji Kamma and Toshikazu Wada. Accelerating the Convolutional Neural Networks by Smart Channel Pruning. In Proceedings of the 22nd Meeting on Image Recognition and Understanding, No. 44, 2019.
- Koji Kamma and Toshikazu Wada. Serialized Residual Network. In Proceedings of the 23rd Meeting on Image Recognition and Understanding, No. 23, 2020.