Applications of Markov Random Field Optimization and 3D Neural Network Pruning in Computer Vision

Zhiwei Xu

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

January 2022

Declaration

I hereby declare that this thesis has been composed by myself based on my previous research outcomes including published and pre-printed papers and that this thesis has not been used for any other degrees internal and external the ANU. The works involved in this thesis are almost all my own in collaboration with the co-authors except where explicitly stated and appropriately cited otherwise. References for used datasets, comparative experiments, and code from other researchers' works are cited in this thesis.

Zhiwei Xu
8 January 2021

To My Family
To My Forever Grandpa
...

Acknowledgments

In retrospect over the past 3.5 years of my Ph.D. study, I have quite a lot of people to thank, especially it is my first time living in a different country.

The first and the most important person whom I would like to thank during my Ph.D. study is my supervisor, Prof. Richard Hartley. It was a long time since when I contacted Richard for the first time in 2014 to when my visa was granted in 2017. Many things happened in between, full of sorrows and happiness. It is an unforgettable memory long in my life and somehow changed me in many aspects. Without his patience and assistance, I was unable to study here.

In my Ph.D. study, I learned a lot from Richard, from how to write a formal mathematical sentence to how to think and write an appropriate paper. He is an excellent mentor not only because of his way of teaching students but also his deep knowledge in research and his way of doing such research. Among those I knew, I never met a person who at around his age is still learning CUDA from scratch, writing new knowledge from scratch, fluting, cycling, and riding scooter with just a helmet, and can walk so fast that I sometimes cannot catch up. He is a supervisor, but more than that, a humorous grandpa. By the approaching of Richard's retirement and the end of my Ph.D. study, I still have many things that can learn from him. I do wish he has a wonderful life after retirement.

I would like to thank Dr. Ajanthan for his patient guidance and knowledge in MRF and his introducing for the collaboration with Dr. Vibhav from Microsoft Research. Thank you both for giving me valuable suggestions and brainstorm during the regular meeting time regardless of different time zones.

I also appreciate my co-supervisors Prof. Hongdong and Prof. Stephen for research discussions. They are extremely amiable and responsible with great ideas and suggestions from their rich experience and knowledge. Many thanks to the group members, Amir, Arif, Kartik, Liyuan, Samira, Shihao, and Yao (in alphabetical order).

Lastly, the most important people in this world whom I really thank without any doubt are my parents, my young brother, my aunt, and my grandpa. The regular contacts with my parents greatly cheered me up when I felt desperate. Their great marriage always makes me believe in true love. No matter what happens, they always stand by my side. Meanwhile, I hope my brother can marry his beloved ASAP.

I am also grateful for the care from my aunt all the time since my brother and I were born. I felt sorry of being absent when my beloved grandpa left us last month. Hope everything works well when he is in heaven with my grandma.

All in all, I appreciate and love everyone involved in my Ph.D. study, no matter listed above or not, you are always in my heart. It is my great fortune to meet and know you among these 7.8 billion people in this world!

Abstract

Recent years witness the rapid development of Convolutional Neural Network (CNN) in various computer vision applications, such as stereo vision, image inpainting, panorama stitching, semantic segmentation, and so on. These application can be traditionally achieved by using Markov Random Field (MRF) optimization methods. Even though the state-of-the-art CNN based methods achieve high accuracy in these tasks, a high level of fine results is somehow difficult to be achieved with an analysable inference or reasoning. For instance, a pairwise MRF optimization method is capable of segmenting objects with the auxiliary edge information through the second-order terms, which is uncertain to be achieved by a deep neural network. MRF optimization methods, however, are able to enhance the performance with explicit theoretical and experimental supports using iterative energy minimization. Such auxiliary information can be obtained by edge detectors for Canny edges, superpixels, and so on.

Secondly, such an edge detector nowadays can be learned by CNNs, and thus, seeking for a suitable approach of transferring the task of a CNN for another task becomes valuable. In our case, it is desirable to fuse the superpixel contours from a state-of-the-art CNN with semantic segmentation results from another state-of-the-art CNN. Such a fusion enhances the object contours in semantic segmentation by aligning the segmentation edges with the superpixel contours. This fusion learning is not limited to semantic segmentation but also other tasks with a mutual effect of multiple off-the-shelf CNNs.

While fusing multiple state-of-the-art CNNs is useful to enhance the performance, each of such CNNs is usually specifically designed and trained with an empirical configuration of resources. For example, one sets the largest batch size or spatial size of inputs within the maximum GPU capacity. With such a large batch size, however, the joint CNN training is possible to be out of GPU memory when fusing with multiple streams of the CNNs. Such a problem is usually involved in efficient CNN training yet with limited resources. This issue is more obvious and severe in 3D CNNs than 2D CNNs due to the high requirement of training resources by network hidden layers. Hence, efficiently compressing these CNNs is a desirable solution.

In this thesis, we introduce our solutions to the aforementioned three problems. To solve *the first problem*, we propose two fast and differentiable message passing algorithms, namely Iterative Semi-Global Matching Revised (ISGMR) and Parallel Tree-Reweighted Message Passing (TRWP), for both energy minimization problems and deep learning applications, that is semantic segmentation with Canny edges in our case. Our experiments on Middlebury stereo vision dataset and image inpainting dataset validate the effectiveness and efficiency of our methods with minimum ener-

gies comparable to the state-of-the-art algorithm TRWS. We also greatly improve the forward and backward propagation speed using CUDA programming on massive parallel trees. Applying these two methods on deep learning semantic segmentation on PASCAL VOC 2012 with Canny edges achieves enhanced segmentation mean Intersection over Union (mIoU).

In *the second problem*, to effectively fuse and finetune multiple CNNs, we present a transparent initialization module. This module identically maps the output of a multiple-layer module (mainly fully-connected layers in our case) to its input at the early stage of finetuning. The pretrained model parameters are then gradually divergent in training as the loss decreases. This transparent initialization has a higher initialization rate, that is a high ratio of non-zero initial parameters, than Net2Net [Chen et al., 2016b] and a higher recovery rate, that is a high ratio of outputs with the same values of the inputs, compared with random initialization and Xavier initialization [Glorot and Bengio, 2010]. Our experiments on PASCAL VOC 2012, PASCAL Context, and ADE20K with superpixel contours validate the effectiveness of the proposed transparent initialization and the sparse encoder with sparse matrix operations. The edges of segmented objects achieve a higher performance ratio and a higher F-measure than the comparable methods.

In *the third problem*, we mainly compress 3D CNNs because 3D CNNs usually cost much more resources, mainly GPU memory and FLOPs, than 2D CNNs. Briefly, we propose a single-shot neuron pruning method with resource constraints. The pruning principle is to remove the neurons with low neuron importance corresponding to small connection sensitivities measured by the magnitude of the gradients of CNN parameter masks. The reweighting strategy with the layerwise consumption of memory or FLOPs improves the pruning ability by avoiding pruning the whole layer(s). Our experiments on point cloud dataset, ShapeNet, and medical image dataset, BraTS'18, using 3D-UNets for 3D semantic segmentation prove the effectiveness of our method. Applying our method to video classification on UCF101 dataset using MobileNetV2 and I3D further strengthens the benefits of our method. We also validate its effectiveness on a two-view stereo matching network with a mixture of 2D CNNs and 3D CNNs on Sceneflow dataset.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computer vision is a multidisciplinary research area that promotes computers to extract and analyse information from images or videos as human visual behaviors. The beginning of computer vision can be dated back to the late 1960s when researchers believed that images captured by a camera can be automatically processed by a computer for "what it sees".

As the number of mathematical theory and quantitative analysis of computer vision increases in various areas since then, research in computer vision develops from the low-level 2D digital image understanding, such as edge detection, texture recognition, image interpolation, to 3D scene analysis and reconstruction, such as camera calibration, depth estimation, optical flow, motion segmentation, and so on. Differently, recent developments of computer vision highly depend on deep learning [LeCun et al., 2015], especially convolutional neural networks with multiple learnable kernels. The early attention starts from the contest of Large Scale Visual Recognition Challenge 2010 (LSVRC2010) [Russakovsky et al., 2014].

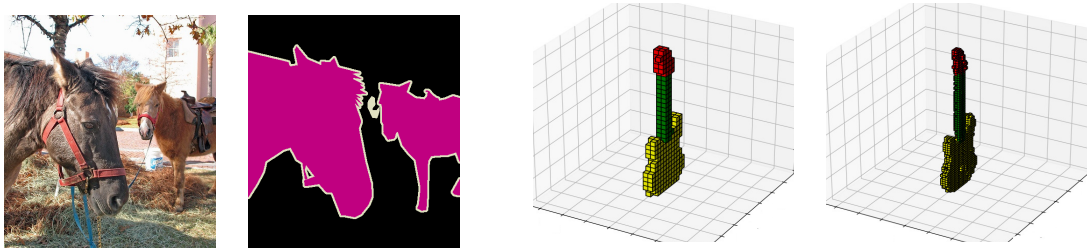
Many of the concepts of these problems can be formulated in optimization frameworks as Markov Random Fields (MRFs), an undirected graphical model. While optimizing on such an MRF is usually treated from the perspective of probabilistic graphical model [Koller and Friedman, 2009; Jordan, 1998], many cast it into an energy minimization problem using the clique factorization of a graph. Forming an energy function for such an energy minimization problem needs to consider the properties of MRF. For instance, on a pairwise MRF, any two non-adjacent variables are conditionally independent; the state of a variable only depends on the states of its adjacent neighbours. This is the same as the property of a Markov chain.

Nevertheless, although many superior MRF algorithms in the aforementioned areas were well exploited in the past, their benefits are not highly used in current deep learning based methods. Regardless of the limited effects of some inferior MRF algorithms, such as the mean-field method in Zheng et al. [2015] and the semi-global matching method in Seki and Pollefeys [2017], this thesis will introduce improved MRF algorithms with backpropagation for deep learning. Meanwhile, inspired by the effects of pairwise priors in MRF algorithms and the fact that many existing deep learning models are off-the-shelf, the idea of composing the advantage of a model for the pairwise priors on another model for the unary priors would be valuable for fusion learning. However, the addition of extra module(s) or another model will

cause the increase of resource consumption mainly at the training phase in deep learning, leading to the issue of out of GPU memory and slow training. An efficient downscale of such models is necessary, for which we propose a single-shot neuron pruning method in this thesis. Details of the motivations are also specifically stated in each chapter.

1.1 Related Tasks

In the contents below, we will briefly introduce some computer vision tasks involved in our works.



(a) RGB Image (b) Segmentation Result (c) 32*32*32 (d) 64*64*64

Figure 1.1: *2D and 3D semantic segmentation. (a)-(b) are 2D semantic segmentation on PASCAL VOC 2012 dataset. (c)-(d) are 3D semantic segmentation on ShapeNet, a sparse point cloud dataset. The guitar model has* 32^3 *and* 64^3 *voxels represented from sparse point clouds. Different colors mean different semantics of the components.*

Semantic Segmentation. Semantic segmentation is a process of assigning one of the predefined labels, also known as semantics, to every pixel in image data. The involved data could be 2D images, 3D point clouds, 3D volumes, and so on. These semantics usually refer to object categories, such as person, traffic light, horse, cow, sea, mountain, and object components, including head, hand, body, eyes of a person, wings and cabin of a plane, and so on. Examples are shown in Fig. 1.1.

Since an object in an image is generally not projected to a single pixel but an area, effective semantic segmentation methods usually rely on local connections of pixels. Thus, traditionally, MRFs or Conditional Random Fields (CRFs) are used to assign the labels based on the relation between every two associate pixels or higher-order cliques. In contrast, this explicit local information could be achieved by learning a sufficiently deep CNN in an implicit manner. Such CNN based methods, however, are usually uncertain about the depth of the networks. Hence, recent state-of-the-art CNNs tend to be very deep and with multiscale inputs for a better adaption from trainsets to testsets.

Stereo Vision. Two-view stereo vision aims at estimating a depth or disparity map using two images captured by two horizontally displaced cameras, see the physical model in Fig. 1.2. Rectification of these two images is required to remove vertical interruptions such that in a scanline along a row of these two images, a disparity can be easily found by a horizontal shift except that there is an occlusion. The cause of

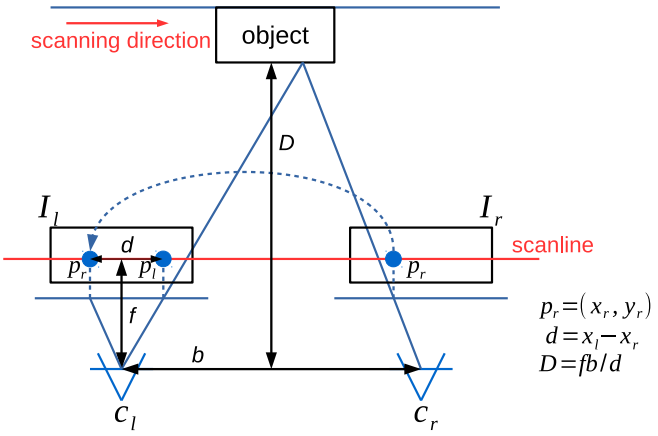
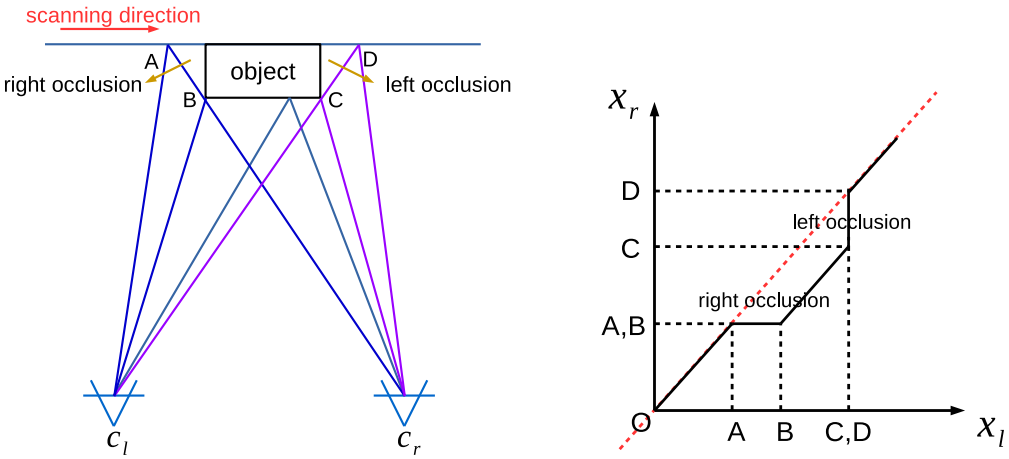


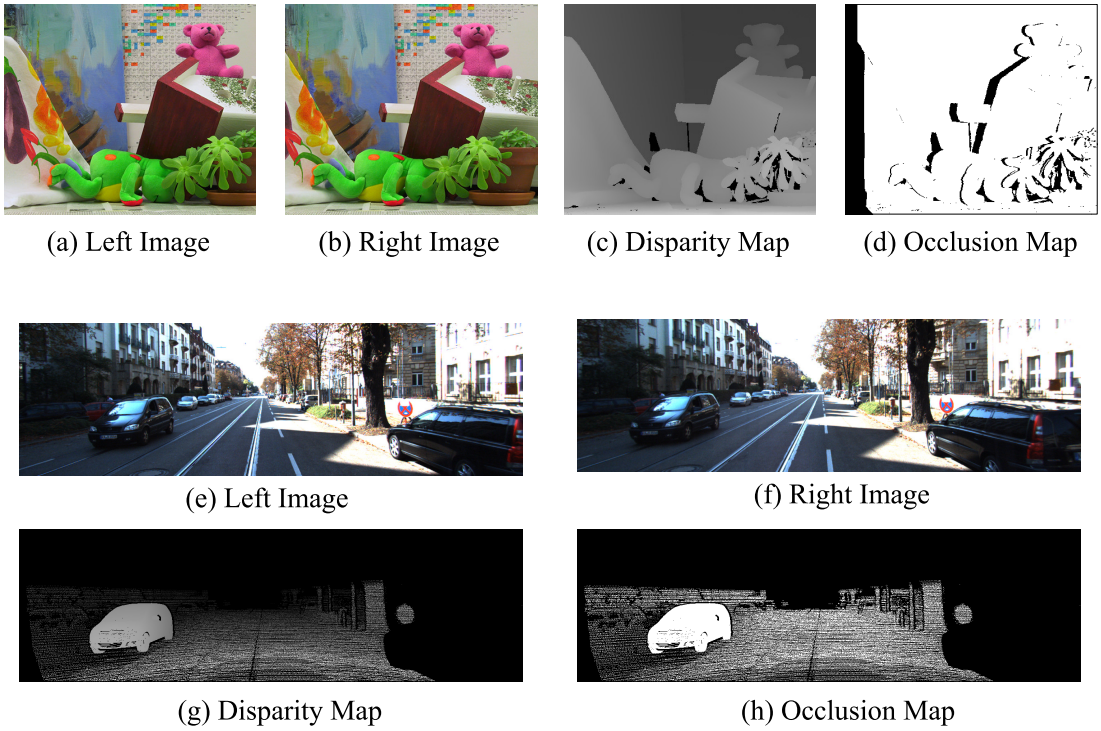
Figure 1.2: *Physical model of two-view stereo vision. I_l and I_r are the left and right images captured by the left camera C_l and the right camera C_r respectively. p_l is a pixel with x-y coordinates, x_l and y_l, of the left image I_l and p_r of I_r. d is a disparity. D is a depth calculated by focal length f, baseline b, and disparity d. Pixel p_r can be traced by a horizontal shift from x_r to x_l by d pixels.*



(a) An Example of Occlusion Setting (b) Occluded Disparity

Figure 1.3: *An occlusion example in two-view stereo vision. (a) is a view scanning from location A to location D. The scene from A to B is occluded in the right view from C_r while the scene from C to D is occluded in the left view from C_l. The disparity values are illustrated in (b) where in X_r axis, A and B have the same disparity and in X_l axis, C and D have the same disparity while in these two areas, the real scenes cannot be seen and projected in the related images.*

the occlusion is analysed in Fig. 1.3. Eventually, in these two images, a non-occluded pixel of one image can be matched with the corresponding one in the other image by horizontally shifting x pixels where x is the disparity value of this non-occluded pixel. This vividly mimics the human binocular vision where a 3D distance sense



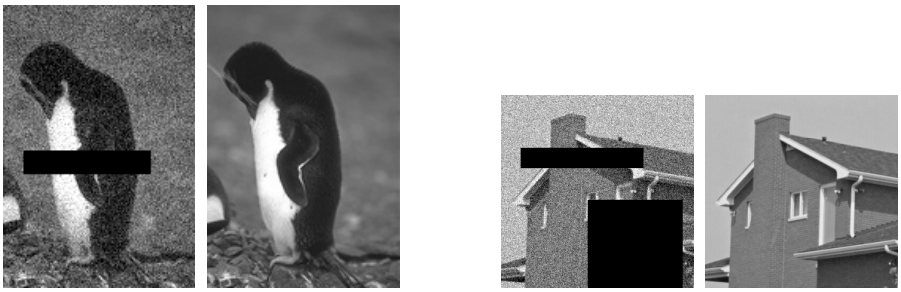
(a) Left Image (b) Right Image (c) Disparity Map (d) Occlusion Map

(e) Left Image (f) Right Image

(g) Disparity Map (h) Occlusion Map

Figure 1.4: *Two-view stereo vision of indoor / outdoor scenes. (a)-(d) are an indoor scene from Middlebury dataset. (e)-(h) are an outdoor scene from KITTI 2015 dataset. (c) is a complete disparity map while (g) is calculated with LIDAR data, and (d) and (h) are binary maps. The black areas in (d) and (h) are occlusion areas in the left images, which cannot be seen from the view of the right camera.*

is aware by such two views. An indoor and an outdoor stereo vision example are shown in Fig. 1.4.

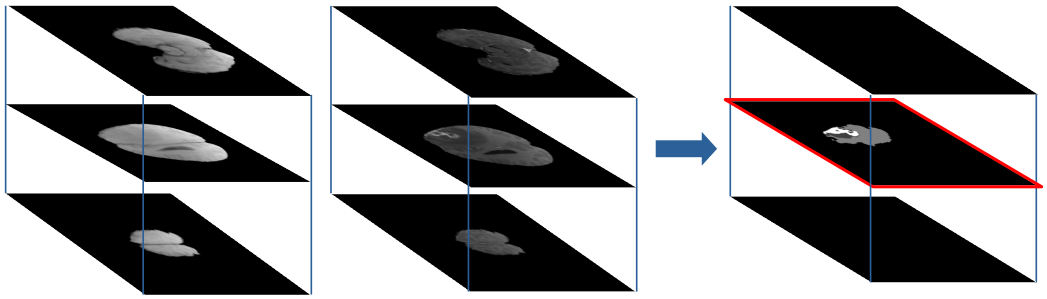


(a) Noisy Image (b) Ground Truth (c) Noisy Image (d) Ground Truth

Figure 1.5: *Image denoising and inpainting from Middlebury dataset.*

Image Denoising and Inpainting. Image denoising and inpainting is to remove noises, caused by such as high-frequency random pixel values, and then fill in realistic pixel values to areas that are damaged, missing, and distorted. A vivid application

is the conservation of ancient photos. This problem can be represented as an MRF labeling task of seeking the relation between a target pixel and its neighbouring pixels by decreasing a formulated energy function. For a grey image, the label is usually in [0, ..., 255]; for a colorful image, the label in each of the RGB channels is in [0, ..., 255]. For instance, in Fig. 1.5, the cropped areas in (a) and (c) need to be filled in and the noises need to be removed to get a complete and high-quality image as (b) and (d) respectively.



(a) T1 Images (b) T1 Contrast Enhanced Images (c) Tumor Segmentations

Figure 1.6: *Medical image semantic segmentation on BraTS 2018, an MRI brain tumor dataset. The dataset has T1, T1CE, T2, and FLAIR images as source data while T1 and T1CE are illustrated in (a) and (b) respectively. Some segmentation slices in (c), marked with a red bounding box, clearly distinguishes different parts of the tumor. Since each type of those images consists of hundreds of scanning MRI slices, we give 3 slices for each as an example.*

Medical Image Segmentation. Medical image segmentation is to identify the shape and location of organs from such as Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) images at the pixel level. These images are usually from a stack of 3D scanning slices. Therefore, the relation between every two neighbouring slices could be informative to trace the organ, such as liver, brain, tissues, of the staging cancer through its 3D structure. Here, we give an example from BraTS 2018 dataset for brain tumor identification and segmentation. In Fig. 1.6, a T1 image sequence and a T1 Contrast-Enhanced (T1CE) image sequence is provided in (a) and (b) respectively, while the complete list of the MRI images are T1, T1CE, T2, and FLuid-Attenuated Inversion Recovery (FLAIR). In (c), the brain tumor is segmented with 3 semantics, Enhancing Tumor (ET), Tumor Core (TC), and Whole Tumor (WT), in some slices that are corresponding to their locations in reality.

1.2 Problem Setup

The involved problems in this thesis are a combination of MRF optimization and CNNs. It starts from 1) the limitations of existing MRF optimization algorithms in deep learning to 2) a fusion learning of pairwise constraints from a superpixel CNN and a semantic segmentation CNN, then to 3) an effective compression of CNNs to achieve largely reduced training resources.

1) Existing MRF inference algorithms in deep learning semantic segmentation and stereo vision are either less effective or less efficient in terms of accuracy and computational complexity respectively. For instance, CRFasRNN [Zheng et al., 2015] used the mean-field method as an MRF inference in deep learning semantic segmentation. SGMNet [Badrinarayanan et al., 2017] used SGM as an MRF inference in deep learning stereo matching. Neither the mean-field method nor the SGM method is superior to some state-of-the-art MRF optimization algorithms, such as TRWS. Although TRWS has a high ability to reduce an energy function, it is inefficient due to the sequential message passing manner across the whole image. Meanwhile, SGM is ineffective due to the single iteration of energy minimization. Therefore, given a more effective and efficient MRF optimization method for deep learning, it is promising to improve the task accuracy from the optimization perspective.

2) From the perspective of pairwise MRF optimization, the pairwise constraints are significant to adapt the adjacent pixel information in the final results, such as aligning object edges in semantic segmentation or stereo vision with binary edge constraints. Since such pairwise constraints can be learned by a CNN, it would be valuable to combine multiple CNNs to improve the accuracy of a specific task. Therefore, as a finetune strategy, seeking a suitable method to jointly learn these CNNs, usually pretrained, is important.

3) Combining multiple complicated CNNs would largely increase the resource requirements since each of such CNNs is specifically designed and usually trained with a large batch size within the maximum capacity of GPUs. The issue of limited resources at the training phase is more obvious and severe in 3D CNNs than 2D CNNs. Hence, remaining important parameters in a CNN for efficient and lightweight training without losing the accuracy is beneficial to reduce resource consumption. This should be generalized to both 2D and 3D CNNs.

1.3 Contributions

In this thesis, we proposed two MRF optimization methods that can be used for traditional MRF optimization and deep learning applications, a fusion network module for semantic segmentation, and an effective network pruning method for 3D CNNs. The main contributions of these works, corresponding to Chapters 3-5, can be categorised into threefolds below:

- We propose two effective and efficient MRF optimization methods, Iterative Semi-Global Matching Revised (ISGMR) and Parallel Tree-Reweighted Message Passing (TRWP), via a message passing manner. Both are fast and differentiable. This benefits traditional energy minimization problems, such as stereo vision and image denoising, as well as deep learning tasks, such as semantic segmentation in our case. Procedures of the forward and backpropa-

gation of the methods are provided and analysed in detail. Experiments on multiple tasks sufficiently validate the effectiveness of our methods, compared with other classical MRF optimization methods. We also accelerate the running speed by using CUDA programming, compared with CPU and PyTorch versions.

- We design a transparent initialization module and a sparse encoder to fuse and finetune multiple pretrained state-of-art neural networks. It improves the semantic segmentation performance with superpixel contours. Specifically, this transparent initialization module can almost 100% recover the input data of the module from its output data and has a high ratio of non-zero initial parameters. It greatly outperforms random initialization, Xavier initialization [Glorot and Bengio, 2010], and Net2Net [Chen et al., 2016b]. This module can also be used for other tasks such as knowledge distillation.

- We present a single-shot resource aware neuron pruning method for 3D CNNs, which is inspired by Lee et al. [2019]. The massively pruned neurons largely reduce not only the dimension of parameters but also the size of the hidden layers which usually take much more GPU memory than the parameters, especially for 3D CNNs. In this method, we propose a reweighting scheme based on the resource, that is memory or FLOPs, consumption of each layer. This greatly alleviates the problem of imbalanced neuron importance among layers, and thus, further improves the pruning capability without losing the accuracy of the tasks. We validate our method on 3D semantic segmentation, video classification, and two-view stereo matching with popular 3D network architectures.

1.4 Thesis Outline

Chapter 1: This chapter lists the contributions of our works and the structure of contents enumerated into individual chapters.

Chapter 2: This chapter introduces the background of Markov Random Fields (MRFs). We illustrate some classical optimization methods (including graph cuts and message passing algorithms, minimum cost (lifted) multicuts) and convolutional neural network pruning. The introduction of the background is associated with the following chapters and of values and interests for a complete understanding of the relevant fields and theoretical supports for future works.

Chapter 3: This chapter introduces two effective and efficient message passing algorithms, Iterative Semi-Global Matching Revised (ISGMR) and Parallel Tree-Reweighted Message Passing (TRWP). They can be applied to both MRF optimization problems and deep learning tasks [Xu et al., 2020a]. We analyse the theoretical evidence that the proposed methods are either more effective or efficient or both than the baseline optimization methods. We also provide the experimental comparison of the optimal energies and execution speed using CUDA implementation. Detailed derivations of

the forward propagation and backward propagation of the related message passing methods are provided in this chapter.

Furthermore, experiments of deep learning semantic segmentation on PASCAL VOC 2012 illustrate that our methods can largely preserve object edges and enhance the segmentation accuracy.

Chapter 4: This chapter introduces a transparent initialization module and a sparse encoder using sparse matrix operations. It aims at refining semantic segmentation results with superpixels for enhanced object contours [Xu et al., 2020b]. The transparent initialization serves as a fusion of multiple state-of-the-art neural networks and finetunes pretrained models by gradually interrupting the parameters of the transparent layers which are initialized by identity mapping. For efficient logit averaging with superpixels based on each pixel, a sparse encoder is used to index the pixels with superpixel indices. This reduces the required memory and computational complexity, especially when the number of superpixels is large.

Experiments with the state-of-art networks for semantic segmentation validate the effectiveness of our method using mean Intersection over Union (mIoU) and edge metrics, that is Performance Ratio (PR) and F-Measure (FM). We also provide a booklet of transparent initialization with a detailed analysis of its structure and its development from simple layers without activation to those with activation.

Chapter 5: This chapter introduces a useful 3D neural network pruning method with resource constraints in a single-shot pruning manner, namely Resource-Aware Neuron Pruning (RANP) [Xu et al., 2020c]. The pruning objects are neurons of filters in each neural network layer. This largely reduces the GPU memory required by network parameters and hidden layers as well as computational complexity induced by FLoating Point Operations per second (FLOPs) for 3D CNNs.

Our experiments on 3D semantic segmentation using sparse point cloud dataset, ShapeNet, and dense medical dataset, BraTS'18, video classification using video dataset UCF101 with MobileNetV2 and the recent state-of-the-art I3D, and two-view stereo matching using dataset SceneFlow with PSM network validate the effectiveness of our RANP. We also provide a comparison of RANP with different initialization methods to demonstrate their effects on the network pruning balance among layers. Specifically, RANP outperforms the orthogonal initialization method that is useful for keeping a balance of parameter pruning for 2D CNNs but not 3D CNNs.

Chapter 6: This chapter concludes our works and presents potential future works as an extension of these works.

1.5 Publications

- RANP: resource aware neuron pruning at initialization for 3D CNNs
  Zhiwei Xu, Thalaiyasingam Ajanthan, Vibhav Vineet, and Richard Hartley
  (under review in *IJCV special issue on 3DV* as an invited submission, 2021)

- Refining semantic segmentation with superpixel by transparent initialization and sparse encoder
  Zhiwei Xu, Thalaiyasingam Ajanthan, and Richard Hartley
  (*arXiv:2010.04363*, 2020)

- RANP: resource aware neuron pruning at initialization for 3D CNNs
  Zhiwei Xu, Thalaiyasingam Ajanthan, Vibhav Vineet, and Richard Hartley
  International Conference on 3D Vision (3DV), pp. 180-189, Virtual Fukuoka, Japan, 2020 (oral and best student paper)

- Fast and differentiable message passing on pairwise Markov random fields
  Zhiwei Xu, Thalaiyasingam Ajanthan, and Richard Hartley
  Asian Conference on Computer Vision (ACCV), pp. 523-540, Virtual Kyoto, Japan, 2020 (oral)

Chapter 2

Background

In this chapter, we introduce the background of MRF optimization with some classical problems and algorithms (that is graph cuts, minimum cost (lifted) multicuts, and message passing algorithms), parameter pruning, and neuron pruning for CNNs.

2.1 MRF Optimization

A Markov Random Field (MRF) optimization problem is usually formulated as an energy minimization form as

$$E(\mathbf{x}) = \sum_{c \in \mathcal{C}} E_c(\mathbf{x}_c) \ , \tag{2.1}$$

where $\mathbf{x}$ is a labeling to a set of random variables, usually nodes of a graph, $\mathcal{C}$ is a set of cliques with different clique sizes. Generally, when the clique size is 1, it represents a graph node; when the clique size is 2, it represents an edge connecting two graph nodes; and for the clique size larger than 2, they are higher-order cliques. The label set for which $\mathbf{x}$ to be assigned can be binary or multilabel, which promotes the energy function to be a binary or multilabel cost function, leading to $x \in \mathcal{B} = \{0, 1\}$ and $x \in \mathcal{L} = \{0, ..., L-1\}$ where the number of labels $L \geq 3$ respectively.

The optimal solution to Eq. (2.1) is

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} E(\mathbf{x}) \ . \tag{2.2}$$

The approaches of minimizing such an energy function include move-making algorithms (graph cuts based) [Boykov et al., 1998, 2001a] and message passing algorithms [Kolmogorov, 2006; Hirschmuller, 2008; Jordan, 1998].

2.1.1 Graph Cuts

In computer vision, the graph cuts optimization problem employs a max-flow/min-cut theory to cut a graph into several distinct components. This is usually formulated by an energy minimization for a feasible labeling such that the cost of cutting the graph (such as unary terms, pairwise terms, higher-order terms) is the minimum.

Graph cuts can solve binary labeling problems with an exact solution. For a multilabel (more than two labels) labeling problem, however, an approximate solution is usually available by decomposing the multilabel labeling problem into several pseudo-boolean labeling problems. It can be expressed as

$$\mathbf{u}^* = \underset{\mathbf{u}\in\mathcal{B}}{\operatorname{argmin}} E(\phi(\mathbf{u})) \quad \text{and} \quad \mathbf{x}^* = \phi(\mathbf{u}^*) \ . \tag{2.3}$$

For such a pseudo-boolean labeling problem, the meaning of the binary label set depends on the involved approach. For instance, the label 0 in alpha expansion means "retain the original label" and 1 means "assigned label α to the target node".

In the sentences below, we will introduce two typical move-making algorithms, that is alpha expansion and alpha-beta swap. Since these algorithms require the energy functions to be submodular, we first introduce the concept of submodularity.

2.1.1.1 Submodular Functions

Definition 1 *Given a set $\mathcal{V}$, a set function f over $\mathcal{V}$ is submodular if*

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B) \ , \ \forall A, B \in \mathcal{V} \ . \tag{2.4}$$

This can be represented as

$$f(B) - f(A \cap B) \geq f(A \cup B) - f(A) \ , \ \forall A, B \in \mathcal{V} \ , \tag{2.5}$$

which indicates a property that the difference of an incremental value of a set function f decreases when an element is added to a larger set of the input variables. In other words, when the set $S = B \setminus (A \cap B) = (A \cup B) \setminus A$ is added to $f(A)$, which has a larger set of self-variables than $f(A \cap B)$, the difference in the increase of f is less than that of adding it to $f(A \cap B)$. Eventually, as the set of self-variables increases, the value of f tends to be a constant such that it is convergent.

Now, we define this set function f using subsets of $\mathcal{V}$ by a pseudo-boolean function. Given a set of indices $\mathcal{V} = \{1, ..., n\}$ and a set of elements in $\mathcal{B}^{\mathcal{V}}$ denoted by $\mathbf{x} = \{x_1, ..., x_n\}$, an element-wise correspondence between vector $\mathbf{x} \in \mathcal{B}^{\mathcal{V}}$ and set $A(\mathbf{x}) \in 2^{\mathcal{V}}$ is given by

$$A(\mathbf{x}) = \{i \in \mathcal{V} | x_i = 1\} \ . \tag{2.6}$$

Given two elements $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{B}^{\mathcal{V}}$ corresponding to subsets $A(\mathbf{x})$ and $A(\mathbf{y})$ in $2^{\mathcal{V}}$, we define

$$\begin{aligned} A(\mathbf{x}) \cap A(\mathbf{y}) &\equiv \mathbf{x} \wedge \mathbf{y} \ , \\ A(\mathbf{x}) \cup A(\mathbf{y}) &\equiv \mathbf{x} \vee \mathbf{y} \ , \end{aligned} \tag{2.7}$$

where $\wedge$ and $\vee$ are min$(\cdot)$ and max$(\cdot)$ operations over two elements given that x_i has an ordering value where $0 < 1$, as

$$\begin{aligned} \mathbf{x} \wedge \mathbf{y} &= \{x_i \wedge y_i | i \in \mathcal{V}\} = \{\min(x_i, y_i) | i \in \mathcal{V}\} \ , \\ \mathbf{x} \vee \mathbf{y} &= \{x_i \vee y_i | i \in \mathcal{V}\} = \{\max(x_i, y_i) | i \in \mathcal{V}\} \ . \end{aligned} \tag{2.8}$$

Then, we have

Definition 2 *A pseudo-boolean set function* $f : \mathcal{B}^{\mathcal{V}} \rightarrow \mathbb{R}$ *is submodular if*

$$f(\mathbf{x}) + f(\mathbf{y}) \geq f(\mathbf{x} \vee \mathbf{y}) + f(\mathbf{x} \wedge \mathbf{y}) \ , \quad \forall \mathbf{x}, \mathbf{y} \in \mathcal{B}^{\mathcal{V}} \ . \tag{2.9}$$

Clearly, for a set of two elements in $\mathcal{V}$, the submodular set function $f : \mathcal{B}^2 \rightarrow \mathbb{R}$ follows

$$f(1,0) + f(0,1) \geq f(1,1) + f(0,0) \ , \tag{2.10}$$

where one can regard $\mathbf{x} = \{1,0\}$ and $\mathbf{y} = \{0,1\}$.

2.1.1.2 Alpha Expansion

For a L-element label set $\mathcal{L}$ ($L \geq 3$), the alpha expansion method decomposes the multilabel labeling problem into L pseudo-boolean labeling problems. In each of these pseudo-boolean labeling problems, the label of a target node is changed to a predefined label α when it satisfies a given condition; otherwise, it remains the current label. This can be written for the t-th iteration as

$$\phi_i^t(\mu_i) = \begin{cases} x_i^t & \text{if } \mu_i = 0 \\ \alpha & \text{if } \mu_i = 1 \end{cases} \ . \tag{2.11}$$

This can be repeated by updating $\mathbf{x}^{t+1} = \phi^t(\mathbf{u}^*)$ till the energy converges such that

$$E(\mathbf{x}^{t+1}) = E(\phi^t(\mathbf{u}^*)) \leq E(\phi^t(\mathbf{u})) = E(\mathbf{x}^t) \ . \tag{2.12}$$

An important condition for the alpha expansion algorithm is that the energy function should be a submodular function. Given a multi-label energy function as

$$E(\mathbf{x}) = \sum_{i \in \mathcal{V}} E_i(x_i) + \sum_{(i,j) \in \mathcal{E}} E_{ij}(x_i, x_j) \ , \tag{2.13}$$

the corresponding pseudo-boolean energy function is

$$E(\phi(\mathbf{u})) = \sum_{i \in \mathcal{V}} E_i(\phi_i(u_i)) + \sum_{(i,j) \in \mathcal{E}} E_{ij}(\phi_i(\mu_i), \phi_j(\mu_j)) \ . \tag{2.14}$$

According to the condition for submodularity in Eq. (2.10), we have

$$E_{ij}(\phi_i(1), \phi_j(1)) + E_{ij}(\phi_i(0), \phi_j(0)) \leq E_{ij}(\phi_i(1), \phi_j(0)) + E_{ij}(\phi_i(0), \phi_j(1)) \ , \tag{2.15}$$

which is equivalent to

$$E_{ij}(\alpha, \alpha) + E_{ij}(x_i, x_j) \leq E_{ij}(\alpha, x_j) + E_{ij}(x_i, \alpha) \ . \tag{2.16}$$

This should hold for all α, x_i, and x_j in $\mathcal{L}$.

2.1.1.3 Alpha-Beta Swap

Another classical move-making MRF optimization method is alpha-beta swap algorithm which operates the binary labeling optimization (a subproblem of the original multilabel problem) over certain nodes that have labels in selected α and β from a label set $\mathcal{L}$. It gives the following labeling strategy.

$$\phi_i^t(\mu_i) = \begin{cases} \alpha & \text{if } x_i^t \in \{\alpha, \beta\} \text{ and } \mu_i = 0 \\ \beta & \text{if } x_i^t \in \{\alpha, \beta\} \text{ and } \mu_i = 1 \\ x_i^t & \text{otherwise} \end{cases} . \tag{2.17}$$

With the same energy function in Eq. (2.13), we analyse the cases of two elements x_i and x_j in $\mathcal{L}$ for the submodularity condition in Eq. (2.10).

1. if $x_i^t \notin \{\alpha, \beta\}$ and $x_j^t \notin \{\alpha, \beta\}$, then

$$E(\phi_i^t(\mu_i),\ \phi_j^t(\mu_j)) = E(x_i^t, x_j^t) , \tag{2.18}$$

   which is free from α and β, and thus, has no effects on the pseudo-boolean optimization in the current iteration.

2. if $x_i^t \in \{\alpha, \beta\}$ and $x_j^t \notin \{\alpha, \beta\}$, then

$$E(\phi_i^t(\mu_i),\ \phi_j^t(\mu_j)) = \bar{\mu}_i E(\alpha, x_j^t) + \mu_i E(\beta, x_j^t) , \tag{2.19}$$

   which forms a linear term over μ_i for the energy value.

3. if $x_i^t \in \{\alpha, \beta\}$ and $x_j^t \in \{\alpha, \beta\}$, it has four cases of (μ_i, μ_j) since $\mu_i, \mu_j \in \{0, 1\}$. According to Eq. (2.15), we have

$$E(\beta, \beta) + E(\alpha, \alpha) \leq E(\beta, \alpha) + E(\alpha, \beta) , \tag{2.20}$$

   for a given pairwise function where $E(x, x) = 0$, the above becomes

$$0 \leq E(\beta, \alpha) + E(\alpha, \beta) . \tag{2.21}$$

   This can be easily achieved for any non-negative pairwise function $E(x_i, x_j)$.

2.1.2 Minimum Cost (Lifted) Multicuts

Different from graph cuts which has s-t nodes for max-flow, **the minimum cost multicut problem** [Grotschel and Wakabayashi, 1989; Chopra and Rao, 1993; Beier et al., 2016] is a binary-labeling optimization problem using graph decompositions as feasible solutions. Distinct components of the decomposed graph (subgraph) are connected by edges, which are called multicuts. In other words, such edges that straddle distinct components are identified as multicuts whose costs, in general, are supposed to be the minimum for the optimal solution.

Given a graph $G = (V, E)$ where V is a set of graph nodes and E is a set of graph edges, a feasible graph decomposition is a partition P of V such that for every subset S of P, all nodes in S are connected representing a component with similar attributes while those in separate Ss represent different attributes. The multicuts of graph G are defined by such a set of edges straddling distinct components that are decomposed from graph G.

With every edge in E associated with a cost $c : E \to \mathbb{R}$ and a binary set $\mathcal{B} = \{0, 1\}$, the minimum cost multicut problem is to find the minimum cost of such edges that straddle distinct components. This can be written as

$$\min_{x \in \mathcal{B}^E} \sum_{e \in E} c_e x_e \ , \tag{2.22a}$$

$$\text{subject to} \quad \forall Y \in \text{cycles}(G) \quad \forall e \in Y : \ x_e \leq \sum_{e' \in Y \setminus \{e\}} x_{e'} \ , \tag{2.22b}$$

where edge e is a multicut when $x_e = 1$ otherwise $x_e = 0$. To be more concrete, the constraint in Eq. (2.22) requires that a subset $M \subset E$ of edges is a multicut of graph G if and only if there is no cycle Y of G intersects with M precisely once. That is for every such Y, it has $|M \cap Y| \neq 1$.

To this end, we introduced the core setup of the minimum cost multicut problem. Now, we introduce **the minimum cost lifted multicut problem** [Andres, 2015].

Note that the minimum cost multicut problem has a limitation that every edge is associated with two neighbouring nodes, and thus, a multicut makes explicit only on two such neighbouring nodes with each in a distinct component. Therefore, a cost or reward is only assigned to pairs of neighbouring nodes. In this case, any connections between every two non-neighbouring nodes are not considered. A lifted multicut aims at solving such a problem by cutting a pair of non-neighbouring nodes (each of which is in a distinct component).

In the heritage of the aforementioned notations, the minimum cost lifted multicut problem introduces an extended set of edges, denoted as E', and thus, the graph is extended from $G = (V, E)$ to $G' = (V, E')$ and edge cost to $c : E' \to \mathbb{R}$. A lifted multicut is an edge $vw \in E' \setminus E$, where $v, w \in E$. The minimum cost lifted multicut problem is then defined as

$$\min_{x \in \mathcal{B}^{E'}} \sum_{e \in E'} c_e x_e \ , \tag{2.23a}$$

$$\text{subject to} \quad \forall Y \in \text{cycles}(G) \quad \forall e \in Y : \ x_e \leq \sum_{e' \in Y \setminus \{e\}} x_{e'} \ , \tag{2.23b}$$

$$\forall vw \in E' \setminus E \quad \forall P \in vw\text{-paths}(G) : \ x_{vw} \leq \sum_{e \in P} x_e \ , \tag{2.23c}$$

$$\forall vw \in E' \setminus E \quad \forall C \in vw\text{-cuts}(G) : \ 1 - x_{vw} \leq \sum_{e \in C} (1 - x_e) \ . \tag{2.23d}$$

The first constraint in Eq. (2.23) is identical to the one in Eq. (2.22) for $|M \cap E| \neq 1$. The other two constraints is to guarantee that for a non-cut lifted edge $vw \in E' \setminus E$

(that is $x_{vw} = 0$), there must exist a corresponding path of edges in E, connecting nodes v and w in the original graph G, that are non-cut (that is $x_e = 0$, for $e \in E$). An application of this minimum cost lifted multicut problem is to fuse multiple feasible solutions for image segmentation [Beier et al., 2016].

2.1.3 Message Passing Algorithms

Let us first introduce a factor graph that will be associated with the energy function later. Given a vector of variables $\mathbf{x} = \{x_1, ..., x_n\}$, a factor graph is a bipartite graph that represents the factorization of a function $g(\cdot)$ as follows,

$$g(\mathbf{x}) = \prod_{a \in \mathcal{F}} f_a(\mathbf{x}_a) \ , \tag{2.24}$$

where $\mathbf{x}_a$ is a vector of variable vertices belonging to $\mathbf{x}$, f_a is a factor vertex connecting a subset of variable vertices, and $\mathcal{F}$ is a set of factor vertices. There is an undirected edge connecting factor vertex f_a and variable vertex (or vertices) x_k if and only if $x_k \in \mathbf{x}_a$.

For instance, for $\mathbf{x} = \{x_1, x_2\}$ with two variable vertices x_1 and x_2, $\mathbf{x}_a$ could be $\{x_1\}, \{x_2\}$, and $\{x_1, x_2\}$. Thus, for one case, Eq. (2.24) can be written as

$$g(\mathbf{x}) = f_1(x_1)\ f_2(x_2)\ f_3(x_1, x_2) \ . \tag{2.25}$$

For $f_3(x_1, x_2)$, there is an edge connecting f_3 (a factor vertex) and x_1 (a variable vertex) as well as an edge connecting f_3 and x_2, where messages are passed from f_3 to x_1 and x_2, denoted by $m_{a \to v}$ through this edge and vice versa, denoted by $m_{v \to a}$. A message contains the effect of one variable on another.

In a general form, a message passing from a variable vertex to a factor vertex and vice versa can be expressed as

- For a message passing from a variable vertex v to a factor vertex a, it contains the effects of the neighboring variable vertices of v, that is $\mathcal{N}(v)$, by

$$m_{v \to a}(x_a) = \prod_{k \in \mathcal{N}(v) \setminus \{a\}} m_{k \to v}(x_v) \ . \tag{2.26}$$

- For a message passing from a factor vertex a to a variable vertex v, it is the sum-product of a factor with messages from all the other variable vertices,

$$m_{a \to v}(x_v) = \sum_{\mathbf{x}'_a : x'_v = x_v} f_a(\mathbf{x}'_a) \prod_{k \in \mathcal{N}(a) \setminus \{v\}} m_{k \to a}(x'_k) \ . \tag{2.27}$$

Relation to Energy Function. One can rewrite Eq. (2.1) from the energy perspective to the probability perspective as

$$P(\mathbf{x}) = \frac{1}{Z} \exp(-E(\mathbf{x}_c)) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \exp(-E_C(\mathbf{x}_c)) = \frac{1}{Z} \prod_{c \in \mathcal{C}} f_c(\mathbf{x}_C) \ , \tag{2.28}$$

where Z is a normalizing constant. It is obvious that the energy function in a MRF is exactly in the same form as a factor graph in Eq. (2.24).

2.1.3.1 Iterated Conditional Modes

In probabilistic graphical models, Iterated Conditional Modes (ICM) [Besag, 1986] is defined as a deterministic algorithm to obtain a local configuration in a MRF by maximizing the joint probability of a variable on the condition of fixing the states of the other variables. This is a greedy algorithm. As ICM is widely used as an image denoiser, we give the following energy function on a noisy image $\mathbf{I}$ and a label set $\mathcal{L}$ at the k-th iteration as an example.

$$E(\mathbf{x}|\mathbf{I}) = \sum_{i \in \mathcal{V}} (1 - \delta(x_i, I_i)) + \lambda \sum_{(i,j) \in \mathcal{E}} (1 - \delta(x_i, x_j^k)) \ , \tag{2.29}$$

where λ is a coefficient for the degree of label smoothness and

$$\delta(x_i, x_j) = \begin{cases} 1 & \text{if } x_i = x_j \\ 0 & \text{otherwise} \end{cases} \ . \tag{2.30}$$

For a given node i, the corresponding energy (or cost) is

$$E_i(\mathbf{x}|\mathbf{I}) = 1 - \delta(x_i, I_i) + \lambda \sum_{(i,j) \in \mathcal{E}} (1 - \delta(x_i, x_j^k)) \ . \tag{2.31}$$

Then, the optimal label of node i is

$$x_i^{k+1} = x_i^* = \underset{x_i \in \mathcal{L}}{\operatorname{argmin}} \, E_i(\mathbf{x}|\mathbf{I}) \ . \tag{2.32}$$

Implementations of ICM could be in a sequential form from the first node to the last one while in each iteration the renewed state of a node is used for the next node or in a parallel form that the state of each node is totally independent on the rest in each iteration, then in the next iteration they will be renewed upon the node states from the former iteration. One can also consider a block-wise form such that each block is independent on the rest for parallelism.

2.1.3.2 Semi-Global Matching

Hirschmuller [2008] proposed Semi-Global Matching (SGM) for stereo vision using a fast approximation of 1-dimensional energy minimization in polynomial time. The core is to perform dynamic programming over a selected vector of graph nodes in a specific direction. In this case, the minimum cost of a node is a min-marginal along that direction. Then, the optimal labels can be obtained by tracing back the edges containing the minimum costs, where such an edge connects two optimal labels.

With the same notations for the energy function in Eq. (2.13), the objective func-

tion of SGM is defined by

$$E(\mathbf{x};\Theta) = \sum_{i\in\mathcal{V}} \theta_i(x_i) + \sum_{(i,j)\in\mathcal{E}} V_1 P[|x_i - x_j| = 1] + \sum_{(i,j)\in\mathcal{E}} V_2 P[|x_i - x_j| \geq 2] \ , \tag{2.33}$$

where $x_i \in \mathcal{L}$ in a multilabel set, V_1 and V_2 are coefficients, $V_1 <= V_2$ in general, and $P[\cdot]$ is an overcomplete probability function with value 1 when the agreement holds and otherwise 0.

In a specific direction $r \in \mathcal{D}$, the subproblem of minimizing the energy function is to find the min-marginal of each node i along r (not a sum of costs from all directions). For simplicity, we replace x_i by d. Then, the cost function to be minimized is

$$\begin{aligned} L_r(i,d) = \theta_i(d) + \min(&L_r(i-r,d), \\ &L_r(i-r,d-1) + V_1, \\ &L_r(i-r,d+1) + V_1, \\ &\min_{j\geq 2, j\in\mathcal{V}} L_r(i-r,d\pm j) + V_2) \ , \end{aligned} \tag{2.34a}$$

$$L_r(i,d) = L_r(i,d) - \min_{k\in\mathcal{L}} L_r(i-r,k) \ , \tag{2.34b}$$

where the last subtraction is to avoid a continuous increase of costs as i increases. The final cost of node i at label d is

$$c_r(i,d) = \sum_{r\in\mathcal{D}} L_r(i,d) \ . \tag{2.35}$$

The corresponding optimal label is obtained by

$$d^* = \operatorname*{argmin}_{d\in\mathcal{L}} c_r(i,d) \ . \tag{2.36}$$

2.1.3.3 Sequential Tree-Reweighted Message Passing

Given a graph $G = (\mathcal{V}, \mathcal{E})$ with a set of nodes $\mathcal{V}$, a set of edges $\mathcal{E}$, and a label set $\mathcal{L}$, an explicit representation of the multilabel energy function in Eq. (2.13) with potentials $\theta = \{\theta_i, \theta_{ij}\}$ can be written as

$$E(\mathbf{x};\theta) = \sum_{i\in\mathcal{V}} \theta_i(x_i) + \sum_{(i,j)\in\mathcal{E}} \theta_{ij}(x_i, x_j) \ , \tag{2.37}$$

which can be rewritten with binary indicators $\phi_i(\lambda) \in \{0,1\}$, where $\phi_i(\lambda) = 1$ means $x_i = \lambda$ and $\phi_i(\lambda) = 0$ for $x_i \neq \lambda$, for every fixed label $\lambda \in \mathcal{L}$ as

$$E(\boldsymbol{\phi};\theta) = \sum_{i\in\mathcal{V}} \sum_{\lambda\in\mathcal{L}} \theta_i(\lambda)\ \phi_i(\lambda) + \sum_{(i,j)\in\mathcal{E}} \sum_{\lambda,\mu\in\mathcal{L}} \theta_{ij}(\lambda,\mu)\ \phi_i(\lambda)\ \phi_j(\mu) \ . \tag{2.38}$$

The solution of such ϕ is an exact solution to the energy minimization of Eq. (2.37), corresponding to the solution of $\mathbf{x}$. This, however, is NP-hard. An approximation using a Linear Programming (LP) relaxation on the indicators is usually used such that $\phi_i(x_i) \in [0,1]$ is real-value and in a **local polytope** of G.

We now introduce tree-reweighted message passing related algorithms. Max-product tree-reweighted message passing (TRW) was first introduced in Wainwright et al. [2005a]; Jordan [1998] in the probabilistic graphical model (by maximizing a factored probabilistic model function). Modified by Kolmogorov in Kolmogorov [2006], sequential TRW was proposed from the perspective of min-sum energy minimization. Monotonicity is guaranteed by monotonic chains (simply understood as a single chain from the first node to the last one in a graph).

By inner-product, the energy function is expressed as

$$E(\mathbf{x};\theta) = \langle \theta, \boldsymbol{\phi}(\mathbf{x}) \rangle = \sum_{c \in \mathcal{C}} \theta_c \, \phi_c(\mathbf{x}) \ , \tag{2.39}$$

where $\mathcal{C}$ is a set of cliques of different clique sizes. While $\boldsymbol{\phi}(\cdot)$ is in a local polytope of G, minimizing Eq. (2.39) is to find a lower bound on the exact solution.

Given clique size 1 and 2, their **min-marginals** can be written as

$$\begin{aligned} \Phi_{i:\lambda}(\phi) &= \min_{\mathbf{x}, x_i = \lambda} E(\mathbf{x};\theta) \ , \\ \Phi_{ij:\lambda\mu}(\phi) &= \min_{\mathbf{x}, x_i = \lambda, x_j = \mu} E(\mathbf{x};\theta) \ . \end{aligned} \tag{2.40}$$

Decomposing graph G into trees, each tree $T \in \mathcal{T}$, we have a larger parameter vector $\boldsymbol{\theta} = \{\theta^T | T \in \mathcal{T}\}$. Note that these θ^Ts will be different from the initial values $\bar{\theta}$ after reparameterization. Then, the range of parameters is extended from the initial $\theta \in \mathbb{R}^{|\mathcal{C}|}$ to $\boldsymbol{\theta} \in \mathbb{R}^{|\mathcal{C}| \times |\mathcal{T}|}$.

Now, the original energy minimization problem can be solved by maximizing its lower bound by

$$\max_{\boldsymbol{\theta} \in \boldsymbol{\mathcal{A}}, \sum_T \rho^T \theta^T \equiv \bar{\theta}} \sum_T \rho^T \min_{\mathbf{x}} \langle \boldsymbol{\theta}^T, \boldsymbol{\phi}(\mathbf{x}) \rangle \ , \tag{2.41}$$

where $\boldsymbol{\mathcal{A}}$ is a set containing all possible $\boldsymbol{\theta} \in \mathbb{R}^{|\mathcal{C}| \times |\mathcal{T}|}$ and ρ^T is a coefficient to ensure the condition $\sum_T \rho^T \theta^T \equiv \bar{\theta}$ in the tree decomposition. Reparameterization of these $\boldsymbol{\theta}$ follows

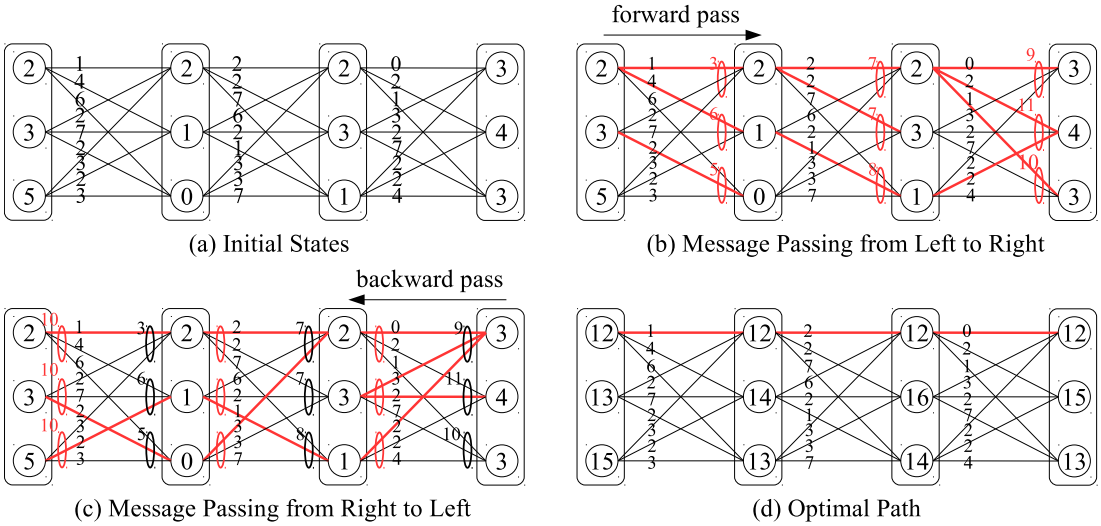
$$\begin{aligned} \theta_{i:\lambda} &= \bar{\theta}_{i:\lambda} + \sum_{(j \to i) \in \mathcal{E}} m_{ji:\lambda} \ , \\ \theta_{ij:\lambda\mu} &= \bar{\theta}_{ij:\lambda\mu} - m_{ji:\lambda} - m_{ij:\mu} \ , \end{aligned} \tag{2.42}$$

where message is calculated by

$$m_{ji:\lambda} = \min_{\mu \in \mathcal{L}} \gamma_{ji} (\bar{\theta}_{j:\mu} + \sum_{(i \to j) \in \mathcal{E}} m_{ij:\mu}) - m_{ij:\mu} + \bar{\theta}_{ji}(\mu, \lambda) \ , \tag{2.43}$$

where $\gamma_{ji} = \rho_{ji} / \rho_j$. The message update in Kolmogorov [2006] follows Eq. (2.43) in two directions, that is where $j > i$ and an inverse direction where $j < i$, and all nodes

of graph G are chosen as a single monotonic chain from the first node to the last one and then from the last one to the first one (corresponding to the two directions). One can refer to Kolmogorov [2006] for more details.



(a) Initial States (b) Message Passing from Left to Right

(c) Message Passing from Right to Left (d) Optimal Path

Figure 2.1: *TRWS example of 4 nodes with 3 labels. The minimum message of a node at a specific label is from all messages on its connected edges, marked by "red circle" for a min(·) operation. Each column is a node containing 3 unary potentials, such as 2, 3, 5 in the first column in (a). Each edge is with a pairwise potential, such as 1, 4, 6 on the edge in (a). The optimal path, marked by "red edges" in (d), is obtained by finding the corresponding costs, that is the sum of unary potentials and massages passed from the left side, in (b), and the right side, in (c).*

[Procedures] The calculation of the lower bound and updates of messages in TRWS follows the steps below: initializing all messages $m_{ij:\mu}$ and $m_{ji:\lambda}$ to 0, then

$$\begin{aligned}
\theta_{i:\lambda} &= \bar{\theta}_{i:\lambda} + \sum_{\substack{\lambda\in\mathcal{L}\\(k,i)\in\mathcal{E}}} m_{ki:\lambda} \ , \\
\Delta_1 &= \min_{\lambda\in\mathcal{L}} \theta_{i:\lambda} \ , \\
\theta_{i:\lambda} &\leftarrow \theta_{i:\lambda} - \Delta_1 \ , \\
\theta_{ij:\lambda\mu} &= \bar{\theta}_{ij:\lambda\mu} - m_{ji:\lambda} \ ,
\end{aligned} \tag{2.44a}$$

and

$$\begin{aligned}
m_{ij:\mu} &\leftarrow \min_{\lambda\in\mathcal{L}} \gamma\theta_{i:\lambda} + \theta_{ij:\lambda\mu} \ , \\
\Delta_2 &= \min_{\mu\in\mathcal{L}} m_{ij:\mu} \ , \\
m_{ij:\mu} &\leftarrow m_{ij:\mu} - \Delta_2 \ , \\
\Delta &\mathrel{+}= \Delta_1 + \Delta_2 \ , \\
\text{s.t.} \quad \lambda,\mu &\in \mathcal{L} \ , \\
i,j &\in \mathcal{V} \ , \\
(i,j) &\in \mathcal{E} \ ,
\end{aligned} \tag{2.44b}$$

where $\overline{\theta}_{i:\lambda}$ is the original unary potential of node i over label λ, $\overline{\theta}_{ij:\lambda\mu}$ is the original edge potential of nodes i and j over labels λ and μ respectively, $m_{ki:\lambda}$ is a message from node k to node i over label λ, $\mathcal{V}$ is a set of N nodes, $\mathcal{E}$ is a set of E edges connecting every two neighbour nodes, γ is an edge appearance over the node appearance in the tree graphs, θ_i and θ_{ij} are updated unary potential and pairwise potential respectively, and Δ is a lower bound consisting of two components Δ_1 and Δ_2.

We show an example of TRWS in Fig. 2.1 with an explanation in the caption.

2.1.3.4 Lazy Message Passing

Lazy message passing aims at iteratively reparameterizing the original potentials $\overline{\theta}_{i:\lambda}$ and $\overline{\theta}_{ij:\lambda\mu}$ by sweeping nodes of tree graphs in multiple scanning directions. When sweeping nodes over a tree in one direction, the two nodes of each edge will be reparameterized by pushing all messages from one node (source node) to the other node (target node) via this edge.

Next, when all source nodes push messages to the target nodes, a source node will pull a message from the target node via all edges connecting this target node. In other words, it retrieves a "redundant" message after pushing all messages via the edges.

[Procedures] With the same notations in Eq. (2.44),

$$\begin{aligned}
\overline{\theta}_{ij:\lambda\mu} &\leftarrow \overline{\theta}_{i:\lambda} + \overline{\theta}_{ij:\lambda\mu}, \quad \forall\mu \ , \\
\overline{\theta}_{i:\lambda} &\leftarrow 0 \ , \\
m_{ij:\mu} &= \min_{\lambda\in\mathcal{L}} \overline{\theta}_{ij:\lambda\mu} \ , \\
\overline{\theta}_{j:\mu} &\leftarrow \overline{\theta}_{j:\mu} + m_{ij:\mu} \ , \\
\Delta_1 &= \min_{\mu\in\mathcal{L}} \overline{\theta}_{j:\mu} \ , \\
\overline{\theta}_{j:\mu} &\leftarrow \overline{\theta}_{j:\mu} - \Delta_1 \ , \\
\overline{\theta}_{ij:\lambda\mu} &\leftarrow \overline{\theta}_{ij:\lambda\mu} - m_{ij:\mu} \ , \quad \forall\lambda,
\end{aligned} \tag{2.45a}$$

and

$$\begin{aligned}
m_{ji:\lambda} &= \min_{\mu\in\mathcal{L}} \overline{\theta}_{ij:\lambda\mu} \ , \\
\overline{\theta}_{i:\lambda} &\leftarrow \overline{\theta}_{i:\lambda} + m_{ji:\lambda} \ , \\
\overline{\theta}_{ij:\lambda\mu} &\leftarrow \overline{\theta}_{ij:\lambda\mu} - m_{ji:\lambda}, \quad \forall\mu \ , \\
\Delta &\mathrel{+}= \Delta_1 \ , \\
\text{s.t.} \quad \lambda,\mu &\in \mathcal{L} \ , \\
i,j &\in \mathcal{V} \ , \\
(i,j) &\in \mathcal{E} \ .
\end{aligned} \tag{2.45b}$$

After reparameterization, we expect as many zero nodes and zero edges as possible. In each tree, the unary potentials will accumulate along the scanning direction from

source nodes to target nodes due to the increasing messages pushed through the edges, unless a Δ is subtracted as a lower bound component.

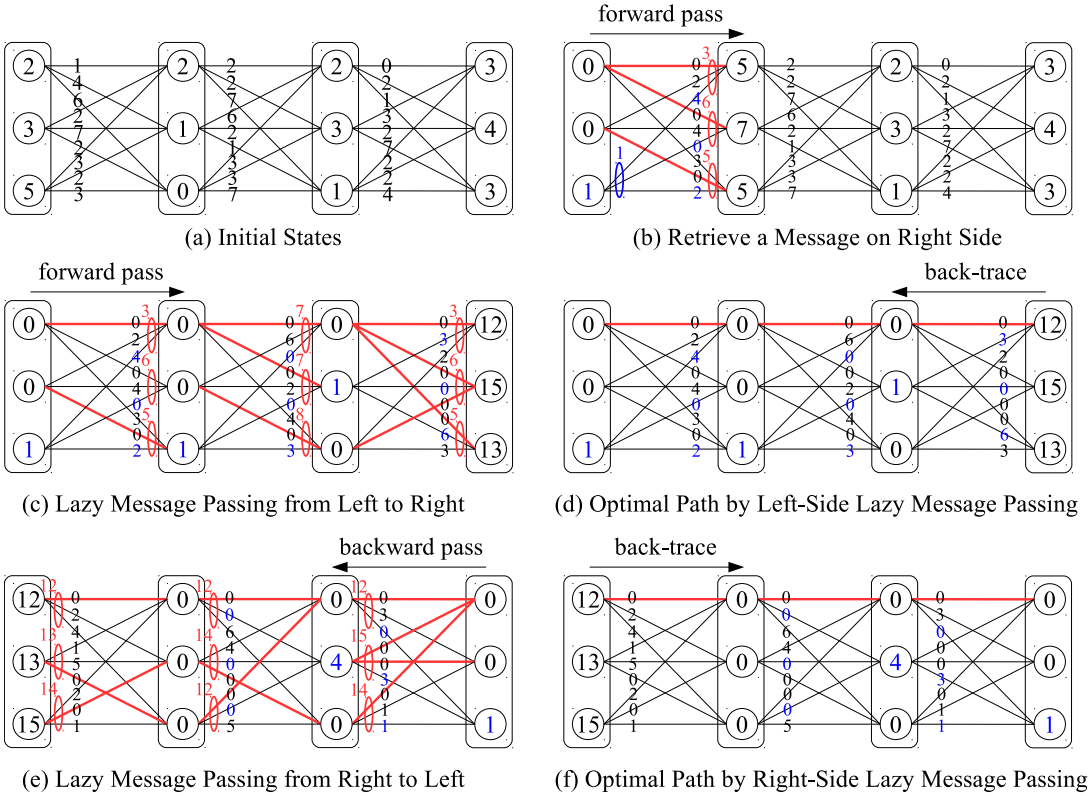


Figure 2.2: *Lazy message passing example of 4 nodes with 3 labels. The core is to push all unary potentials to the edges such that after the unary reparameterization, all unary potentials become zero. Then, redundant messages are retrieved in the inverse direction. These retrieved messages are marked blue as updated unary potentials. (b) is one case of (c) while (c) and (e) are the overall lazy message passing from the left side and the right side respectively. The optimal path in (d) is corresponding to the passing in (c), while (f) is to (e). The final optimal paths, "red edges" in (d) and (f), are the same, indicating that one direction lazy message passing is sufficient to find the optimal path.*

We show an example of the lazy message passing in Fig. 2.2 with an explanation in the caption.

2.1.3.5 Min-Marginal Message Passing

A min-marginal labeling targets at passing messages to a given node at a specific label from all the rest nodes in a subgraph, possibly a tree. Thus, the min-marginal energy of such a node is

$$\begin{aligned} \Psi_{i:\lambda} &= \min_{\mathbf{x}|x_i=\lambda} E(\mathbf{x}) \ , \\ \text{s.t.} \quad & \lambda \in \mathcal{L} \ , \end{aligned} \tag{2.46}$$

where $\Psi_{i:\lambda}$ is the min-marginal energy of node i at label λ and $E(\mathbf{x})$ is the energy corresponding to labeling $\mathbf{x}$.

The procedures of the min-marginal message passing are similar to TRWS except that 1) the optimal path can be decided by the connections between every two nodes with min-marginals instead of back-tracing from the end nodes and that 2) every node or edge of a tree can be an end node or edge, which is for the end of a message passing, while in TRWS only the first node or the last node of the tree ends a message passing.

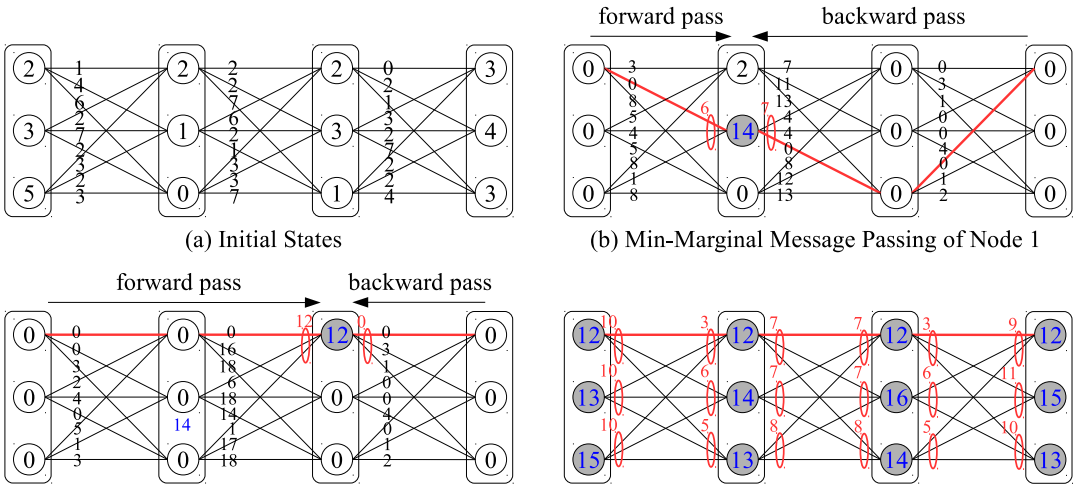


Figure 2.3: *Min-marginal message passing example of 4 nodes with 3 labels. This aims at a specific node or edge to which the messages are passed from different directions, from the left and right sides in our case. The message update is the same as lazy message passing while no message retrieving is required. A min-marginal is the minimum cost using the sum of unary potential and its surrounding messages. (b) and (c) are two cases of (d). One can easily find the optimal path, "red edges" in (d), by selecting the edges connecting each two nodes with min-marginals.*

We show an example of the min-marginal message passing in Fig. 2.3 with an explanation in the caption.

The message passing inference can also be found in Graph Convolutional Networks (GCN), see the review in Zhang et al. [2019b] and Battaglia1 et al. [2018] for more details. GCN is efficient when the number of involved nodes and edges is small with sparse inputs, for instance, in human pose estimation, molecular structure reconstruction, and text prediction. For dense inputs, such as images or videos with each having latent variables, however, GCN is less efficient than direct parallel programming on GPU along each scanning direction and associate labels. The benefits of this parallel programming along the scanlines were indicated in the aforementioned message passing examples.

2.2 Convolutional Neural Network Pruning

This section has values in modern convolutional neural networks and a correlation with Chapter 5.

Recent years witness the development of convolutional neural networks in various computer vision and machine learning applications. Some of the network models, however, are rather computationally expensive and memory inefficient, requiring a long training time and massive GPU memory. Hence, accelerating and compressing such huge models becomes important and practical. Among these research areas, typical directions include model pruning, quantization, binarization, low-rank factorization, knowledge distillation, and so on. Specifically, model pruning arises a lot of attention and interests due to the direct effects on network hidden layers and the memory for parameter and feature storage.

As a warm start of Chapter 5 for 3D CNN pruning, we briefly introduce the background of parameter and neuron (also known as channel) pruning and their difference from a practical perspective.

2.2.1 Parameter Pruning

A convolutional neural network consists of hierarchical convolution filters. The number of such filters constructs the depth of a network model, and the number of features, that is the output of a convolution layer, forms the width of the network model. Given that the dimension of a convolution layer is (out_channel, in_channel, h, w), where h and w denote height and width of a kernel, parameter pruning aims at removing redundant (that is unimportant) elements of these out_channel $\times$ in_channels $\times$ h $\times$ w values. This results in a sparse matrix where generally 0 means the element is removed and 1 if retained.

Even though such a sparse matrix leads to less non-zero parameters than the original matrix without parameter pruning, it hardly reduces the complexity of convolution operations and the memory for parameters and hidden layers unless sparse convolution operations (sparse matrix multiplication and addition) are used and the sparse matrices are stored in a sparse manner, such as hash tables. Hence, we further introduce neuron, also known as channel, pruning, which is comparable to the parameter pruning.

2.2.2 Neuron Pruning

Still, given a layer with (out_channel, in_channel, h, w) parameters, a neuron generally refers to the one in the out_channel dimension. Therefore, neuron pruning aims at searching for such redundant neurons and removing them from the layers. In this case, labeling a neuron redundant means all (in_channel, h, w) elements are redundant. The advantage of neuron pruning over parameter pruning is the unnecessity of a sparse matrix and sparse convolution operations but simply to generate narrow layers followed by the ordinary dense convolution operations.

This not only reduces the memory of parameters but also, more importantly, the memory required by hidden layers, especially for 3D network models. One can refer to "Motivation" in Chapter 5 for more details and concrete samples.

Chapter 3

Optimization of Markov Random Fields in Deep Learning

In this chapter, we observe the existing limitations of the effectiveness of optimization ability and the efficiency for a fast inference, in some selected Markov Random Fields (MRF) optimization algorithms, such as mean-field [Zheng et al., 2015] and Semi-Global Matching (SGM) [Seki and Pollefeys, 2017] methods, in deep learning semantic segmentation and stereo vision. Hence, we propose two effective and efficient MRF optimization algorithms, namely Iterative Semi-Global Matching Revised (ISGMR) and Parallel Tree-Reweighted Message Passing (TRWP), for both optimization problems and deep learning with differentiability and fast inference using CUDA programming with tree decomposition.

3.1 Motivation

Optimization of MRFs has been a well-studied problem for decades with a significant impact on many computer vision applications such as stereo vision [Hirschmuller, 2008], image segmentation [Boykov and Jolly, 2011], texture modeling [Hassner and Sklansky, 1980]. The widespread use of these MRF optimization algorithms is currently limited due to imperfect MRF modelling [Szeliski et al., 2008] because of hand-crafted model parameters, the usage of inferior inference methods, and non-differentiability for parameter learning. Thus, better inference capability and computing efficiency are essential to improve its performance on optimization and modelling, such as energy optimization and end-to-end learning.

Even though parameter and structural learning with MRFs has been employed successfully in certain cases, well-known algorithms such as Mean-Field (MF) [Zheng et al., 2015; Krähenbühl and Koltun, 2011] and Semi-Global Matching (SGM) [Seki and Pollefeys, 2017], are suboptimal in terms of optimization capability. Specifically, the choice of an MRF algorithm for optimization is driven by its inference ability, and for learning capability through efficient forward and backward propagation and parallelization capabilities.

In this work, we consider message passing algorithms due to their generality, high inference ability, and differentiability, and provide efficient CUDA implementations

of their forward and backward propagation by exploiting massive parallelism. In particular, we revise the popular SGM method [Hirschmuller, 2008] and derive an iterative version noting its relation to traditional message passing algorithms [Drory et al., 2014]. In addition, we introduce a highly parallelizable version of the state-of-the-art Sequential Tree-Reweighted Message Passing (TRWS) algorithm [Kolmogorov, 2006], which is more efficient than TRWS and has similar minimum energies. For both these methods, we derive efficient backpropagation by unrolling their message updates and cost aggregation and discuss massively parallel CUDA implementations which enable their feasibility in end-to-end learning.

Our experiments on the standard stereo and denoising benchmarks demonstrate that our Iterative and Revised SGM method (ISGMR) obtains much lower energies compared to the standard SGM and our Parallel TRW method (TRWP) is two orders of magnitude faster than TRWS with virtually the same minimum energies and that both outperform the popular MF and SGM inferences. Their performance is further evaluated by the end-to-end learning for semantic segmentation on PASCAL VOC 2012 dataset.

Furthermore, we empirically evaluate various implementations of the forward and backward propagation of these algorithms and demonstrate that our CUDA implementation is the fastest, with *at least 690 times speed-up* in backpropagation compared to a PyTorch GPU version.

Contributions of this chapter can be summarised as:

- We introduce two message passing algorithms, ISGMR and TRWP, where ISGMR has higher optimization effectiveness than SGM and TRWP is much faster than TRWS. Both of them outperform the popular SGM and MF inferences.
- Our ISGMR and TRWP are massively parallelized on GPU and can support any pairwise potentials. The CUDA implementation of the backpropagation is at least 690 times faster than the PyTorch auto-gradient version on GPU.
- The differentiability of ISGMR and TRWP is presented with gradient derivations, with effectiveness validated by end-to-end learning for semantic segmentation.

3.2 Related Work

In MRF optimization, estimating the optimal latent variables can be regarded as minimizing a particular energy function with given model parameters. Even if the minimum energy is obtained, high accuracy cannot be guaranteed since the model parameters of these MRFs are usually handcrafted and imperfect. To tackle this problem, learning-based methods were proposed. However, most of these methods rely greatly on finetuning the network architecture or adding learnable parameters to increase the fitting ability with ground truth. This may not be effective and usually requires high GPU memory.

Nevertheless, considering the highly effective MRF optimization algorithms, the field of exploiting their optimization capability with parameter learning to alleviate each other's drawbacks is rarely explored. A few works provide this capability in certain cases, such as CRFasRNN in semantic segmentation [Zheng et al., 2015] and SGMNet in stereo vision [Seki and Pollefeys, 2017], with less effective MRF algorithms, that is MF and SGM respectively. Meanwhile, although graph convolutional networks, see a review in Zhang et al. [2019b] and Battaglia1 et al. [2018], present message passing inference by aggregating information from connected nodes and then diffusing the update information, it is inefficient when the input data has a large data size or a high dimension. Thus, it is important to adopt highly effective and efficient MRF inference algorithms for optimization and end-to-end learning.

3.2.1 MRF Optimization

Determining an effective MRF optimization algorithm needs a thorough study of the possibility of their optimization capability, differentiability, and time efficiency. In the two main categories of MRF optimization algorithms, namely move-making algorithms (known as graph cuts) [Ajanthan et al., 2016, 2015; Boykov et al., 2001b; Carr and Hartley, 2009; Hartley and Ajanthan, 2018; Veksler, 2012] and message passing algorithms [Hirschmuller, 2008; Jordan, 1998; Kwon et al., 2010; Kolmogorov, 2006; Murphy et al., 1999; Pearl, 1988; Wainwright and Jordan, 2008; Wang et al., 2014], the state-of-the-art methods are α-expansion [Boykov et al., 2001b] and Sequential Tree-Reweighted Message Passing (TRWS) [Kolmogorov, 2006] respectively. The move-making algorithms, however, cannot easily be used for parameter learning as they are not differentiable and are usually limited to certain types of energy functions.

In contrast, message passing algorithms adapt better to any energy functions and can be made differentiable and fast if well designed. Some works in probabilistic graphical models indeed demonstrate the learning ability of TRW algorithms with sum-product and max-product [Jordan, 1998; Wainwright and Jordan, 2008] message passing. A comprehensive study and comparison of these methods can be found in Middlebury [Szeliski et al., 2008] and OpenGM [Kappes et al., 2013]. Although Hirschmuller [2008] is not in the benchmark, it was proved to have a high running efficiency due to the fast one-dimensional Dynamic Programming (DP) that is independent in each scanline and scanning direction [Hirschmuller, 2008].

3.2.2 End-to-End Learning

Sum-product TRW [Domke, 2013; Taskar et al., 2003; Tsochantaridis et al., 2005] and mean-field [Zheng et al., 2015; Liu et al., 2015b; Lin et al., 2016] have been used for end-to-end learning for semantic segmentation, which presents their highly effective learning ability. Meanwhile, for stereo vision, several MRF/CRF based methods [Seki and Pollefeys, 2017; Zhang et al., 2019a; Knobelreiter et al., 2017], such as SGM-related, have been proposed. These further indicate the high efficiency of selected MRF optimization algorithms in end-to-end learning.

In our work, we improve optimization effectiveness and time efficiency based on classical SGM and TRWS. In particular, we revise the standard SGM and make it iterative in order to improve its optimization capability. We denote the resulting algorithm as ISGMR. Our other algorithm, TRWP, is a massively parallelizable version of TRWS, which greatly increases running speed without losing the optimization effectiveness.

3.3 Message Passing Algorithms

We first briefly review the typical form of a pairwise MRF energy function and discuss two highly parallelizable message passing approaches, ISGMR and TRWP. Such a parallelization capability is essential for fast implementation on GPU and enables relatively straightforward integration to existing deep learning models.

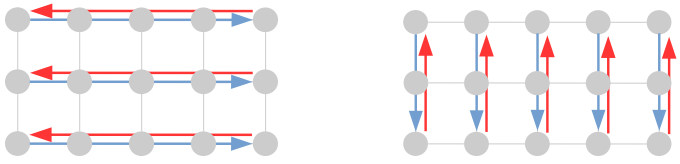
3.3.1 Pairwise MRF Energy Function

Let X_i be a random variable taking label $x_i \in \mathcal{L}$. A pairwise MRF energy function defined over a set of such variables, parametrized by $\mathbf{\Theta} = \{\theta_i, \theta_{i,j}\}$, is written as

$$E(\mathbf{x} \mid \mathbf{\Theta}) = \sum_{i \in \mathcal{V}} \theta_i(x_i) + \sum_{(i,j) \in \mathcal{E}} \theta_{i,j}(x_i, x_j) \ , \tag{3.1}$$

where θ_i and $\theta_{i,j}$ denote unary potentials and pairwise potentials respectively, $\mathcal{V}$ is the set of vertices (corresponding, for instance, to image pixels or superpixels), and $\mathcal{E}$ is the set of edges in the MRF (usually encoding a 4-connected or 8-connected grid).

3.3.2 Iterative Semi-Global Matching Revised



(a) (b)

Figure 3.1: *An example of 4-connected SGM on a grid MRF: left-right, right-left, up-down, and down-up. Message passing along all these scanlines can be accomplished in parallel.*

We first introduce the standard SGM for stereo vision supporting only a single iteration. With its connection to message passing, we then revise its message update equation and introduce an iterative version. Fig. 3.1 shows a 4-connected SGM on a grid MRF.

3.3.2.1 Revised Semi-Global Matching

We cast the popular SGM algorithm [Hirschmuller, 2008] as an optimization method for a particular MRF and discuss its relation to message passing as noted in Drory et al. [2014]. In SGM, pairwise potentials are simplified for all edges $(i, j) \in \mathcal{E}$ as

$$\theta_{i,j}(\lambda, \mu) = \theta_{i,j}(|\lambda - \mu|) = \begin{cases} 0 & \text{if } \lambda = \mu \ , \\ P_1 & \text{if } |\lambda - \mu| = 1 \ , \\ P_2 & \text{if } |\lambda - \mu| \geq 2 \ , \end{cases} \tag{3.2}$$

where $0 < P_1 \leq P_2$. The idea of SGM relies on cost aggregation in multiple directions (each direction having multiple one-dimensional scanlines) using Dynamic Programming (DP). The main observation made by Drory et al. [2014] is that, in SGM the unary potentials are over-counted $|\mathcal{R}| - 1$ times (where $\mathcal{R}$ denotes the set of directions) compared to the standard message passing and this over-counting corrected SGM is shown to perform slightly better in Facciolo et al. [2015]. Noting this, we use symbol $m_i^r(\lambda)$ to denote the message-vector passed **to** node i, along a scan-line in the direction r, **from** the previous node, denoted $i - r$. This is a vector indexed by $\lambda \in \mathcal{L}$. Now, the SGM update is *revised* from

$$m_i^r(\lambda) = \min_{\mu \in \mathcal{L}} \left(\theta_i(\lambda) + m_{i-r}^r(\mu) + \theta_{i-r,i}(\mu, \lambda) \right) \ , \tag{3.3}$$

which is the form given in Hirschmuller [2008], to

$$m_i^r(\lambda) = \min_{\mu \in \mathcal{L}} \left(\theta_{i-r}(\mu) + m_{i-r}^r(\mu) + \theta_{i-r,i}(\mu, \lambda) \right) \ . \tag{3.4}$$

The $m_i^r(\lambda)$ represents the minimum cost due to possible assignments to all nodes previous to node i along the scanline in direction r, and assigning label λ to node i. It does not include the cost $\theta_i(\lambda)$ associated with node i itself.

Since subtracting a fixed value for all λ from messages preserves minima, the message $m_i^r(\lambda)$ can be reparametrized as

$$m_i^r(\lambda) = m_i^r(\lambda) - \min_{\mu \in \mathcal{L}} m_i^r(\mu) \ , \tag{3.5}$$

which does not alter the minimum energy. Since the values of $\theta_i(\lambda)$ are not included in the messages, the final cost at a particular node i at label λ is *revised* from

$$c_i(\lambda) = \sum_{r \in \mathcal{R}} m_i^r(\lambda) \tag{3.6}$$

to

$$c_i(\lambda) = \theta_i(\lambda) + \sum_{r \in \mathcal{R}} m_i^r(\lambda) \ , \tag{3.7}$$

which is the sum of messages over all the directions plus the unary term. The final

labelling is then obtained by

$$x_i^* = \underset{\lambda \in \mathcal{L}}{\operatorname{argmin}}\, c_i(\lambda) \;, \quad \forall\, i \in \mathcal{V} \;. \tag{3.8}$$

Here, the message update in the revised SGM, *i.e.*, Eq. (3.4), is performed in parallel for all scanlines for all directions. This massive parallelization makes it suitable for real-time applications [Hernandez-Juare et al., 2016] and end-to-end learning for stereo vision [Seki and Pollefeys, 2017].

3.3.2.2 Iteration of Revised Semi-Global Matching

Algorithm 1: Forward Propagation of ISGMR

Input: Energy parameters $\boldsymbol{\Theta} = \{\theta_i, \theta_{i,j}(\cdot,\cdot)\}$, set of nodes $\mathcal{V}$, edges $\mathcal{E}$, directions $\mathcal{R}$, iteration number K. We replace $m^{r,k}$ by m^r and $m^{r,k+1}$ by $\hat{m}^r$ for simplicity.

Output: Labelling $\mathbf{x}^*$ for optimization, costs $\{c_i(\lambda)\}$ for learning, indices $\{p^r_{k,i}(\lambda)\}$ and $\{q^r_{k,i}\}$ for backpropagation.

$\hat{\mathbf{m}} \leftarrow 0$ and $\mathbf{m} \leftarrow 0$ ▷initialize all messages
for iteration $k \in \{1, \ldots, K\}$ **do**
  forall directions $r \in \mathcal{R}$ **do** ▷**parallel**
    forall scanlines t in direction r **do** ▷**parallel**
      for node i in scanline t **do** ▷**sequential**
        for label $\lambda \in \mathcal{L}$ **do**
          $\Delta(\lambda,\mu) \leftarrow \theta_{i-r}(\mu) + \theta_{i-r,i}(\mu,\lambda) + \hat{m}^r_{i-r}(\mu) + \sum_{d \in \mathcal{R}\setminus\{r,r^-\}} m^d_{i-r}(\mu)$
          $p^r_{k,i}(\lambda) \leftarrow \mu^* \leftarrow \operatorname{argmin}_{\mu\in\mathcal{L}} \Delta(\lambda,\mu)$ ▷store index
          $\hat{m}^r_i(\lambda) \leftarrow \Delta(\lambda,\mu^*)$ ▷message update Eq. (3.9)
        $q^r_{k,i} \leftarrow \lambda^* \leftarrow \operatorname{argmin}_{\lambda\in\mathcal{L}} \hat{m}^r_i(\lambda)$ ▷store index
        $\hat{m}^r_i(\lambda) \leftarrow \hat{m}^r_i(\lambda) - \hat{m}^r_i(\lambda^*)$ ▷reparametrization Eq. (3.5)
  $\mathbf{m} \leftarrow \hat{\mathbf{m}}$ ▷update messages after iteration
$c_i(\lambda) \leftarrow \theta_i(\lambda) + \sum_{r\in\mathcal{R}} m^r_i(\lambda), \forall i \in \mathcal{V}, \lambda \in \mathcal{L}$ ▷Eq. (3.7)
$x^*_i \leftarrow \operatorname{argmin}_{\lambda\in\mathcal{L}} c_i(\lambda), \forall i \in \mathcal{V}$ ▷Eq. (3.8)

In spite of the revision for the over-counting problem, the 3-penalty pairwise potential in Eq. (3.2) is insufficient to obtain dominant penalties under a large range of disparities in different camera settings. To this end, we consider more general pairwise potentials $\theta_{i,j}(\lambda,\mu)$ and introduce an iterative version of the revised SGM. The message update for the iterative version is

$$m_i^{r,k+1}(\lambda) = \min_{\mu\in\mathcal{L}} \left(\theta_{i-r}(\mu) + \theta_{i-r,i}(\mu,\lambda) + m^{r,k+1}_{i-r}(\mu) + \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} m^{d,k}_{i-r}(\mu)\right) , \tag{3.9}$$

where r^- denotes the opposite direction of r and $m^{r,k+1}_{i-r}(\mu)$ denotes the updated

message in kth iteration while $m_{i-r}^{r,k}(\mu)$ is updated in $(k-1)$th iteration. The exclusion of the messages from direction r^- is important to ensure that the update is analogous to the standard message passing and the same energy function is minimized at each iteration. A simple combination of several standard SGMs does not satisfy this rule and performs worse than our iterative version, as reported in Tables 3.1(a)-3.3(a). Usually, $\mathbf{m}^r$ for all $r \in \mathcal{R}$ are initialized to 0, the exclusion of r^- from $\mathcal{R}$ is thus redundant for a single iteration but not multiple iterations. Even so, messages can be reparametrized by Eq. (3.5).

After multiple iterations, the final cost for node $i \in \mathcal{V}$ is calculated by Eq. (3.7), and the final labelling is calculated in the same manner as Eq. (3.8). We denote this iterative and revised SGM as ISGMR, summarized in Algorithm 1.

In sum, the improvement of ISGMR from SGM lies in the exclusion of over-counted unary terms by Eq. (3.4) to increase the effects of pairwise terms as well as the iterative energy minimization by Eq. (3.9) to further decrease the energy with updated messages.

3.3.3 Parallel Tree-Reweighted Message Passing

TRWS [Kolmogorov, 2006] is another state-of-the-art message passing algorithm that optimizes the Linear Programming (LP) relaxation of a general pairwise MRF energy given in Eq. (3.1). The main idea of the family of TRW algorithms [Wainwright et al., 2005b] is to decompose the underlying graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of the MRF with parameters $\boldsymbol{\Theta}$ into a combination of trees where the sum of parameters of all the trees is equal to that of the MRF, *i.e.*, $\sum_{T \in \mathcal{T}} \boldsymbol{\Theta}_T = \boldsymbol{\Theta}$. Then, at each iteration message passing is performed in each of these trees independently, followed by an averaging operation. Even though any combinations of trees would theoretically result in the same final labelling, the best performance is achieved by choosing a monotonic chain decomposition and a sequential message passing update rule, which is TRWS. Interested readers please refer to Kolmogorov [2006] for more details.

Since we intend to enable fast message passing by exploiting parallelism, our idea is to choose a tree decomposition that can be massively parallelized, denoted as TRWP. In the literature, edge-based or tree-based parallel TRW algorithms have been considered, namely, TRWE and TRWT in the probability space (specifically sum-product message passing) rather than for minimizing the energy [Wainwright et al., 2005b]. Optimizing in the probability domain involves exponential calculations which are prone to numerical instability, and the sum-product version requires $\mathcal{O}(|\mathcal{R}||\mathcal{L}|)$ times more memory compared to the min-sum message passing in back-propagation.

Correspondingly, our TRWP directly minimizes the energy in the min-sum message passing fashion similar to TRWS, and thus, its update can be written as

$$m_i^r(\lambda) = \min_{\mu \in \mathcal{L}} \left(\rho_{i-r,i}(\theta_{i-r}(\mu) + \sum_{d \in \mathcal{R}} m_{i-r}^d(\mu)) - m_{i-r}^{r^-}(\mu) + \theta_{i-r,i}(\mu, \lambda) \right) . \tag{3.10}$$

Algorithm 2: Forward Propagation of TRWP

Input: Energy parameters $\mathbf{\Theta} = \{\theta_i, \theta_{i,j}(\cdot,\cdot)\}$, set of nodes $\mathcal{V}$, edges $\mathcal{E}$, directions $\mathcal{R}$, tree decomposition coefficients $\{\rho_{i,j}\}$, iteration number K.

Output: Labelling $\mathbf{x}^*$ for optimization, costs $\{c_i(\lambda)\}$ for learning, indices $\{p^r_{k,i}(\lambda)\}$ and $\{q^r_{k,i}\}$ for backpropagation.

$\mathbf{m} \leftarrow 0$ ▷initialize all messages
for iteration $k \in \{1, \dots, K\}$ **do**
  for direction $r \in \mathcal{R}$ **do** ▷**sequential**
    forall scanlines t in direction r **do** ▷**parallel**
      for node i in scanline t **do** ▷**sequential**
        for label $\lambda \in \mathcal{L}$ **do**
          $\Delta(\lambda,\mu) \leftarrow \rho_{i-r,i}\big(\theta_{i-r}(\mu) + \sum_{d\in\mathcal{R}} m^d_{i-r}(\mu)\big) - m^{r^-}_{i-r}(\mu) + \theta_{i-r,i}(\mu,\lambda)$
          $p^r_{k,i}(\lambda) \leftarrow \mu^* \leftarrow \operatorname{argmin}_{\mu\in\mathcal{L}} \Delta(\lambda,\mu)$ ▷store index
          $m^r_i(\lambda) \leftarrow \Delta(\lambda,\mu^*)$ ▷message update Eq. (3.10)
        $q^r_{k,i} \leftarrow \lambda^* \leftarrow \operatorname{argmin}_{\lambda\in\mathcal{L}} m^r_i(\lambda)$ ▷store index
        $m^r_i(\lambda) \leftarrow m^r_i(\lambda) - m^r_i(\lambda^*)$ ▷reparametrization Eq. (3.5)
$c_i(\lambda) \leftarrow \theta_i(\lambda) + \sum_{r\in\mathcal{R}} m^r_i(\lambda), \forall i \in \mathcal{V}, \lambda \in \mathcal{L}$ ▷Eq. (3.7)
$x^*_i \leftarrow \operatorname{argmin}_{\lambda\in\mathcal{L}} c_i(\lambda), \forall i \in \mathcal{V}$ ▷Eq. (3.8)

Here, the coefficient $\rho_{i-r,i} = \gamma_{i-r,i}/\gamma_{i-r}$, where $\gamma_{i-r,i}$ and γ_{i-r} are the number of trees containing the edge $(i-r,i)$ and the node $i-r$ respectively in the considered tree decomposition. For loopy belief propagation, since there is no tree decomposition, $\rho_{i-r,i} = 1$. For a 4-connected graph decomposed into all horizontal and vertical one-dimensional trees, we have $\rho_{i-r,i} = 0.5$ for all edges.

Note that, similar to ISGMR, we use the scanline to denote a tree. The above update can be performed in parallel for all scanlines in a single direction; however, the message updates over a scanline are sequential. The same reparametrization Eq. (3.5) is applied. While TRWP cannot guarantee the non-decreasing monotonicity of the lower bound of energy, it dramatically improves the forward propagation speed and yields virtually similar minimum energies to those of TRWS. The procedure is in Algorithm 2.

In sum, our TRWP benefits from a high speed-up without losing optimization capability by the massive GPU parallelism over individual trees that are decomposed from the single-chain tree in TRWS. All trees in each direction r are paralleled by Eq. (3.10).

3.3.4 Relation between ISGMR and TRWP

Both ISGMR and TRWP use messages from neighbouring nodes to perform recursive and iterative message updates via dynamic programming. Comparison of Eq. (3.9) and Eq. (3.10) indicates the introduction of the coefficients $\{\rho_{i-r,i}\}$. This is due to the tree decomposition, which is analogous to the difference between loopy belief

propagation and TRW algorithms. The most important difference, however, is the way message updates are defined. Specifically, within an iteration, ISGMR can be parallelized over all directions since the most updated messages $\hat{\mathbf{m}}^r$ are used only for the current scanning direction r and previous messages are used for the other directions (refer Eq. (3.9)). In contrast, aggregated messages in TRWP are up-to-date *direction-by-direction*, which largely contributes to the improved effectiveness of TRWP over ISGMR.

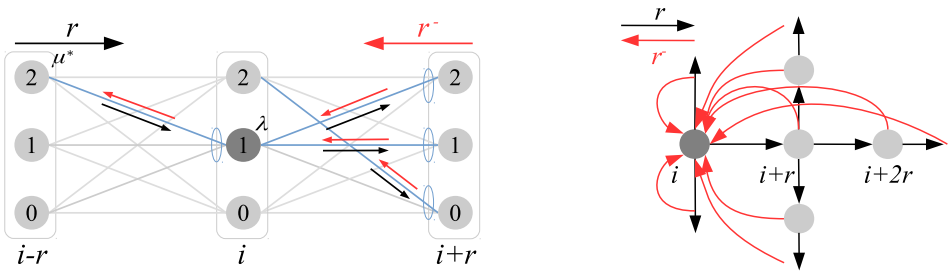
3.3.5 Fast Implementation by Tree Parallelization

Independent trees make the parallelization possible. We implemented on CPU and GPU, where for the C++ multi-thread versions (CPU), 8 threads on Open Multi-Processing (OpenMP) [Dagum and Menon, 1998] are used while for the CUDA versions (GPU), 512 threads per block are used. Each tree is headed by its first node by interpolation. The node indexing details for efficient parallelism are provided in Sec. 3.7.2. In the next section, we derive efficient backpropagation through each of these algorithms for parameter learning.

3.4 Differentiability of Message Passing

Effective and differentiable MRF optimization algorithms can greatly improve the performance of end-to-end learning. Typical methods such as CRFasRNN for semantic segmentation [Zheng et al., 2015] by MF and SGMNet for stereo vision [Seki and Pollefeys, 2017] by SGM use inferior inferences in the optimization capability compared to ISGMR and TRWP.

In order to embed ISGMR and TRWP into end-to-end learning, differentiability of them is required and essential. Below, we describe the gradient updates for the learnable MRF model parameters, and detailed derivations are given in Sec. 3.5. The backpropagation pseudocodes are in Algorithms 3-4.



(a) Message Passing (b) Gradient Accumulation

Figure 3.2: *Forward and backward propagation, a target node is in dark gray, r: forward direction, r^-: backpropagation direction. (a) blue ellipse: min operation as MAP, blue line: an edge having the minimum message. (b) a message gradient at node i accumulated from nodes in r^-.*

Algorithm 3: Backpropagation of ISGMR

Input: Partial energy parameters $\{\theta_{i,j}\}$, gradients of final costs $\nabla \mathbf{c} = \{\nabla c_i(\lambda)\}$, set of nodes $\mathcal{V}$, edges $\mathcal{E}$, directions $\mathcal{R}$, indices $\{p^r_{k,i}(\lambda)\}$, $\{q^r_{k,i}\}$, iteration number K. We replace $\nabla m^{r,k+1}$ by $\nabla \hat{m}^r$ and $\nabla m^{r,k}$ by ∇m^r for simplicity.

Output: Gradients $\{\nabla\theta_i, \nabla\theta_{i,j}(\cdot,\cdot)\}$.

$\nabla \mathbf{m^r} \leftarrow \nabla \mathbf{\Theta_i} \leftarrow \nabla \mathbf{c}$, $\nabla \mathbf{\Theta_{i,j}} \leftarrow 0$ ▷back Eq. (3.7)
$\nabla \hat{\mathbf{m}}^\mathbf{r} \leftarrow \nabla \mathbf{m^r}$ ▷back message updates
for iteration $k \in \{K, ..., 1\}$ **do**
  $\nabla \mathbf{m^r} \leftarrow 0$ ▷zero-out
  forall directions $r \in \mathcal{R}$ **do** ▷**parallel**
    forall scanlines t in direction r **do** ▷**parallel**
      for node i in scanline t **do** ▷**sequential**
        $\lambda^* \leftarrow q^r_{k,i} \in \mathcal{L}$ ▷extract index
        $\nabla \hat{m}^r_i(\lambda^*) \mathrel{-}= \sum_{\lambda\in\mathcal{L}} \nabla \hat{m}^r_i(\lambda)$ ▷back Eq. (3.5)
        for label $\lambda \in \mathcal{L}$ **do**
          $\mu^* \leftarrow p^r_{k,i}(\lambda) \in \mathcal{L}$ ▷extract index
          $\nabla \theta_{i-r}(\mu^*) \mathrel{+}= \nabla \hat{m}^r_i(\lambda)$ ▷back Eq. (3.9)
          $\nabla \hat{m}^r_{i-r}(\mu^*) \mathrel{+}= \nabla \hat{m}^r_i(\lambda)$
          $\nabla m^d_{i-r}(\mu^*) \mathrel{+}= \nabla \hat{m}^r_i(\lambda), \forall d \in \mathcal{R} \setminus \{r, r^-\}$
          $\nabla \theta_{i-r,i}(\mu^*, \lambda) \mathrel{+}= \nabla \hat{m}^r_i(\lambda)$
    $\nabla \hat{\mathbf{m}}^r \leftarrow 0$ ▷zero-out
  $\nabla \mathbf{m^r} \mathrel{+}= \nabla \hat{\mathbf{m}}^\mathbf{r}$ ▷gather history gradients
  $\nabla \hat{\mathbf{m}}^\mathbf{r} \leftarrow \nabla \mathbf{m^r}$ ▷back message updates after iteration

Since ISGMR and TRWP use min-sum message passing, no exponent and logarithm are required. Only indices in message minimization and reparametrization are stored in two unsigned 8-bit integer tensors, denoted as $\{p^r_{k,i}(\lambda)\}$ and $\{q^r_{k,i}\}$ with indices of direction r, iteration k, node i, and label λ. This makes the backpropagation time less than 50% of the forward propagation time. In Fig. 3.2(a), the gradient updates in backpropagation are performed along edges that have the minimum messages in the forward direction. In Fig. 3.2(b), a message gradient at node i is accumulated from all following nodes after i from all backpropagation directions. Below, we denote the gradient of a variable $*$ from loss L as $\nabla * = dL/d*$.

For ISGMR at kth iteration, the gradients of the model parameters in Eq. (3.9) are

$$\begin{aligned}\nabla\theta_i(\lambda) =& \nabla c_i(\lambda) + \sum_{v\in\mathcal{L}} \sum_{r\in\mathcal{R}} \sum_{\mu\in\mathcal{L}} \Big(\nabla m^{r,k+1}_{i+2r}(\mu)\Big|_{v=p^r_{k,i+2r}(\mu)} \\ &+ \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} \nabla m^{d,k}_{i+r+d}(\mu)\Big|_{v=p^d_{k,i+r+d}(\mu)} \Big)\Big|_{\lambda=p^r_{k,i+r}(v)},\end{aligned} \tag{3.11}$$

$$\nabla\theta_{i-r,i}(\mu,\lambda) = \nabla m^{r,k+1}_i(\lambda)\Big|_{\mu=p^r_{k,i}(\lambda)}. \tag{3.12}$$

Algorithm 4: Backpropagation of TRWP

Input: Partial energy parameters $\{\theta_{i,j}\}$, gradients of final costs $\nabla\mathbf{c} = \{\nabla c_i(\lambda)\}$, tree decomposition coefficients $\{\rho_{i,j}\}$, set of nodes $\mathcal{V}$, edges $\mathcal{E}$, directions $\mathcal{R}$, indices $\{p^r_{k,i}(\lambda)\}$, $\{q^r_{k,i}\}$, iteration number K.

Output: Gradients $\{\nabla\theta_i, \nabla\theta_{i,j}(\cdot,\cdot)\}$.

1 $\nabla\mathbf{m^r} \leftarrow \nabla\mathbf{\Theta_i} \leftarrow d\mathbf{c}$, $d\mathbf{\Theta_{i,j}} \leftarrow 0$ ▷back Eq. (3.7)
2 **for** iteration $k \in \{K, ..., 1\}$ **do**
3 **for** direction $r \in \mathcal{R}$ **do** ▷**sequential**
4 **forall** scanlines t in direction r **do** ▷**parallel**
5 **for** node i in scanline t **do** ▷**sequential**
6 $\lambda^* \leftarrow q^r_{k,i} \in \mathcal{L}$ ▷extract index
7 $\nabla m^r_i(\lambda^*) \mathrel{-}= \sum_{\lambda\in\mathcal{L}} \nabla m^r_i(\lambda)$ ▷back Eq. (3.5)
8 **for** label $\lambda \in \mathcal{L}$ **do**
9 $\mu^* \leftarrow p^r_{k,i}(\lambda) \in \mathcal{L}$ ▷extract index
10 $\nabla\theta_{i-r}(\mu^*) \mathrel{+}= \rho_{i-r,i}\nabla m^r_i(\lambda)$ ▷back Eq. (3.10)
11 $\nabla m^d_{i-r}(\mu^*) \mathrel{+}= \rho_{i-r,i}\nabla m^r_i(\lambda), \forall d \in \mathcal{R}$
12 $\nabla m^{r^-}_{i-r}(\mu^*) \mathrel{-}= \nabla m^r_i(\lambda)$
13 $\nabla\theta_{i-r,i}(\mu^*, \lambda) \mathrel{+}= \nabla m^r_i(\lambda)$
14 $\nabla\mathbf{m}^r \leftarrow 0$ ▷zero-out

Importantly, within an iteration in ISGMR, $\nabla\mathbf{m}^{\mathbf{r},k}$ are updated but do not affect $\nabla\mathbf{m}^{\mathbf{r},k+1}$ until the backpropagation along all directions $\mathbf{r}$ is executed (line 18 in Algorithm 3. This is because within kth iteration, independently updated $\mathbf{m}^{r,k+1}$ in r will not affect $\mathbf{m}^{d,k}, \forall d \in \mathcal{R} \setminus \{r, r^-\}$, until the next iteration (line 12 in Algorithm 1).

In contrast, message gradients in TRWP from a direction will affect messages from other directions since, within an iteration in the forward propagation, message updates are *direction-by-direction*. For TRWP at kth iteration, $\nabla\theta_i(\lambda)$ is

$$\begin{aligned}\nabla\theta_i(\lambda) = \nabla c_i(\lambda) + \sum_{v\in\mathcal{L}}\sum_{r\in\mathcal{R}}\sum_{\mu\in\mathcal{L}}\Big(-\nabla m^{r^-}_i(\mu)\Big|_{v=p^{r^-}_{k,i}(\mu)} \\ + \sum_{d\in\mathcal{R}} \rho_{i+r,i+r+d}\nabla m^d_{i+r+d}(\mu)\Big|_{v=p^d_{k,i+r+d}(\mu)}\Big)\Big|_{\lambda=p^r_{k,i+r}(v)},\end{aligned} \tag{3.13}$$

where coefficient $\rho_{i+r,i+r+d}$ is for the edge connecting node $i+r$ and its next one in direction d which is denoted as node $i+r+d$, and the calculation of $\nabla\theta_{i-r,i}(\lambda,\mu)$ is in the same manner as Eq. (3.12) by replacing $m^{r,k+1}$ with m^r.

The backpropagation of TRWP can be derived similarly to ISGMR. We must know that gradients of the unary potentials and the pairwise potentials are accumulated along the opposite direction of the forward scanning direction. Therefore, an updated message is, in fact, a new variable, and its gradient should not be accumulated by its previous value but set to 0. This is extremely important, especially in ISGMR. It requires the message gradients to be accumulated and assigned in every iteration (lines 17-18 in Algorithm 3) and be zero-out (lines 4 and 16 in Algorithm 3 and line 14 in Algorithm 4). Gradient derivations of ISGMR and attributes are provided below.

3.5 Gradients Derivations in ISGMR

In the past, the unary terms generally consist of specific features obtained by feature extraction algorithms. This limits their effects on the inference. Due to many learning methods, however, the unary terms and pairwise terms can be updated in iterative optimization. Hence, for end-to-end learning by using our inference methods, we need to differentiate our inference algorithms with detailed gradient derivation.

In the following, we replace $m^{r,k}$ by m^r and $m^{r,k+1}$ by $\hat{m}^r$ for simplicity. This is because from the practical implementation, messages in direction r should be updated instead of allocating new memories in each iteration to avoid GPU memory increase. Thus, we only use two variables m^r and $\hat{m}^r$ for messages before and after an iteration.

3.5.1 Explicit Representation of Forward Propagation

Since message update in ISGMR relies on recursively updated messages $\hat{\mathbf{m}}^r$ in each scanning direction r and messages $\mathbf{m}^r$ from all the other directions updated in the previous iteration, an explicit ISGMR message update is

$$\begin{aligned}\hat{m}_i^r(\lambda) = \min_{\mu\in\mathcal{L}} \Big(\theta_{i-r}(\mu) + \theta_{i-r,i}(\mu,\lambda) + \hat{m}_{i-r}^r(\mu) + \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} m_{i-r}^d(\mu)\Big) \ , \\ \forall i \in \mathcal{V}, \forall \lambda \in \mathcal{L}, \forall r \in \mathcal{R} \ .\end{aligned} \tag{3.14}$$

Applying message reparametrization by

$$\hat{m}_i^r(\lambda) = \hat{m}_i^r(\lambda) - \min_{k\in\mathcal{L}} \hat{m}_i^r(k), \quad \forall i \in \mathcal{V}, \forall \lambda \in \mathcal{L}, \forall r \in \mathcal{R} \ . \tag{3.15}$$

After updating messages in **all directions within an iteration**, we assign the updated message $\hat{\mathbf{m}}$ to $\mathbf{m}$ by

$$m_i(\lambda) = \hat{m}_i(\lambda), \quad \forall i \in \mathcal{V}, \forall \lambda \in \mathcal{L} \ . \tag{3.16}$$

Eventually, **after all iterations**, unary potentials and updated messages from all directions will be aggregated by

$$c_i(\lambda) = \theta_i(\lambda) + \sum_{d\in\mathcal{R}} m_i^d(\lambda), \quad \forall i \in \mathcal{V}, \forall \lambda \in \mathcal{L} \ . \tag{3.17}$$

Different from optimization with winner-takes-all for labelling in learning by $\mathbf{x}_i = \text{argmin}_{\lambda\in\mathcal{L}} c_i(\lambda), \forall i \in \mathcal{V}$, a regression with disparity confidences calculated by the final costs is used to fit with the real-valued ground truth disparities $\mathbf{g} = \{g_i\}, \forall i \in \mathcal{V}$. Generally, the disparity confidence $f_i(\lambda)$ with a normalization such as SoftMin() is

$$f_i(\lambda) = \text{SoftMin}(c_i(\lambda)), \quad \forall i \in \mathcal{V}, \forall \lambda \in \mathcal{L} \ , \tag{3.18}$$

and the regression for real-valued disparity $\mathbf{d} = \{d_i\}, \quad \forall i \in \mathcal{V}$ is

$$d_i = \sum_{\lambda \in \mathcal{L}} \lambda f_i(\lambda), \quad \forall i \in \mathcal{V} \,. \tag{3.19}$$

The loss function $L(\mathbf{d}, \mathbf{g})$ can be standard L_1 or smooth L_1 loss function.

3.5.2 Derivations of Differentiability

Now we do backpropagation at kth iteration for learnable parameters $\{\theta_i, \theta_{i,j}\}$. With the same notations in Sec. 3.4, $\{p^r_{k,i}(\lambda)\}$ and $\{q^r_{k,i}\}$ are indices stored in the forward propagation from message minimization and reparameterization respectively, and $\nabla * = dL/d*$.

3.5.2.1 Gradients of Unary Potentials

Proposition: Gradients of unary potentials $\{\theta_i(\lambda)\}$ are represented by

$$\begin{aligned} \nabla\theta_i(\lambda) &= \nabla c_i(\lambda) + \sum_{v\in\mathcal{L}} \sum_{r\in\mathcal{R}} \nabla \hat{m}^r_{i+r}(v)\big|_{\lambda = p^r_{k,i+r}(v)} \\ &= \nabla c_i(\lambda) + \sum_{v\in\mathcal{L}} \sum_{r\in\mathcal{R}} \sum_{\mu\in\mathcal{L}} \Big(\nabla \hat{m}^r_{i+2r}(\mu)\big|_{v = p^r_{k,i+2r}(\mu)} \\ &\qquad + \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} \nabla m^d_{i+r+d}(\mu)\Big|_{v = p^d_{k,i+r+d}(\mu)} \Big)\Big|_{\lambda = p^r_{k,i+r}(v)} \,. \end{aligned} \tag{3.20}$$

Derivation:

The backpropagation from Eq. (3.19)-Eq. (3.14) is

$$\begin{aligned} \nabla\theta_i(\lambda) &= \frac{dL}{d\theta_i(\lambda)} \\ &= \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \frac{\partial L}{\partial d_j(v)} \frac{\partial d_j(v)}{\partial f_j(v)} \frac{\partial f_j(v)}{\partial c_j(v)} \frac{\partial c_j(v)}{\partial \theta_i(\lambda)} && \triangleright\text{back Eq. (3.19)-Eq. (3.18)} \\ &= \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \nabla c_j(v) \Big(\frac{\partial c_j(v)}{\partial \theta_j(v)} \frac{\partial \theta_j(v)}{\partial \theta_i(\lambda)} + \sum_{r\in\mathcal{R}} \frac{\partial c_j(v)}{\partial m^r_j(v)} \frac{\partial m^r_j(v)}{\partial \theta_i(\lambda)} \Big) && \triangleright\text{back Eq. (3.17)} \\ &= \nabla c_i(\lambda) + \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \sum_{r\in\mathcal{R}} \nabla m^r_j(v) \frac{\partial m^r_j(v)}{\partial \theta_i(\lambda)} \\ &= \nabla c_i(\lambda) + \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \sum_{r\in\mathcal{R}} \nabla m^r_j(v) \frac{\partial m^r_j(v)}{\partial \hat{m}^r_j(v)} \frac{\partial \hat{m}^r_j(v)}{\partial \theta_i(\lambda)} && \triangleright\text{back Eq. (3.16)} \\ &= \nabla c_i(\lambda) + \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \sum_{r\in\mathcal{R}} \nabla \hat{m}^r_j(v) \frac{\partial \hat{m}^r_j(v)}{\partial \theta_i(\lambda)} \,. \end{aligned} \tag{3.21}$$

With backpropagation of Eq. (3.15) using an implicit message reparametrization with index $v^* = q^r_{k,j}$ at kth iteration, $\nabla \hat{m}^r_j(v)$ in the second term above is updated by

$$\nabla \hat{m}^r_j(v) \leftarrow \begin{cases} \nabla \hat{m}^r_j(v) & \text{if } v \neq v^* \,, \\ -\sum_{v' \in \mathcal{L} \setminus v^*} \nabla \hat{m}^r_j(v') & \text{otherwise} \,. \end{cases} \tag{3.22}$$

Derivation of Eq. (3.22):

Explicit representation of Eq. (3.15) is $\tilde{m}^r_i(\lambda) = \hat{m}^r_i(\lambda) - \hat{m}^r_i(\lambda^*)$, where $\lambda^* = q^r_{k,i}$, then we have

$$\begin{aligned} \nabla \hat{m}^r_i(\lambda) &= \frac{\partial L}{\partial \hat{m}^r_i(\lambda)} \\ &= \sum_{i' \in \mathcal{V}} \sum_{\lambda' \in \mathcal{L}} \frac{\partial L}{\partial \tilde{m}^r_{i'}(\lambda')} \frac{\partial \tilde{m}^r_{i'}(\lambda')}{\partial \hat{m}^r_i(\lambda)} \\ &= \sum_{i' \in \mathcal{V}} \sum_{\lambda' \in \mathcal{L}} \frac{\partial L}{\partial \tilde{m}^r_{i'}(\lambda')} \Big(\frac{\partial \tilde{m}^r_{i'}(\lambda')}{\partial \hat{m}^r_{i'}(\lambda')} \frac{\partial \hat{m}^r_{i'}(\lambda')}{\partial \hat{m}^r_i(\lambda)} + \frac{\partial \tilde{m}^r_{i'}(\lambda')}{\partial \hat{m}^r_{i'}(\lambda^*)} \frac{\partial \hat{m}^r_{i'}(\lambda^*)}{\partial \hat{m}^r_i(\lambda)} \Big) \\ &= \frac{\partial L}{\partial \tilde{m}^r_i(\lambda)} - \sum_{\lambda' \in \mathcal{L}} \frac{\partial L}{\partial \tilde{m}_i(\lambda')} \Big|_{\lambda = \lambda^*} \\ &= \begin{cases} \nabla \tilde{m}^r_i(\lambda) & \text{if } \lambda \neq \lambda^* \,, \\ -\sum_{\lambda' \in \mathcal{L} \setminus \lambda^*} \nabla \tilde{m}^r_i(\lambda') & \text{otherwise} \,. \end{cases} \end{aligned} \tag{3.23}$$

Back to the implicit message reparametrization with $\nabla \tilde{m}^r$ replaced by $\nabla \hat{m}^r$, we have

$$\nabla \hat{m}^r_i(\lambda) = \begin{cases} \nabla \hat{m}^r_i(\lambda) & \text{if } \lambda \neq \lambda^* \,, \\ -\sum_{\lambda' \in \mathcal{L} \setminus \lambda^*} \nabla \hat{m}^r_i(\lambda') & \text{otherwise} \,. \end{cases} \tag{3.24}$$

End of the derivation of Eq. (3.22).

Next, we continue the backpropagation through Eq. (3.14) for unary potentials as

$$\begin{aligned} \nabla \theta_i(\lambda) &= \nabla c_i(\lambda) + \sum_{j \in \mathcal{V}} \sum_{v \in \mathcal{L}} \sum_{r \in \mathcal{R}} \nabla \hat{m}^r_j(v) \frac{\partial \hat{m}^r_j(v)}{\partial \theta_i(\lambda)} && \triangleright \text{from Eq. (3.21)} \\ &= \nabla c_i(\lambda) + \sum_{v \in \mathcal{L}} \sum_{r \in \mathcal{R}} \nabla \hat{m}^r_{i+r}(v) \frac{\partial \hat{m}^r_{i+r}(v)}{\partial \theta_i(\lambda)} && \triangleright \text{back Eq. (3.14) without recursion} \\ &= \nabla c_i(\lambda) + \sum_{v \in \mathcal{L}} \sum_{r \in \mathcal{R}} \nabla \hat{m}^r_{i+r}(v) \Big|_{\lambda = p^r_{k,i+r}(v)} \,. && \triangleright \text{satisfy argmin() rule in Eq. (3.14)} \end{aligned} \tag{3.25}$$

Derivation of $\nabla c_i(\lambda)$ by backpropagation from the loss function, disparity regression, and SoftMin(), can be obtained by PyTorch autograd directly. For the readability of derivations by avoiding using $\{m^r_{i+r}(m^r_i(\theta_{i-r}(\lambda))), m^r_{i+2r}(m^r_{i+r}(m^r_i(\theta_{i-r}(\lambda)))), ...\}$, we do not write the recursion of gradients in the derivations. Below, we derive

$\nabla \hat{m}^r_{i+r}(v)$ in the backpropagation.

3.5.2.2 Gradients of Messages

For notation readability, we first derive message gradient $\nabla \hat{m}^r_i(\lambda)$ instead of $\nabla \hat{m}^r_{i+r}(v)$.

Proposition: Gradients of messages $\{\hat{m}^r_i(\lambda)\}$ are represented by

$$\nabla \hat{m}^r_i(\lambda) = \sum_{v\in\mathcal{L}} \Big(\nabla \hat{m}^r_{i+r}(v)\big|_{\lambda=p^r_{k,i+r}(v)} + \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} \nabla m^d_{i+d}(v)\Big|_{\lambda=p^d_{k,i+d}(v)} \Big) \,. \tag{3.26}$$

Derivation:

$$\begin{aligned}
\nabla \hat{m}^r_i(\lambda) &= \frac{dL}{d\hat{m}^r_i(\lambda)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v) \frac{\partial c_j(v)}{\partial \hat{m}^r_i(\lambda)} && \triangleright \text{back Eq. (3.19)-Eq. (3.18)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v) \sum_{d\in\mathcal{R}} \frac{\partial c_j(v)}{\partial m^d_j(v)} \frac{\partial m^d_j(v)}{\partial \hat{m}^r_i(\lambda)} && \triangleright \text{back Eq. (3.17)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v) \sum_{d\in\mathcal{R}} \frac{\partial c_j(v)}{\partial m^d_j(v)} \frac{\partial m^d_j(v)}{\partial \hat{m}^d_j(v)} \frac{\partial \hat{m}^d_j(v)}{\partial \hat{m}^r_i(\lambda)} && \triangleright \text{back Eq. (3.16)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}^d_j(v) \frac{\partial \hat{m}^d_j(v)}{\partial \hat{m}^r_i(\lambda)} \,,
\end{aligned} \tag{3.27}$$

then we update $\nabla \hat{m}^d_j(v)$ by Eq. (3.22) and continue as follows,

$$\begin{aligned}
\nabla \hat{m}^r_i(\lambda) &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}^d_j(v) \frac{\partial \hat{m}^d_j(v)}{\partial \hat{m}^r_i(\lambda)} && \triangleright \text{from Eq. (3.27)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}^d_j(v) \Big(\sum_{\lambda'\in\mathcal{L}} \frac{\partial \hat{m}^d_j(v)}{\partial \hat{m}^d_{j-d}(\lambda')} \frac{\partial \hat{m}^d_{j-d}(\lambda')}{\partial \hat{m}^r_i(\lambda)} \\
&\quad + \sum_{d'\in\mathcal{R}\setminus\{d,d^-\}} \sum_{\lambda'\in\mathcal{L}} \frac{\partial \hat{m}^d_j(v)}{\partial m^{d'}_{j-d}(\lambda')} \frac{\partial m^{d'}_{j-d}(\lambda')}{\partial \hat{m}^r_i(\lambda)} \Big) && \triangleright \text{back Eq. (3.14)} \\
&= \sum_{v\in\mathcal{L}} \Big(\nabla \hat{m}^r_{i+r}(v)\big|_{\lambda=p^r_{k,i+r}(v)} \\
&\quad + \sum_{d\in\mathcal{R}} \sum_{d'\in\mathcal{R}\setminus\{d,d^-\}} \sum_{\lambda'\in\mathcal{L}} \nabla \hat{m}^d_j(v) \frac{\partial \hat{m}^d_j(v)}{\partial m^{d'}_{j-d}(\lambda')} \frac{\partial m^{d'}_{j-d}(\lambda')}{\partial \hat{m}^r_i(\lambda)} \Big) \,.
\end{aligned} \tag{3.28}$$

Since $m^{d'}_{j-d}(\lambda')$ is differentiable by $\hat{m}^r_i(\lambda)$ due to Eq. (3.16) and, for ISGMR, message

gradients in directions except the current direction r come from the next iteration (since in the forward propagation these messages come from the previous iteration), we have

$$\begin{aligned}
\nabla \hat{m}_i^r(\lambda) &= \sum_{v\in\mathcal{L}} \Big(\nabla \hat{m}_{i+r}^r(v)\big|_{\lambda=p_{k,i+r}^r(v)} \\
&\quad + \sum_{d\in\mathcal{R}} \sum_{d'\in\mathcal{R}\setminus\{d,d^-\}} \sum_{\lambda'\in\mathcal{L}} \nabla \hat{m}_j^d(v) \frac{\partial \hat{m}_j^d(v)}{\partial m_{j-d}^{d'}(\lambda')} \frac{\partial m_{j-d}^{d'}(\lambda')}{\partial \hat{m}_i^r(\lambda)} \Big) && \triangleright \text{from Eq. (3.28)} \\
&= \sum_{v\in\mathcal{L}} \Big(\nabla \hat{m}_{i+r}^r(v)\big|_{\lambda=p_{k,i+r}^r(v)} \\
&\quad + \sum_{d\in\mathcal{R}} \nabla \hat{m}_{i+d}^d(v) \frac{\partial \hat{m}_{i+d}^d(v)}{\partial m_i^r(\lambda)} \frac{\partial m_i^r(\lambda)}{\partial \hat{m}_i^r(\lambda)}\Big|_{r\notin\{d,d^-\}} \Big) && \triangleright \text{due to Eq. (3.16)} \\
&= \sum_{v\in\mathcal{L}} \Big(\nabla \hat{m}_{i+r}^r(v)\big|_{\lambda=p_{k,i+r}^r(v)} \\
&\quad + \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} \nabla m_{i+d}^d(v)\Big|_{\lambda=p_{k,i+d}^d(v)} \Big) .
\end{aligned} \tag{3.29}$$

Here, updating the message gradient at node i depends on its next node $i+r$ along the scanning direction r; this scanning direction is opposite to the forward scanning direction, and thus, it depends on node $i+r$ instead of $i-r$. The gradient of message $m_i^r(\lambda)$ can be derived in the same way.

Now one can derive $\nabla \hat{m}_{i+r}^r(v)$ in the same manner of $\nabla \hat{m}_i^r(\lambda)$ and apply it to Eq. (3.25) to obtain Eq. (3.20).

3.5.2.3 Gradients of Pairwise Potentials

Proposition: Gradients of pairwise potentials $\{\theta_{i-r,i}(\mu,\lambda)\}$ are represented by

$$\nabla \theta_{i-r,i}(\mu,\lambda) = \nabla \hat{m}_i^r(\lambda)\big|_{\mu=p_{k,i}^r(\lambda)}, \quad \forall i\in\mathcal{V}, \forall r\in\mathcal{R}, \forall \lambda,\mu\in\mathcal{L} . \tag{3.30}$$

Derivation:

$$\begin{aligned}
\nabla \theta_{i-r,i}(\mu,\lambda) &= \frac{dL}{d\theta_{i-r,i}(\mu,\lambda)} \\
&= \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \nabla c_j(v) \frac{\partial c_j(v)}{\partial \theta_{i-r,i}(\mu,\lambda)} && \triangleright \text{back Eq. (3.19)-Eq. (3.18)} \\
&= \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \nabla c_j(v) \sum_{d\in\mathcal{R}} \frac{\partial c_j(v)}{\partial m_j^d(v)} \frac{\partial m_j^d(v)}{\partial \theta_{i-r,i}(\mu,\lambda)} && \triangleright \text{back Eq. (3.17)} \\
&= \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \sum_{d\in\mathcal{R}} \nabla c_j(v) \frac{\partial c_j(v)}{\partial m_j^d(v)} \frac{\partial m_j^d(v)}{\partial \hat{m}_j^d(v)} \frac{\partial \hat{m}_j^d(v)}{\partial \theta_{i-r,i}(\mu,\lambda)} && \triangleright \text{back Eq. (3.16)} \\
&= \sum_{j\in\mathcal{V}} \sum_{v\in\mathcal{L}} \sum_{d\in\mathcal{R}} \nabla \hat{m}_j^d(v) \frac{\partial \hat{m}_j^d(v)}{\partial \theta_{i-r,i}(\mu,\lambda)} .
\end{aligned} \tag{3.31}$$

Now we update $\nabla \hat{m}_j^d(v)$ by Eq. (3.22). Then

$$\begin{aligned}\nabla\theta_{i-r,i}(\mu,\lambda) &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}_j^d(v)\frac{\partial \hat{m}_j^d(v)}{\partial \theta_{i-r,i}(\mu,\lambda)} && \triangleright\text{from Eq. (3.31)}\\ &= \nabla \hat{m}_i^r(\lambda)\big|_{\mu=p_{k,i}^r(\lambda)} \ . && \triangleright\text{back Eq. (3.14) without recursion}\end{aligned} \tag{3.32}$$

One can note that the memory requirement of $\{\theta_{i-r,i}(\mu,\lambda)\}$ is $4\sum_{r\in\mathcal{R}}|\mathcal{E}^r||\mathcal{L}||\mathcal{L}|$ bytes using single-precision floating-point values. This will be high when the number of disparities $|\mathcal{L}|$ is large. In practical, since the pairwise potentials can be decomposed by $\theta_{i,j}(\lambda,\mu) = \theta_{i,j}V(\lambda,\mu), \forall(i,j)\in\mathcal{E}, \forall\lambda,\mu\in\mathcal{L}$ with edge weights $\theta_{i,j}$ and a pairwise function $V(\cdot,\cdot)$, it takes up $4(\sum_{r\in\mathcal{R}}|\mathcal{E}^r| + |\mathcal{L}||\mathcal{L}|)$ bytes in total, which is much less than $4\sum_{r\in\mathcal{R}}|\mathcal{E}^r||\mathcal{L}||\mathcal{L}|$ above. Therefore, we additionally provide the gradient derivations of these two terms, edge weights and pairwise functions, for practical implementations of the backpropagation.

3.5.2.4 Gradients of Edge Weights

Proposition: Gradients of edge weights $\{\theta_{i-r,i}\}$ are represented by

$$\nabla\theta_{i-r,i} = \sum_{v\in\mathcal{L}} \nabla \hat{m}_i^r(v) V(p_{k,i}^r(v), v), \quad \forall i\in\mathcal{V}, \forall r\in\mathcal{R} \ . \tag{3.33}$$

Derivation:

$$\begin{aligned}\nabla\theta_{i-r,i} = \frac{dL}{d\theta_{i-r,i}} &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v)\frac{\partial c_j(v)}{\partial \theta_{i-r,i}} && \triangleright\text{back Eq. (3.19)-Eq. (3.18)}\\ &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v)\sum_{d\in\mathcal{R}}\frac{\partial c_j(v)}{\partial m_j^d(v)}\frac{\partial m_j^d(v)}{\partial \theta_{i-r,i}} && \triangleright\text{back Eq. (3.17)}\\ &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla c_j(v)\frac{\partial c_j(v)}{\partial m_j^d(v)}\frac{\partial m_j^d(v)}{\partial \hat{m}_j^d(v)}\frac{\partial \hat{m}_j^d(v)}{\partial \theta_{i-r,i}} && \triangleright\text{back Eq. (3.16)}\\ &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}_j^d(v)\frac{\partial \hat{m}_j^d(v)}{\partial \theta_{i-r,i}} \ .\end{aligned} \tag{3.34}$$

Again, before updating gradients of edge weights by Eq. (3.14), $\nabla \hat{m}_j^d(v)$ is updated by Eq. (3.22). Then

$$\begin{aligned}\nabla\theta_{i-r,i} &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}_j^d(v)\frac{\partial \hat{m}_j^d(v)}{\partial \theta_{i-r,i}} && \triangleright\text{from Eq. (3.34)}\\ &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{d\in\mathcal{R}} \nabla \hat{m}_j^d(v)\frac{\partial \hat{m}_j^d(v)}{\partial \theta_{j-d,j}}V(p_{k,j}^d(v), v)\frac{\partial \theta_{j-d,j}}{\partial \theta_{i-r,i}} && \triangleright\text{back Eq. (3.14), no recursion}\\ &= \sum_{v\in\mathcal{L}} \nabla \hat{m}_i^r(v) V(p_{k,i}^r(v), v) \ .\end{aligned} \tag{3.35}$$

In the case that when edge weights are undirected, *i.e.*, $\theta_{i,j} = \theta_{j,i}$, the derivations above still hold, and if $\theta_{i,j} = \theta_{j,i}$ are stored in the same tensor, $\nabla\theta_{i,j}$ will be accumulated by adding $\nabla\theta_{j,i}$ for storing the gradient of this edge weight. This is also applied to the gradient of pairwise potentials in Eq. (3.30) above.

3.5.2.5 Gradients of Pairwise Functions

Proposition: Gradients of a pairwise function $V(\cdot,\cdot)$ are

$$\nabla V(\lambda,\mu) = \sum_{j\in\mathcal{V}}\sum_{r\in\mathcal{R}} \theta_{j-r,j}\nabla\hat{m}_j^r(\mu)\Big|_{\lambda=p_{k,j}^r(\mu)}, \quad \forall\lambda,\mu\in\mathcal{L}\,. \tag{3.36}$$

Derivation:

$$\begin{aligned}
\nabla V(\lambda,\mu) &= \frac{dL}{dV(\lambda,\mu)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v)\frac{\partial c_j(v)}{\partial V(\lambda,\mu)} && \triangleright\text{back Eq. (3.19)-Eq. (3.18)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}} \nabla c_j(v)\sum_{r\in\mathcal{R}}\frac{\partial c_j(v)}{\partial m_j^r(v)}\frac{\partial m_j^r(v)}{\partial V(\lambda,\mu)} && \triangleright\text{back Eq. (3.17)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{r\in\mathcal{R}} \nabla c_j(v)\frac{\partial c_j(v)}{\partial m_j^r(v)}\frac{\partial m_j^r(v)}{\partial\hat{m}_j^r(v)}\frac{\partial\hat{m}_j^r(v)}{\partial V(\lambda,\mu)} && \triangleright\text{back Eq. (3.16)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{r\in\mathcal{R}} \nabla\hat{m}_j^r(v)\frac{\partial\hat{m}_j^r(v)}{\partial V(\lambda,\mu)}\,.
\end{aligned} \tag{3.37}$$

$\nabla\hat{m}_j^r(v)$ is updated by Eq. (3.22). Then

$$\begin{aligned}
\nabla V(\lambda,\mu) &= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{r\in\mathcal{R}} \nabla\hat{m}_j^r(v)\frac{\partial\hat{m}_j^r(v)}{\partial V(\lambda,\mu)} && \triangleright\text{from Eq. (3.37)} \\
&= \sum_{j\in\mathcal{V}}\sum_{v\in\mathcal{L}}\sum_{r\in\mathcal{R}} \nabla\hat{m}_j^r(v)\sum_{\lambda'\in\mathcal{L}}\frac{\partial\hat{m}_j^r(v)}{\partial V(\lambda',v)}\frac{\partial V(\lambda',v)}{\partial V(\lambda,\mu)} && \triangleright\text{from Eq. (3.14)} \\
&= \sum_{j\in\mathcal{V}}\sum_{r\in\mathcal{R}} \theta_{j-r,j}\nabla\hat{m}_j^r(\mu)\Big|_{\lambda=p_{k,j}^r(\mu)}\,.
\end{aligned} \tag{3.38}$$

3.5.3 Characteristics of Backpropagation

3.5.3.1 Accumulation

Since a message update usually has several components, its gradient is therefore accumulated when backpropagating through every component. For instance, in Eq. (3.25), the gradient of unary potential $\nabla\theta_i(\lambda)$ has $\nabla c_i(\lambda)$ and $\nabla\hat{m}_{i+r}^r(v), \forall r\in\mathcal{R}$ and $\forall v$ satisfying $\lambda = p_{k,i+r}^r(v)$ at kth iteration. It is calculated recursively but not at

once due to multiple nodes on a tree, multiple directions, and multiple iterations. In Eq. (3.29), the message gradient of a node relies on the gradient of all nodes after it in the forward propagation since this message will be used to all the message updates after this node.

3.5.3.2 Zero Out Gradients

Message gradients are not accumulated throughout the backpropagation but should be zeroed out in some cases. In more details, in the forward propagation, the repeated usage of $\mathbf{m}^{\mathbf{r}}$ and $\hat{\mathbf{m}}^{\mathbf{r}}$ is for all iterations but the messages are, in fact, new variables whenever they are updated. Since the gradient of a new message must be initialized to 0, zeroing out the gradients of the new messages is important. Specifically, in ISGMR that within an iteration $\mathbf{m}^{\mathbf{r}} \leftarrow \hat{\mathbf{m}}^{\mathbf{r}}$ is executed only when message updates in all directions are done. Thus, $\nabla\mathbf{m}^{\mathbf{r}}$ must be zeroed out after $\nabla\hat{\mathbf{m}}^{\mathbf{r}} \leftarrow \nabla\mathbf{m}^{\mathbf{r}}$. Similarly, after using $\nabla\hat{\mathbf{m}}^{\mathbf{r}}$ to update the gradients of learnable parameters and messages, $\nabla\hat{\mathbf{m}}^{\mathbf{r}} \leftarrow 0, \forall r \in \mathcal{R}$.

3.6 Experiments

Below, we evaluated the optimization capability of message passing algorithms on stereo vision and image denoising with fixed yet qualified data terms from benchmark settings. In addition, differentiability was evaluated by end-to-end learning for 21-class semantic segmentation. The experiments include effectiveness and efficiency studies of the message passing algorithms.

We implemented SGM, ISGMR, TRWP in C++ with single and multiple threads, PyTorch, and CUDA from scratch. PyTorch versions are for time comparison and gradient checking. For a fair comparison, we adopted benchmark code of TRWS from Szeliski et al. [2008] with general pairwise functions; MF followed Eq. (4) in Krähenbühl and Koltun [2011]. For iterative SGM, unary potentials were reparametrized by Eq. (3.6). OpenGM [Kappes et al., 2013] can be used for more comparisons in optimization noting TRWS as one of the most effective inference methods.

Our experiments were on 3.6GHz i7-7700 Intel(R) Core(TM) and Tesla P100 SXM2.

3.6.1 Optimization for Stereo Vision and Image Denoising

The capability of minimizing an energy function determines the significance of selected algorithms. We compared our ISGMR and TRWP with MF, SGM with single and multiple iterations, and TRWS. The evaluated energies are calculated with 4 connections.

3.6.1.1 Datasets

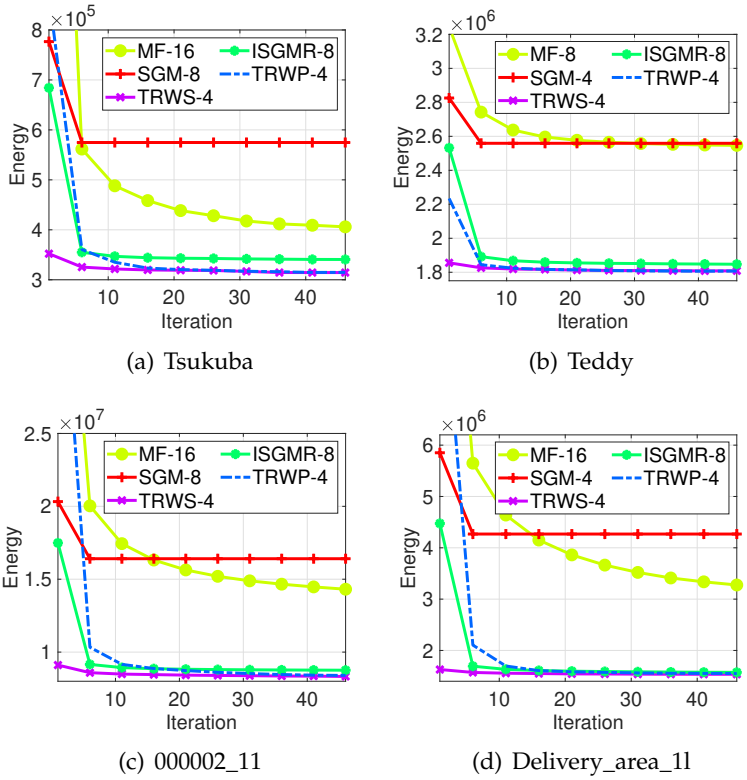
For stereo vision, we used Tsukuba, Teddy, Venus, Map, and Cones from Middlebury [Scharstein and Szeliski, 2002, 2003], 000041_10 and 000119_10 from KITTI2015

[Menze et al., 2018, 2015b], and delivery_area_1l and facade_1 from ETH3D two-view [Schops et al., 2017] for different types of stereo views. For image denoising, Penguin and House from Middlebury dataset [1] were used.

3.6.1.2 MRF Model Parameters

Model parameters include unary and pairwise potentials. In practice, the pairwise potentials consist of a pairwise function and edge weights, as $\theta_{i,j}(\lambda, \mu) = \theta_{i,j} V(\lambda, \mu)$. For the pairwise function $V(\cdot, \cdot)$, one can adopt (truncated) linear, (truncated) quadratic, Cauchy, Huber, *etc.*, [Hartley and Zisserman, 2003]. For the edge weights $\theta_{i,j}$, some methods apply a higher penalty on edge gradients under a given threshold. We set it as a constant for the comparison with SGM. Moreover, we adopted edge weights in Szeliski et al. [2008] and pairwise functions for Tsukuba, Teddy, and Venus, and Ajanthan et al. [2015] for Cones and Map; for the others, the pairwise function was linear and edges weights were 10.

3.6.1.3 Number of Directions Matters



(a) Tsukuba (b) Teddy

(c) 000002_11 (d) Delivery_area_1l

Figure 3.3: *Convergence with connections having the minimum energy in Table 3.1(a).*

[1] http://vision.middlebury.edu/MRF/results

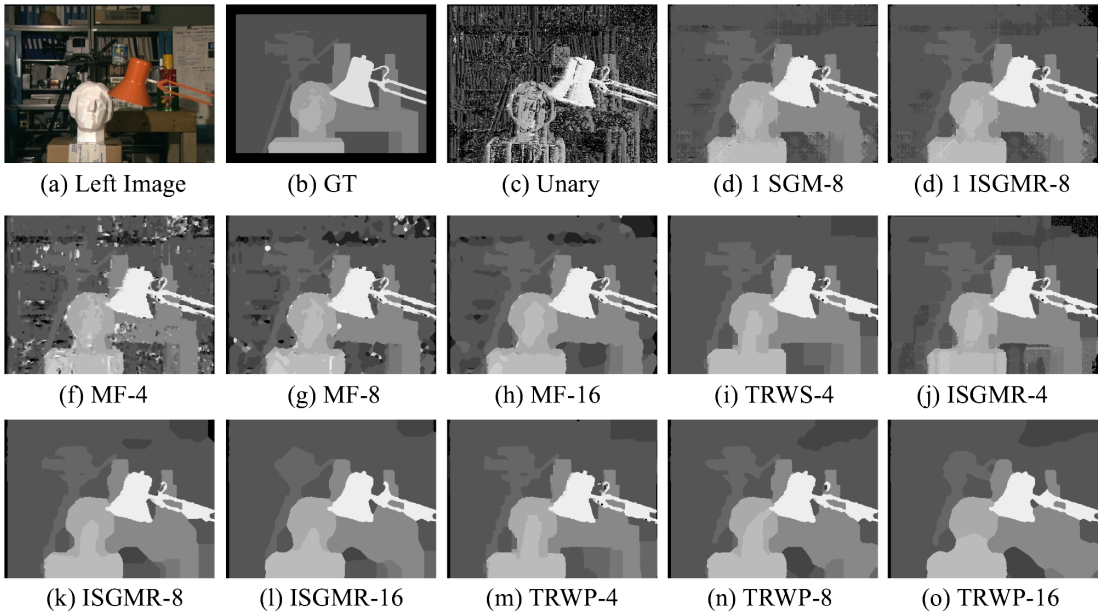


Figure 3.4: *Disparities of Tsukuba. (d)-(e) are at the 1st iteration. (f)-(o) are at the 50th iteration. (j) and (l) have the lowest energies in ISGMR-related and TRWP-related methods respectively. TRWP-4 and TRWS-4 have similar disparities for the most parts.*

In Fig. 3.3, ISGMR-8 and TRWP-4 outperform the others in ISGMR-related and TRWP-related methods in most cases. From the experiments, 4 directions are sufficient for TRWP, but for ISGMR energies with 8 directions are lower than those with 4 directions. This is because messages from 4 directions in ISGMR are insufficient to gather local information due to independent message updates in each direction. In contrast, messages from 4 directions in TRWP are highly updated in each direction and affected by those from the other directions. Note that in Eq. (3.7) messages from all directions are summed equally, this makes the labels by TRWP over-smooth within the connection area, for example, the camera is oversmooth "TRWP-8" in Fig. 3.4. Overall, TRWP-4 and ISGMR-8 are the best.

3.6.1.4 ISGMR versus SGM

Facciolo et al. [2015] demonstrates the decrease in energy of the over-count corrected SGM compared with the standard SGM. The result shows the improved optimization results achieved by subtracting unary potentials $(|\mathcal{R}| - 1)$ times. For experimental completion, we show both the decreased energies and improved disparity maps produced by ISGMR. From Tables 3.1(a)-3.3(a), SGM-related energies are much higher than ISGMR's because of the over-counted unary potentials. Moreover, ISGMR at the 50th iteration has much a lower energy value than the 1st iteration, indicating the importance of iterations, and is also much lower than those for MF and SGM at the 50th iteration.

3.6.1.5 TRWP versus TRWS

TRWP and TRWS have the same manner of updating messages and could have similar minimum energies. Generally, TRWS has the lowest energy; at the 50th iteration, however, TRWP-4 has lower energies, for instance, Tsukuba and Teddy in Table 3.1(a) and Penguin and House in Table 3.3(a). For TRWP, 50 iterations are sufficient to show its high optimization capability, as shown in Fig. 3.3.

Table 3.1: *Energy minimization on Middlebury dataset with constant edge weights. For Map, ISGMR-4 has the lowest energy among ISGMR-related methods; for others, ISGMR-8 and TRWP-4 have the lowest energies in ISGMR-related and TRWP-related methods respectively. ISGMR is more effective than SGM in the optimization, and TRWP-4 outperforms MF and SGM. ISGMR and TRWP outperform MF and SGM.*

(a) Energy with 4 Connections for All Methods

Method	Tsukuba		Teddy		Venus		Cones		Map	
	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter
MF-4	3121704	1620524	3206347	2583784	108494928	14618819	9686122	6379392	1116641	363221
SGM-4	873777	644840	2825535	2559016	5119933	2637164	3697880	3170715	255054	216713
TRWS-4	352178	314393	1855625	1807423	1325651	1219774	2415087	2329324	150853	143197
ISGMR-4 (ours)	824694	637996	2626648	1898641	4595032	1964032	3296594	2473646	215875	148049
TRWP-4 (ours)	869363	314037	2234163	1806990	32896024	1292619	3284868	2329343	192200	143364
MF-8	2322139	504815	3244710	2545226	68718520	2920117	7762269	3553975	840615	213827
SGM-8	776706	574758	2868131	2728682	4651016	2559933	3631020	3309643	243058	222678
ISGMR-8 (ours)	684185	340347	2532071	1847833	4062167	1285330	3039638	2398060	195718	149857
TRWP-8 (ours)	496727	348447	1981582	1849287	8736569	1347060	2654033	2396257	162432	151970
MF-16	1979155	404404	3315900	2622047	43077872	1981096	6741127	3062965	638753	204737
SGM-16	710727	587376	2907051	2846133	4081905	2720669	3564423	3413752	242932	232875
ISGMR-16 (ours)	591554	377427	2453592	1956343	3222851	1396914	2866149	2595487	190847	165249
TRWP-16 (ours)	402033	396036	1935791	1976839	2636413	1486880	2524566	2660964	162655	164704

(b) Energy with x Connections (x is the number of directions in "Method")

Method	Tsukuba		Teddy		Venus		Cones		Map	
	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter
MF-8	5153239	1076395	5669860	4189986	163694880	6093467	16168929	6539845	1859067	436475
SGM-8	1596446	1171778	4769341	4473302	9553166	5081133	6366360	5655313	490646	444662
ISGMR-8 (ours)	1349205	547347	3704341	2165853	8187317	2157980	4684708	3154010	332786	210569
TRWP-8 (ours)	998407	542407	2566672	2125447	19978228	2260610	3866053	3069597	252052	205110
MF-16	9331735	2012684	10895190	7814147	250118992	9608646	30523986	11037735	3099885	883725
SGM-16	3073027	2561056	8807121	8545043	18307864	12338719	12134483	11419232	1033776	987815
ISGMR-16 (ours)	2257414	1069727	5560332	2849363	13726551	4751164	7368559	4932437	572939	342205
TRWP-16 (ours)	1676753	1050736	3420761	2775229	14973863	4902130	5975966	4784404	420635	338740

3.6.2 End-to-End Learning for Semantic Segmentation

Although deep network and multi-scale strategy on CNN make semantic segmentation smooth and continuous on object regions, effective message passing inference on pairwise MRFs is beneficial for fine results with auxiliary edge information. The popular denseCRF [Krähenbühl and Koltun, 2011] demonstrated the effectiveness of using MF inference and the so-called dense connections; our experiments, however,

Table 3.2: *Energy minimization on 3 image pairs of KITTI2015 dataset and 2 of ETH-3D dataset with constant edge weights. ISGMR is more effective than SGM in the optimization in both single and multiple iterations, and TRWP-4 outperforms MF and SGM. ISGMR and TRWP outperform MF and SGM.*

(a) Energy with 4 Connections

Method	000002_11		000041_10		000119_10		Delivery_area_1l		Facade_1s	
	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter
MF-4	82523536	44410056	69894016	36163508	72659040	42392548	19945352	9013862	13299859	6681882
SGM-4	24343250	18060026	15926416	12141643	24999424	18595020	5851489	4267990	1797314	1429254
TRWS-4	9109976	8322635	6876291	6491169	10811576	9669367	1628879	1534961	891282	851273
ISGMR-4 (ours)	22259606	12659612	14434318	9984545	23180608	18541970	5282024	2212106	1572377	980151
TRWP-4 (ours)	40473776	8385450	30399548	6528642	36873904	9765540	9899787	1546795	2851700	854552
MF-8	61157072	18416536	53302252	16473121	57201868	21320892	16581587	4510834	10978978	3422296
SGM-8	20324684	16406781	13740635	11671740	20771096	16652122	5396353	4428411	1717285	1464208
ISGMR-8 (ours)	17489158	8753990	11802603	6639570	18411930	10173513	4474404	1571528	1438210	884241
TRWP-8 (ours)	18424062	8860552	13319964	6678844	20581640	10445172	4443931	1587917	1358270	889907
MF-16	46614232	14192750	40838292	12974839	44706364	16708809	13223338	3229021	9189592	2631006
SGM-16	18893122	16791762	13252150	12162330	19284684	16936852	5092094	4611821	1670997	1535778
ISGMR-16 (ours)	15455787	9556611	10731068	6806150	16608803	11037483	3689863	1594877	1324235	937102
TRWP-16 (ours)	11239113	9736704	8187380	6895937	13602307	11309673	2261402	1630973	1000985	950607

(b) Energy with x Connections (x is the number of directions in "Method")

Method	000002_11		000041_10		000119_10		Delivery_area_1l		Facade_1s	
	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter	1 iter	50 iter
MF-8	138483664	38453168	121249136	34120932	131181920	45884212	38093264	9572694	25024242	7448966
SGM-8	41237220	32364522	25991992	42714064	42714064	33276052	10748253	8638491	3278185	2736228
ISGMR-8 (ours)	32325396	11604810	20145410	8614330	35004328	14463813	8192084	2006168	2428810	1189081
TRWP-8 (ours)	36362752	11647502	25214586	8627464	42051436	14858602	9288541	2011627	2401460	1177447
MF-16	244990128	64062440	219675200	58400964	248971408	83290128	72133336	14976071	48832616	12981426
SGM-16	77678152	67940160	49492572	83162200	20541282	18319484	20541282	18319484	6463677	5865138
ISGMR-16 (ours)	51084200	17869246	32157608	14078210	58888264	23918132	12099823	3095897	4009805	1975452
TRWP-16 (ours)	36362752	11647502	24616778	14085857	54757076	26091176	8421742	3134283	2879525	1964657

illustrated that with local connections, superior inferences, such as TRWS, ISGMR, and TRWP, have a better convergence ability than MF and SGM to improve the performance.

Below, we adopted TRWP-4 and ISGMR-8 as our inference methods and negative logits from DeepLabV3+ [Chen et al., 2018b] as unary terms. Edge weights from Canny edges are in the form of $\theta_{ij} = 1 - |e_i - e_j|$, where e_i is a binary Canny edge value at node i. Potts model was used for pairwise function $V(\lambda, \mu)$. Since MF required much larger GPU memory than others due to its dense gradients, for practical purposes we used MF-4 for learning with the same batch size 12 within our GPU memory capacity.

3.6.2.1 Datasets

We used PASCAL VOC 2012 [Everingham et al., 2014] and Berkeley benchmark [Hariharan et al., 2011], with 1449 samples of the PASCAL VOC 2012 valset for validation and the other 10582 for training. These datasets identify 21 classes with 20 objects and 1 background.

Table 3.3: *Energy minimization for image denoising at the 50th iteration with* 4, 8, 16 *connections (all numbers divided by* 10^3*). Our ISGMR or TRWP performs best.*

(a) Energy with 4 Connections

Method	Penguin	House
MF-4	46808	50503
SGM-4	31204	66324
TRWS-4	15361	37572
ISGMR-4 (ours)	16514	37603
TRWP-4 (ours)	15358	37552
MF-8	21956	47831
SGM-8	37520	76079
ISGMR-8 (ours)	15899	39975
TRWP-8 (ours)	16130	40209
MF-16	20742	55513
SGM-16	47028	87457
ISGMR-16 (ours)	17035	46997
TRWP-16 (ours)	17516	47825

(b) Energy with x Connections (x is the number of directions in "Method")

Method	Penguin	House
MF-8	41071	85663
SGM-8	71715	146949
ISGMR-8 (ours)	21807	45054
TRWP-8 (ours)	21351	44999
MF-16	80286	169324
SGM-16	188323	369652
ISGMR-16 (ours)	37226	55954
TRWP-16 (ours)	36307	55777

3.6.2.2 CNN Learning Parameters

We trained the state-of-the-art DeepLabV3+ (ResNet101 as the backbone) with an initial learning rate 0.007, "poly" learning rate decay scheduler, and image size 512×512. Negative logits from DeepLabV3+ served as unary terms, the learning rate was decreased for learning message passing inference with 5 iterations, *i.e.*, 1e-4 for TRWP and SGM and 1e-6 for ISGMR and MF. Note that we experimented with all of these learning rates for involved inferences and selected the best for demonstration, for instance, for MF the accuracy by 1e-6 is much higher than the one by 1e-4.

Table 3.4: *Term weight for TRWP-4 on PASCAL VOC2012 val set.*

Method	λ	mIoU (%)
+TRWP-4	1	79.27
+TRWP-4	10	79.53
+TRWP-4	20	79.65
+TRWP-4	30	79.44
+TRWP-4	40	79.60

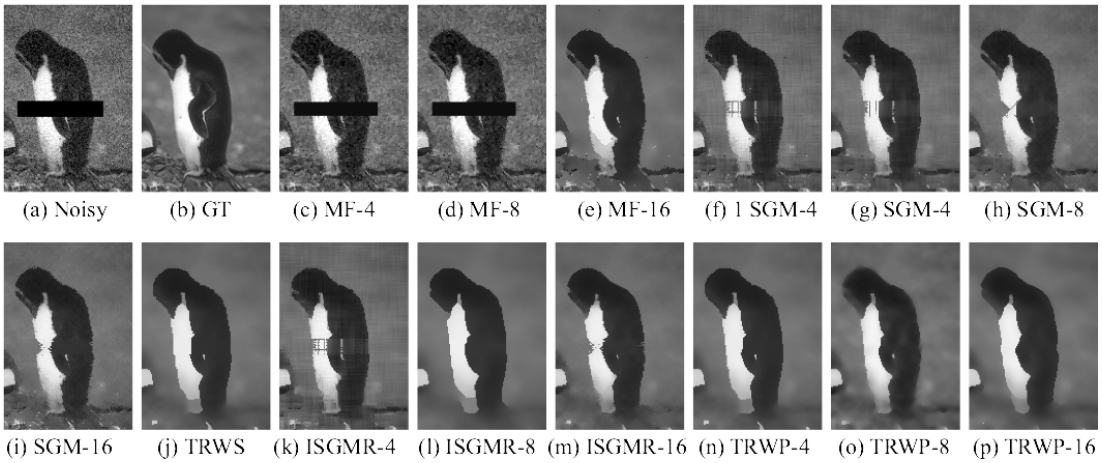


Figure 3.5: *Penguin denoising corresponding to the minimum energies in Table 3.3(a). ISGMR-8 and TRWP-4 are our proposals.*

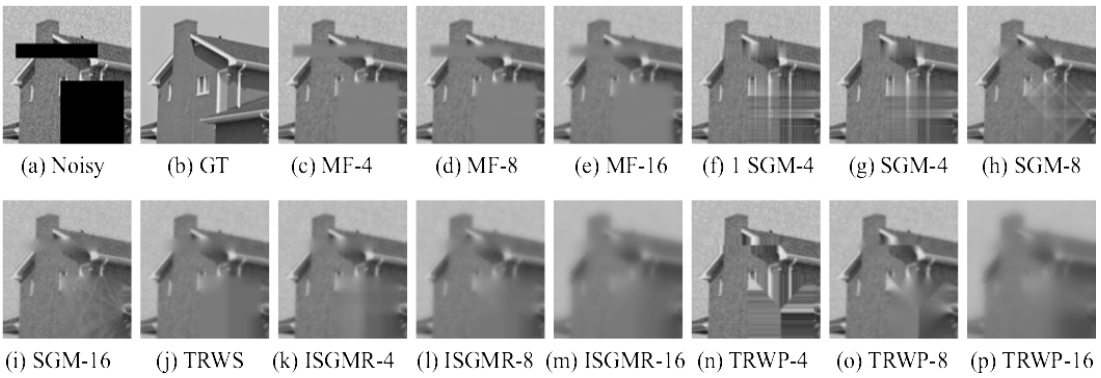
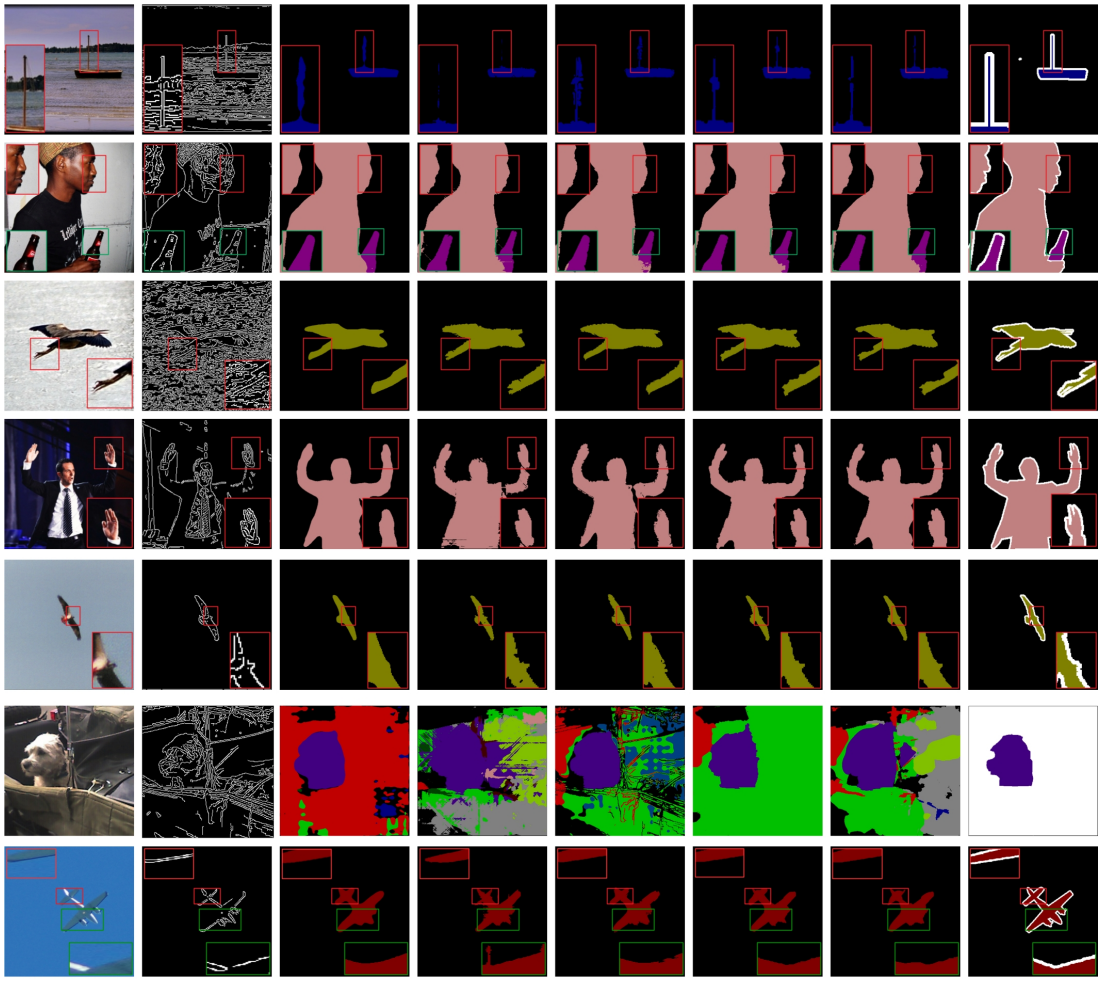


Figure 3.6: *House denoising corresponding to the minimum energies in Table 3.3(a). ISGMR-8 and TRWP-4 are our proposals.*

Table 3.5: *Full comparison on PASCAL VOC2012 val set.*

Method	λ	mIoU (%)
DeepLabV3+ [Chen et al., 2018b]	-	78.52
+SGM-8 [Hirschmuller, 2008]	5	78.94
+MF-4 [Krähenbühl and Koltun, 2011]	5	77.89
+ISGMR-8 (ours)	5	78.95
+TRWP-4 (ours)	20	79.65

In Table 3.4, ISGMR-8 and TRWP-4 outperform the baseline DeepLabV3+ [Chen et al., 2018b], SGM-8 [Hirschmuller, 2008], and MF-4 [Krähenbühl and Koltun, 2011]. Semantic segmentation by ISGMR-8 and TRWP-4 are more sharp, accurate, and aligned with the Canny edges and Ground Truth (GT) edges, shown in white, than the other inference methods, such as SGM-8 and MF-4 (see Table 3.5 and Fig. 3.7).



(a) RGB (b) Canny (c) baseline (d) SGM-8 (d) MF-4 (e) ISGMR-8 (f) TRWP-4 (g) GT

Figure 3.7: *Semantic segmentation on PASCAL VOC2012 valset. The last two rows are failure cases due to poor unary terms and missing edges. ISGMR-8 and TRWP-4 are ours.*

3.6.3 Speed Improvement

Speed-up by parallelized message passing on a GPU enables a fast inference and end-to-end learning. To be clear, we compared forward and backward propagation times for different implementations using 256×512 size images with 32 and 96 labels.

3.6.3.1 Forward Propagation Time

In Table 3.6, the forward propagation by CUDA implementation is the fastest. Our CUDA versions of ISGMR-8 and TRWP-4 are at least 24 and 7 times faster than PyTorch GPU versions at 32 and 96 labels respectively. In PyTorch GPU versions, we used tensor-wise tree parallelization to highly speed it up for a fair comparison. Obviously, GPU versions are much faster than CPU versions.

Table 3.6: *Forward propagation time with 32 and 96 labels. Our CUDA version is averaged over 1000 trials; others over 100 trials. Our CUDA version is 7–32 times faster than the PyTorch GPU version. The C++ versions are with a single and 8 threads. Unit: second.*

Method	PyTorch CPU		PyTorch GPU		C++ single		C++ multiple		CUDA (ours)		Speed-up PyT/CUDA	
	32	96	32	96	32	96	32	96	32	96	32	96
TRWS-4	-	-	-	-	1.95	13.30	-	-	-	-	-	-
ISGMR-4	1.43	11.70	0.96	1.13	3.23	25.19	0.88	5.28	0.03	0.15	32×	8×
ISGMR-8	3.18	24.78	1.59	1.98	8.25	71.35	2.12	15.90	0.07	0.27	23×	7×
ISGMR-16	7.89	52.76	2.34	4.96	30.76	273.68	7.70	62.72	0.13	0.53	18×	9×
TRWP-4	1.40	11.74	0.87	1.08	1.84	15.41	0.76	4.46	0.03	0.15	29×	7×
TRWP-8	3.19	24.28	1.57	1.98	6.34	57.25	1.88	14.22	0.07	0.27	22×	7×
TRWP-16	7.86	51.85	2.82	5.08	28.93	262.28	7.41	60.45	0.13	0.52	22×	10×

3.6.3.2 Backpropagation Time

Table 3.7: *Backpropagation time. The PyTorch GPU version is averaged on 10 trials and our CUDA version on 1000 trials. Ours is 691–1062 times faster than the PyTorch GPU version. Unit: second. Note: to make the comparison accurate, we keep 4 decimal digits for time precision.*

Method	PyTorch GPU		CUDA (ours)		Speed-up PyT/CUDA	
	32	96	32	96	32	96
ISGMR-4	7.3762	21.4818	0.0105	0.0311	702×	691×
ISGMR-8	18.8833	55.9235	0.0234	0.0672	807×	832×
ISGMR-16	58.2265	173.0217	0.0618	0.1818	942×	952×
TRWP-4	7.3460	21.4514	0.0089	0.0265	825×	810×
TRWP-8	18.8586	55.9392	0.0208	0.0585	907×	956×
TRWP-16	58.2642	172.9487	0.0558	0.1628	1044×	1062×

In Table 3.7, the backpropagation time clearly distinguishes the higher efficiency of CUDA versions than PyTorch GPU versions. On average, the CUDA versions are at least 690 times faster than PyTorch GPU versions, and only a low memory is used to store indices for backpropagation. This makes the backpropagation much faster than the forward propagation and enables its feasibility in deep learning. Analysis of PyTorch GPU version and our CUDA implementation are in Sec. 3.7.3.

3.7 Implementation Details and Computational Complexity

3.7.1 Maintaining Energy Function in Iterations

With the same notations in Eq. (3.1) and Eq. (3.9), let a general energy function in a MRF defined as

$$E(\mathbf{x}|\mathbf{\Theta}) = \sum_{i\in\mathcal{V}} \theta_i(x_i) + \sum_{(i,j)\in\mathcal{E}} \theta_{i,j}(x_i, x_j) \ . \tag{3.39}$$

In the standard SGM and ISGMR, given a node i and an edge from nodes j to i, the message will be updated at kth iteration as follows,

$$m_i^{r,k+1}(\lambda) = \min_{\mu\in\mathcal{L}} \left(\theta_{i-r}(\mu) + \theta_{i-r,i}(\mu,\lambda) + m_{i-r}^{r,k+1}(\mu) + \sum_{d\in\mathcal{R}\setminus\{r,r^-\}} m_{i-r}^{d,k}(\mu)\right) \ . \tag{3.40}$$

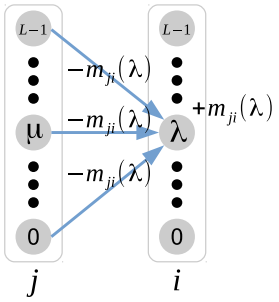


Figure 3.8: *Energy function maintained in an iterative message passing. When adding a term $m_{ji}(\lambda)$ to node i at label λ, the same value should be subtracted on all edges connecting node i at label λ.*

In Fig. 3.8, however, if we add a term $m_i(\lambda)$ to node i at label λ via $m_{ji}(\lambda)$ from node j to node i at label λ, the same value should be subtracted along all edges connecting this node i, that is $\forall(i,j) \in \mathcal{E}$, in order to maintain the same Eq. (3.39) in optimization. This supports the exclusion of r^- from $\mathcal{R}$ in Eq. (3.40). This is important for multiple iterations because the non-zero messages after the 1st iteration, as additional terms, will change the energy function via Eq. (3.40). Hence, a simple combination of many standard SGMs will change the energy function due to the lack of the subtraction above.

3.7.2 Indexing First Nodes by Interpolation

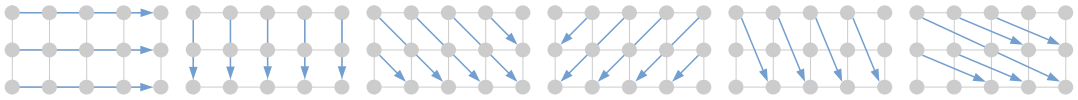


Figure 3.9: *Multi-direction message passing (forward passing in 6 directions). (a) horizontal trees. (b) vertical trees. (c) symmetric trees from up-left to down-right. (d) symmetric trees from up-right to down-left. (e) asymmetric narrow trees with height and width steps $S = (S_h, S_w) = (2,1)$. (f) asymmetric wide trees with $S = (1,2)$.*

Tree graphs contain horizontal, vertical, and diagonal (including symmetric, asymmetric wide, and asymmetric narrow) trees, shown in Fig. 3.9. Generally, the horizontal and vertical trees are for 4-connected graphs, symmetric trees are for 8-connected graphs, and asymmetric trees are for more than 8-connected graphs, resulting in different ways of indexing the first nodes for parallelization. In the following, we denote an image size with height H and width W, coordinates of the first node in vertical and horizontal directions as p_h and p_w respectively, and scanning steps in vertical and horizontal directions as S_h and S_w respectively.

3.7.2.1 Horizontal and Vertical Graph Trees

Coordinate of the first node of a horizontal and vertical tree, $p = (p_h, p_w)$, can be presented by $(p_h, 0)$ and $(0, p_w)$ respectively in the forward pass, and $(p_h, W-1)$ and $(H-1, p_w)$ respectively in the backward pass.

3.7.2.2 Symmetric and Asymmetric Wide Graph Trees

Coordinate of the first node $p = (p_h, p_w)$ is calculated by

$$\begin{aligned} N &= W + (H-1) * \text{abs}(S_w) \ , \\ p_w &= [0 : N-1] - (H-1) * \max\left(S_w, 0\right) \ , \\ p_h &= \begin{cases} 0 & \text{if } S_h > 0 \ , \\ H-1 & \text{otherwise} \ , \end{cases} \end{aligned} \tag{3.41}$$

where N is tree number, $\text{abs}(*)$ is absolution, and T_s is shifted indices of trees.

3.7.2.3 Asymmetric Narrow Graph Tees

Coordinate of the first node p by interpolation is calculated by

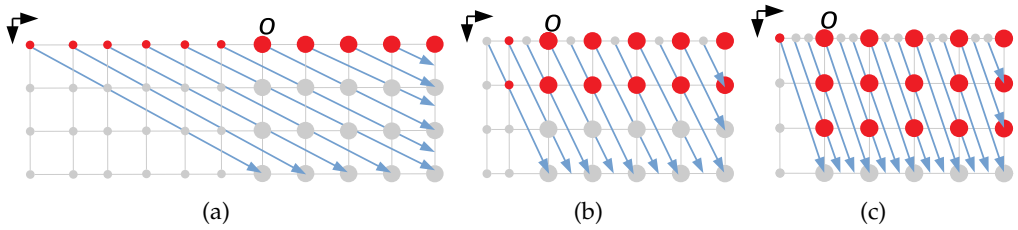
$$\begin{aligned} c_1 &= \text{mod}(T_s, \text{abs}(S_h)) \ , \\ c_2 &= \frac{\text{float}(T_s)}{\text{float}(\text{abs}(S_h))} \ , \\ p_h &= \begin{cases} \text{mod}(\text{abs}(S_h) - c_1, \text{abs}(S_h)) & \text{if } S_w > 0 \ , \\ c_1 & \text{otherwise} \ , \end{cases} \\ p_h &= H - 1 - p_h \text{ if } S_h < 0 \ , \\ p_w &= \begin{cases} \text{ceil}(c_2) & \text{if } S_w > 0 \ , \\ \text{floor}(c_2) & \text{otherwise} \ , \end{cases} \end{aligned} \tag{3.42}$$

where $\text{mod}(*)$ is modulo, $\text{floor}(*)$ and $\text{ceil}(*)$ are two integer approximations, $\text{float}(*)$ is data conversion for single-precision floating-point values, and the rest share the same notations in Eq. (3.41).

Although ISGMR and TRWP are parallelized over individual trees, message updates on a tree are sequential. The interpolation for asymmetric diagonals avoids as many redundant scanning as possible, shown in Fig. 3.10. This is more practical for realistic stereo image pairs that the width is much larger than the height.

3.7.3 PyTorch GPU Version versus Our CUDA Version

For the compared PyTorch GPU version, we highly paralleled individual trees in each direction while sequential message updates in each tree (equally scanline) are iterative. As Pytorch auto-grad is not customized for our min-sum message passing algorithms, these iterative message updates require to allocate new GPU memory



(a) (b) (c)

Figure 3.10: *Interpolation in asymmetric graph trees in the forward passing. (a) asymmetric wide trees with steps $S = (1, 2)$. (b) asymmetric narrow trees with $S = (2, 1)$. (c) asymmetric narrow trees with $S = (3, 1)$. Red circles are the first nodes of trees; large circles are within the image size; small circles are interpolated; o is the axes center. Coordinates of the interpolations in (a) are integral; those in (b)-(c) round to the nearest integers by Eq. (3.42).*

for each updated message, which makes it very inefficient and memory-consuming. Its backpropagation is slower since extra memory is needed to unroll the forward message passing to compute gradients of messages and all intermediate variables that require gradients.

In contrast, our implementation is specific to the min-sum message passing. This min-sum form greatly accelerates our backpropagation by updating gradients only related to the indices which are stored in pre-allocated GPU memory during forward pass (line 10 in Alg. 1). For example, from node i to $i + r$ in Fig. 2(a), forward pass needs messages over 9 edges (grey lines); but only one (1 of 3 blue lines) from $i + r$ to i requires gradient updates in the backpropagation. This makes our CUDA implementation much faster than the PyTorch GPU version, especially the backpropagation with at least 690× speed-up.

3.7.4 Computational Complexity of Min-Sum & Sum-Product TRW

Given a graph with parameters $\{\theta_i, \theta_{i,j}\}$, maximum iteration K, set of edges $\{\mathcal{E}^r\}$, disparities $\mathcal{L}$, directions $\mathcal{R}$, computational complexities of min-sum and sum-product TRW are shown below. For the efficient implementation, let $\theta_{i,j}(\lambda, \mu) = \theta_{i,j} V(\lambda, \mu)$.

3.7.4.1 Computational Complexity of Min-Sum TRW

Representation of a message update in min-sum TRW is

$$m_i^r(\lambda) = \min_{\mu \in \mathcal{L}} \left(\rho_{i-r,i}\big(\theta_{i-r}(\mu) + \sum_{d \in \mathcal{R}} m_{i-r}^d(\mu)\big) - m_{i-r}^{r^-}(\mu) + \theta_{i-r,i} V(\mu, \lambda) \right). \tag{3.43}$$

In our case where the maximum disparity is less than 256, memory for the backpropagatio of the min-sum TRW above is only for indices $\mu^* = p_{k,i}^r(\lambda) \in \mathcal{L}$ from message minimization with $K \sum_{r \in \mathcal{R}} |\mathcal{E}^r||\mathcal{L}|$ bytes 8-bit unsigned integer values, as well as for indices from message reparametrization with $K \sum_{r \in \mathcal{R}} |\mathcal{E}^r|$ bytes. In total, the min-sum TRW needs $K \sum_{r \in \mathcal{R}} |\mathcal{E}^r| \, (|\mathcal{L}| + 1)$ bytes for the backpropagation.

3.7.4.2 Computational Complexity of Sum-Product TRW

Representation of a message update in sum-product TRW is

$$\begin{aligned}\exp(-m_i^r(\lambda)) &= \sum_{\mu\in\mathcal{L}} \exp\Big(-\rho_{i-r,i}\big(\theta_{i-r}(\mu) + \sum_{d\in\mathcal{R}} m_{i-r}^d(\mu)\big) + m_{i-r}^{r^-}(\mu) \\ &\quad - \theta_{i-r,i}V(\mu,\lambda)\Big) \\ &= \sum_{\mu\in\mathcal{L}} \Big(\exp\big(-\rho_{i-r,i}\theta_{i-r}(\mu)\big) \prod_{d\in\mathcal{R}} \exp\big(-\rho_{i-r,i}m_{i-r}^d(\mu)\big) \\ &\quad \exp(m_{i-r}^{r^-}(\mu)) \exp\big(-\theta_{i-r,i}V(\mu,\lambda)\big)\Big)\,.\end{aligned} \tag{3.44}$$

Usually, it can be represented as

$$\tilde{m}_i^r(\lambda) = \sum_{\mu\in\mathcal{L}} \Big(\exp^{-\rho_{i-r,i}\underline{\theta_{i-r}(\mu)}_1} \prod_{d\in\mathcal{R}} \big(\underline{\tilde{m}_{i-r}^d(\mu)}_2\big)^{\rho_{i-r,i}} \frac{1}{\underline{\tilde{m}_{i-r}^{r^-}(\mu)}_3} \exp^{-\underline{\theta_{i-r,i}}_4\underline{V(\mu,\lambda)}_5} \Big)\,. \tag{3.45}$$

Problem 1: Numerical Overflow: For single-precision floating-point data, a valid numerical range of x in $\exp(x)$ is less than around 88.7229; otherwise, it will be infinite. Therefore, for the exponential index in Eq. (3.44), a numerical overflow will happen quite easily. One solution is to reparametrize these messages to a small range, such as $[0,1]$, in the same manner as SoftMax(), which requires logarithm to find the maximum index, followed by exponential operations.
Problem 2: Low efficiency OR high memory requirement in backpropagation: In the backpropagation, due to the factorization in Eq. (3.45), it needs to rerun the forward propagation to calculate intermediate values OR store all these values in the forward propagation. However, the former makes the backpropagation at least as slow as the forward propagation while the later requires a large memory, $K\sum_{r\in\mathcal{R}}|\mathcal{E}^r||\mathcal{L}|\,(8|\mathcal{L}|+4|\mathcal{R}||\mathcal{L}|+4)$ bytes single-precision floating-point values.
Derivation:

For one message update in Eq. (3.45), the gradient calculation of terms 1,2-3,4,5 (underlined) requires $4\times\{|\mathcal{L}|,|\mathcal{R}||\mathcal{L}|,1,|\mathcal{L}|\}$ bytes respectively. For K iterations, set of directions $\mathcal{R}$, edges $\{\mathcal{E}^r\}$, $\forall r\in\mathcal{R}$, it requires $K\sum_{r\in\mathcal{R}}|\mathcal{E}^r||\mathcal{L}|\,(8|\mathcal{L}|+4|\mathcal{R}||\mathcal{L}|+4)$ bytes in total. This is in $\mathcal{O}(|\mathcal{R}||\mathcal{L}|)$ order higher than the memory requirement in the min-sum TRW memory requirement, $K\sum_{r\in\mathcal{R}}|\mathcal{E}^r|\,(|\mathcal{L}|+1)$ bytes.

3.8 Conclusion

In this chapter, we introduce two fast and differentiable message passing algorithms, namely, ISGMR and TRWP. While ISGMR improved the effectiveness of SGM, TRWP sped up TRWS by two orders of magnitude without loss of solution quality. Besides, our CUDA implementations achieved at least 7 times and 690 times speed-up compared to PyTorch GPU versions in the forward and backward propagation respectively. These enable end-to-end learning with effective and efficient MRF opti-

mization algorithms. Experiments of stereo vision and image denoising as well as end-to-end learning for semantic segmentation validated the effectiveness and efficiency of our proposals.

Chapter 4

Refining Semantic Segmentation with Superpixels

In this chapter, we propose a transparent initialization module and a sparse encoder for semantic segmentation with edge constraints from superpixel contours obtained by CNNs. This aims at jointly learning multiple pretrained state-of-the-art CNNs for different tasks with the hope of embedding useful information from each other. The strategy of the transparent initialization module is to identically mapping the outputs of additional layers to the inputs as a parameter initialization for the additional layers at the early training phase, and then gradually finetune these pretrained models through the additional layers.

4.1 Motivation

Semantic segmentation is an essential task in computer vision which requires mapping image pixels of interesting objects to predefined semantic labels. Applications include autonomous driving [Feng et al., 2020], object identification [Mottaghi et al., 2014; Salscheider, 2019], image editing [Hong et al., 2018], and scene analysis [Hofmarcher et al., 2019]. Recent developments of semantic segmentation are greatly promoted by deep learning on several large-scale datasets [Hariharan et al., 2011; Lin et al., 2014], resulting in effective networks [Long et al., 2015; Zhao et al., 2017; Zheng et al., 2015; Chen et al., 2017, 2016a, 2018b; Zhang et al., 2018b, 2020]. Semantic segmentation obtained by these methods, however, is not substantially aligned with object edges.

This problem can be alleviated by using qualified edge-preserving methods [Krähenbühl and Koltun, 2011; Zheng et al., 2015; Chen et al., 2017] or independently learning edges for semantic segmentation [Jampani et al., 2018; Gadde et al., 2016; Chen et al., 2016a]. Nevertheless, those edges are usually incomplete and oversegmented. To solve this problem, denseCRF [Krähenbühl and Koltun, 2011] based methods aggregate object features over a large and dense range using bilateral filters with high-dimensional lattice computations. This, however, could be more efficient and desirable to learn on locally oversegmented areas, such as superpixels that aggregate similar pixels into a higher-order clique [Achanta et al., 2012; Li and Chen, 2015].

Existing superpixel segmentation approaches are mainly categorized into traditional methods, such as SLIC [Achanta et al., 2012], LSC [Li and Chen, 2015], Crisp [Isola et al., 2014], BASS [Uziel et al., 2019], and those using neural networks, such as SSN [Jampani et al., 2018], affinity loss [Tu et al., 2018], and superpixels by FCN [Yang et al., 2020]. Although those traditional methods have qualified performance on superpixel segmentation, they cannot be easily embedded into neural networks for end-to-end learning due to nondifferentiability or large computational complexity. In contrast, Jampani et al. [2018] provides an end-to-end learning for superpixel semantic segmentation, but the pixel labels in each superpixel are not always consistent. Similarly, Yang et al. [2020] uses a fully convolutional network to learn superpixels for stereo matching, which is the current state-of-the-art in superpixel learning. However, the problem of inconsistent pixel labels in each superpixel will also occur when [Yang et al., 2020] is applied to semantic segmentation.

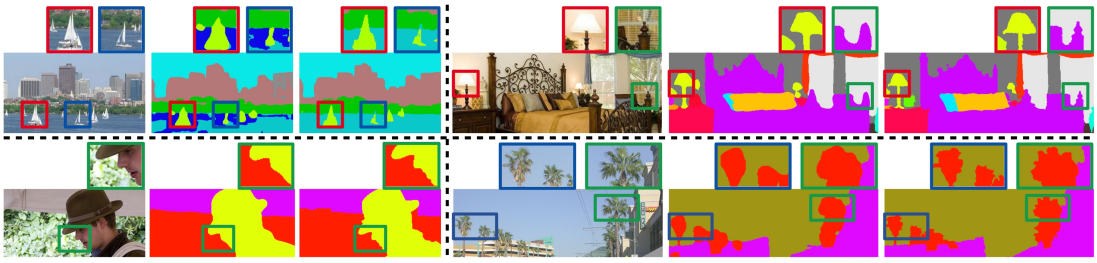


Figure 4.1: *Edge comparison with the state-of-the-art methods, that is ResNeSt200 on ADE20K (top row) and ResNeSt269 on PASCAL Context (bottom row). Left: RGB, middle: state-of-the-art, right: ours. Ours has a better alignment with object edges than the state-of-the-art methods.*

To deal with the problems of edge loss and inconsistent superpixel labelling, we jointly train these networks with additional fully-connected layers using transparent initialization and logit consistency, resulting in enhanced object edges in Fig. 4.1. Specifically, transparent initialization warm starts the training by recovering the pretrained network output from its input at initialization, followed by a gradual improvement in learning superpixels. Simultaneously, logit consistency with sparse encoder enables efficient logit averaging to guarantee consistent pixel labels in each superpixel. Furthermore, we used the popular performance ratio [Khaire and Thakur, 2012] and F-measure [Stutz et al., 2018; Flach and Kull, 2015] to evaluate the quality of semantic segmentation edges. Contributions of this work are:

- We propose an end-to-end joint learning strategy to fuse multiple state-of-the-art networks to enhance the performance of one network (semantic segmentation) with auxiliary features from the others (superpixels).
- We are the first to propose an effective identity mapping, namely transparent initialization, with densely initialized parameters for additional fully-connected layers to warm up the joint learning.

- We propose a simple yet efficient and learnable logit consistency to ensure consistent labels in each superpixel. Both are indexed and operated sparsely by sparse encoder and decoder as a technical contribution.

4.2 Related Work

4.2.1 Semantic Segmentation

Semantic segmentation can be traced back to early techniques [Thoma, 2016] based on classifiers, such as random decision forests [Schroff et al., 2008], SVMs [Wang et al., 2011], and graphic models with MRFs and CRFs [Jordan, 1998; Krähenbühl and Koltun, 2011; Ajanthan et al., 2017]. In contrast, modern state-of-the-art methods rely on advanced exploitation of deep CNN classifiers, such as ResNet [He et al., 2016], DenseNet [Huang et al., 2017], and VGG16 [Simonyan and Zisserman, 2015]. Fully Convolutional Network (FCN) [Long et al., 2015] and related methods are typical architectures that use rich image features from classifiers usually pretrained on ImageNet [Deng et al., 2009]. SegNet [Badrinarayanan et al., 2017] uses a U-Net structure for an encoder-decoder module to compensate for low resolution by using multiple upsampled feature maps. In addition, several multiscale contextual fusion methods [Chen et al., 2016a; Lin et al., 2017; Chen et al., 2018b] have been proposed to aggregate pyramid feature maps for fine-grained segmentation. Typically, DeepLabV3+ [Chen et al., 2018b] combines spatial pyramid pooling and encoder-decoder modules to refine the segmentation along with object boundaries. Recently, attention-based networks [Li et al., 2019b; Zhao et al., 2018; Chen et al., 2018c; Fu et al., 2019] improve object labelling confidence by aggregating features of a single pixel with those from other positions. Zhang et al. [2020] introduced a split-attention block in ResNet (ResNeSt), which enables multiscale scores with softmax for attention across feature-map groups. This achieves the new state-of-the-art performance in semantic segmentation and image classification. Due to the limited capability of network architectures, however, a huge improvement on mean Intersection over Union (mIoU) is hard to achieve; see the minor improvement in Zhao et al. [2019].

4.2.2 Superpixel Segmentation

Superpixel segmentation has been well studied for years with a comprehensive survey in Stutz et al. [2018]. In contrast to classical methods that initialize superpixel regions with seeds and cluster pixel sets using distance measurement [Wang et al., 2013], boundary pixel exchange [Bergh et al., 2015], *etc.*, the widely-used SLIC based methods [Achanta et al., 2012; Liu et al., 2015a; Li and Chen, 2015; Achanta and Susstrunk, 2017] employ (weighted) K-means clustering on pixel feature vectors to group neighbouring pixels. In deep learning, SSN [Jampani et al., 2018] first proposes an end-to-end learning framework for superpixels with differentiable SLIC for semantic segmentation and optical flow. By comparison, Yang et al. [2020] replaces the soft K-means manner in SSN with a simple fully convolutional network and ap-

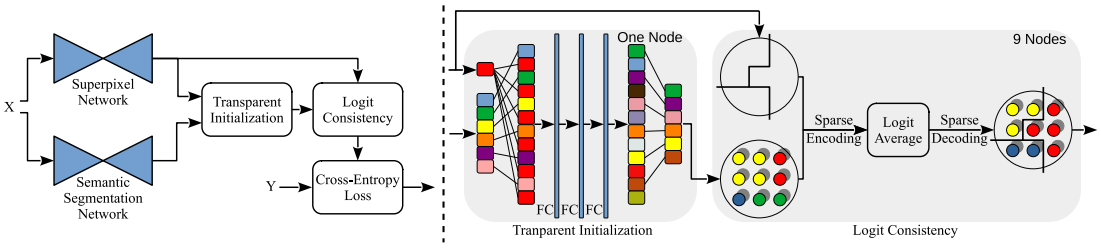


Figure 4.2: *Flowchart with an example of 6 labels. "X": input image, "Y": semantic segmentation ground truth, "Transparent Initialization": identity mapping by Fully-Connected (FC) layers, "Logit Consistency": consistent pixel labels in each superpixel. A node is a pixel. The last FC layer in transparent initialization is initialized by the right inverse of the matrix multiplication of all the other FC layers.*

plies it to stereo matching with downsampling and upsampling modules. While SSN results in superpixel-level semantic segmentation, the pixel labelling is not aligned with the superpixels. For instance, inconsistent labels exist in each superpixel, which reduces the effects of superpixels on semantic assignments.

4.2.3 Superpixel Semantic Segmentation

Some works use superpixels to optimize graph relations [Gould et al., 2014; Xing et al., 2016] or downsample images as a pooling alternative to max or average pooling [Gadde et al., 2016; Park et al., 2017; Schuurmans et al., 2018]. These methods usually use fixed superpixels obtained by traditional methods mentioned above ("Superpixel Segmentation") and lose the exact alignment of segmentation edges with superpixel contours after upsampling. This, however, can be easily achieved by our logit consistency module. Moreover, fixed superpixels are unsuitable for end-to-end learning since traditional methods, such as SLIC [Achanta et al., 2012], are computationally expensive due to CPU execution and inflexible for fine-tuning superpixels around object edges, especially for small objects. Instead, superpixels learned by CNN alleviate this problem.

4.3 Methodology

The flowchart of our method is shown in Fig. 4.2. Subsequently, we discuss each of these modules in detail.

4.3.1 Network Architectures

We use the state-of-the-art DeepLabV3+ [Chen et al., 2018b] and ResNeSt [Zhang et al., 2020] for semantic segmentation and the state-of-the-art superpixels with FCN [Yang et al., 2020] for superpixel contours.

DeepLabV3+ achieves high and robust performance with atrous spatial pyramid pooling for multiscale feature maps and encoder-decoder modules for deep features with different output strides. It is widely used as a network backbone for semantic

segmentation due to these well-studied modules evaluated by empirical experiments. ResNeSt [Zhang et al., 2020] replaces ResNet with multiscale scores using softmax-based feature map attention and achieves the new state-of-the-art. The loss function for semantic segmentation is the standard cross-entropy loss of predicted logits and ground truth labelling. Readers can refer Chen et al. [2018b] and Zhang et al. [2020] for more details.

Superpixel with FCN [Yang et al., 2020] is the current state-of-the-art superpixel network. The core idea is to construct a distance-based loss function with aggregations of neighbouring pixel and superpixel properties and locations. This is similar to the SLIC method [Achanta et al., 2012], where the property vectors can be CIELAB colors or one-hot semantic encoding. The loss function for superpixel network with FCN [Yang et al., 2020] is in Eq. (4.1).

Given an image with N_p pixels and N_s superpixels, we denote the subsets of pixels as $\mathcal{P} = \{\mathcal{P}_0, ..., \mathcal{P}_{N_s-1}\}$, where $\mathcal{P}_i$ is a subset of pixels belonging to superpixel i. With pixel property $\mathbf{f} \in \mathbb{R}^{N_p \times K}$ (K features for each pixel) and probability map $\mathbf{q} \in \mathbb{R}^{N_s \times N_p}$, the loss function is

$$\mathcal{L}(\mathbf{f}, \mathbf{q}) = \sum_{p \in \mathcal{P}} E\left(\mathbf{f}(p), \mathbf{f}'(p)\right) + \frac{m}{D}||\mathbf{c}(p) - \mathbf{c}'(p)||_2 \tag{4.1}$$

with

$$\mathbf{u}_s = \frac{\sum_{p \in \mathcal{P}_s} \mathbf{f}(p) q_s(p)}{\sum_{p \in \mathcal{P}_s} q_s(p)}, \quad \mathbf{l}_s = \frac{\sum_{p \in \mathcal{P}_s} \mathbf{c}(p) q_s(p)}{\sum_{p \in \mathcal{P}_s} q_s(p)}, \tag{4.2a}$$

$$\mathbf{f}'(p) = \sum_{s \in \mathcal{N}_p} \mathbf{u}_s q_s(p), \quad \mathbf{c}'(p) = \sum_{s \in \mathcal{N}_p} \mathbf{l}_s q_s(p), \tag{4.2b}$$

where m is a weight balancing the effects of property and coordinates on loss, D is a superpixel sampling interval in proportion to superpixel size, $\mathbf{c}(i) = [x_i, y_i]^T$ for all $i \in \{1, ..., N_p\}$ are pixel coordinates, $E(\cdot, \cdot)$ is a distance measure function involving L_2 norm or cross-entropy, $q_s(p) \in \mathbf{q}$ is the probability of pixel p belonging to superpixel s, and $\mathcal{N}_p$ is a set of superpixels surrounding pixel p.

Here, $\mathbf{u}_s$ and $\mathbf{l}_s$ are superpixel-level property vector and central coordinates aggregated from involved pixels respectively, and $\mathbf{f}'$ and $\mathbf{c}'$ are pixel-level property and coordinates aggregated from surrounding superpixels. This updates between pixels and superpixels until the loss converges.

4.3.2 Learning with Transparent Initialization

Although some semantic segmentation datasets, such as PASCAL VOC [Everingham et al., 2014] and Berkeley benchmark [Hariharan et al., 2011], have no accurate edges for supervised edge learning, it can be compensated for by a superpixel network on edge-specified datasets, such as SBDS500 [Arbelaez et al., 2010]. Due to the domain gap of datasets, however, pretrained superpixel models are more desirable than learning from scratch for fast loss convergence. Superpixel maps pretrained

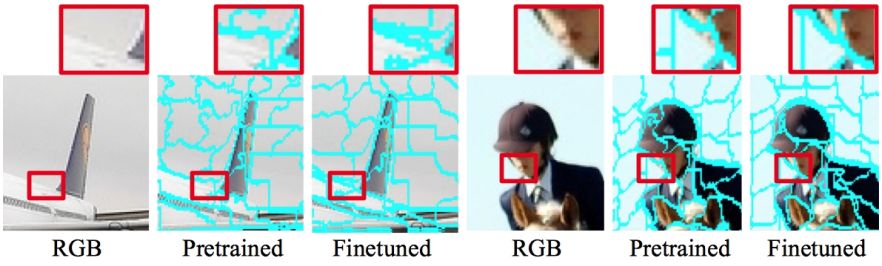


Figure 4.3: *Pretrained superpixles on BSDS500 [Yang et al., 2020] versus fine-tuned superpixels on PASCAL VOC 2012 using our joint learning. Fine-tuned superpixels recover accurate object edges to alleviate the domain gap between datasets. Best view by zoom-in.*

on BSDS500, however, are not always suitable for semantic segmentation datasets. Hence, fine-tuning by joint learning is necessary to improve the quality of superpixel contours, compared with using the pretrained network, as shown in Fig. 4.3.

Therefore, in our proposal, linear layers or convolutional layers with 1×1 kernels are added to fuse the outputs of superpixel and semantic segmentation networks. Either or both can be pretrained. Importantly, selecting an appropriate initialization on these additional layers is important to avoid overriding the effect of learned network parameters. It is straightforward to cast the layer operations as an identity mapping between input and output at the early training stage. Net2Net [Chen et al., 2016b] achieves this by using identity matrices to initialize linear layers. This method, however, is inefficient in learning since the identity matrices will result in highly sparse gradients. In addition, it cannot handle activation functions with negative values.

In contrast, we introduce *transparent initialization* with non-zero values for dense gradients, while identically mapping the layer input to output and preserving the effect of the learned parameters of the pretrained networks.

4.3.2.1 Affine Layers

A linear layer without activation (such as a convolution or fully-connected layer) can be written as $\mathbf{y} = \mathbf{x}\mathbf{A} + \mathbf{b}$ with layer weights $\mathbf{A}$, bias $\mathbf{b}$, and input $\mathbf{x}$, by matrix multiplication with $\tilde{\mathbf{y}} = [\mathbf{y}, \mathbf{1}]$:

$$\tilde{\mathbf{y}} = \tilde{\mathbf{x}}M = [\mathbf{x}, \mathbf{1}] \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{b} & \mathbf{1} \end{bmatrix}. \tag{4.3}$$

For simplicity the mapping Eq. (4.3) will be denoted as $\mathbf{y} = \mathbf{x}\,\mathtt{A}$ where $\mathtt{A}$ denotes the affine transformation, and we place the functions on the right. Note the difference between $\mathtt{A}$, an affine transform and $\mathbf{A}$, a matrix. If an activation function σ is included, then the mapping is $\mathbf{x} \mapsto \mathbf{x}\,\mathtt{A}\,\sigma$.

The right-inverse of an affine transformation Eq. (4.3) is represented by the matrix

$$M^R = \begin{bmatrix} \mathbf{A}^R & \mathbf{0} \\ -\mathbf{b}\mathbf{A}^R & \mathbf{1} \end{bmatrix}, \tag{4.4}$$

satisfying $MM^R = I$, the identity map. Here, $\mathbf{A}^R$ is the right-inverse of matrix $\mathbf{A}$, which exists if $\mathbf{A}$ has dimension $m \times n$ and rank m, in which case $\mathbf{A}^R = \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)^{-1}$.

4.3.2.2 Transparent Initialization

The idea behind transparent layer initialization is to construct a sequence of $k \geq 2$ affine layers denoted $\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_k$ such that $\mathtt{A}_k$ is a right-inverse of the product $\mathtt{A}_1 \ldots \mathtt{A}_{k-1}$, satisfying $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_k = \mathtt{I}$, the identity transformation.

A necessary condition for this right inverse to exist is that $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1}$ should be of full rank. Since a matrix with random entries will almost surely have full rank, this leads to the following condition.

Theorem 1 *Let a sequence of affine transformations $\mathtt{A}_i : \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}$, for $i = 1, \ldots, k-1$ be chosen at random. Then $\mathtt{A}_1 \ldots \mathtt{A}_{k-1}$ almost surely has a right inverse if and only if $m_i \geq m_0$ for $i = 1, \ldots, k-1$.*

Therefore, the strategy for selecting a set of parameters for a sequence of affine layers may be described as follows. For the sequence of layers to represent an identity transform, we need that the input and output space have the same dimension, namely $m_0 = m_k$. Then

1. Select intermediate dimensions $m_1, \ldots, m_{k-1}$ such that $m_i \geq m_0$ for all $i = 1, \ldots, k-1$.

2. Define random affine transforms $\mathtt{A}_i$ by selecting the entries of matrices $\mathbf{A}_i$ and vectors $\mathbf{b}_i$ randomly, using a suitable random number generator, for instance by a zero-mean normal (Gaussian) distribution.

3. Compute the composition $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1}$ by matrix multiplication, and take its right-inverse.

4. Set $\mathtt{A}_k$ to equal $(\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1})^R$.

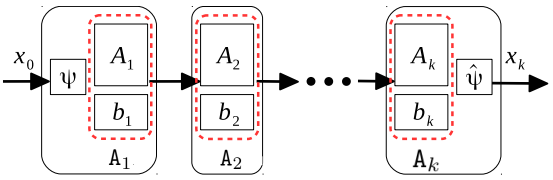
The resulting composite mapping $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_k$ is the identity affine transformation.

4.3.2.3 With Activation

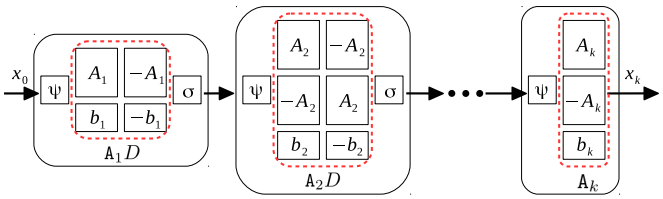
Usually, each affine mapping $\mathtt{A}$ will be followed by an activation function σ. For now, we assume that this is ReLU$(\cdot)$. Our approach including activations is to apply the activation to both $\mathbf{x}$ and $-\mathbf{x}$ and then sum them. Consider

$$\mathbf{x} \overset{D}{\mapsto} [\mathbf{x}, -\mathbf{x}] \overset{\sigma}{\mapsto} [\mathbf{x}^+, \mathbf{x}^-] \overset{S}{\mapsto} \mathbf{x}^+ - \mathbf{x}^- = \mathbf{x}\,, \tag{4.5}$$

where mappings D (duplicate) and S (subtract) are the mappings as shown above, and $\mathbf{x}^+$ and $\mathbf{x}^-$ are the positive and negative components of $\mathbf{x}$. This shows that $\mathbf{x}D\sigma S = \mathbf{x}$, and so $D\sigma S$ is the identity mapping. Note also that both D and S are affine (in fact linear) transformations.



(a) Without Activation



(b) With Activation

Figure 4.4: *Transparent initialization.* $\psi : \mathbf{x} \mapsto [\mathbf{x}, \mathbf{1}]$; $\hat{\psi} : [\mathbf{x}, \mathbf{1}] \mapsto \mathbf{x}$; σ*: activation function. The three modules in (b), from the left to the right, are corresponding to Eq. (4.8)-Eq. (4.10).*

Applying this to a sequence of affine transformations $\mathtt{A}_i$ such that $\mathbf{x}\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_k = \mathbf{x}$ gives

$$\mathbf{x}\,(\mathtt{A}_1 D\sigma S)\,(\mathtt{A}_2 D\sigma S)\,\ldots\,(\mathtt{A}_{k-1} D\sigma S)\,\mathtt{A}_k = \mathbf{x}\,. \tag{4.6}$$

Bracketing this differently gives

$$\mathbf{x}(\mathtt{A}_1 D\sigma)\,(S\mathtt{A}_2 D\sigma)\,\ldots\,(S\mathtt{A}_{k-1} D\sigma)\,(S\mathtt{A}_k) = \mathbf{x}\,, \tag{4.7}$$

where now each bracket is an affine mapping followed by the activation σ (except the last). This may then be implemented as a sequence of affine layers (convolution or fully-connected) with activations. Thus, $\mathtt{A}_1 D$ is the mapping

$$\mathbf{x} \mapsto \mathbf{x} \begin{bmatrix} \mathbf{A}_1 & -\mathbf{A}_1 \end{bmatrix} + [\mathbf{b}_1, -\mathbf{b}_1]\,. \tag{4.8}$$

The mapping $S\mathtt{A}_i D$ is the affine transformation

$$[\mathbf{x}^+, \mathbf{x}^-] \mapsto [\mathbf{x}^+, \mathbf{x}^-] \begin{bmatrix} \mathbf{A}_i & -\mathbf{A}_i \\ -\mathbf{A}_i & \mathbf{A}_i \end{bmatrix} + [\mathbf{b}_i, -\mathbf{b}_i] \tag{4.9}$$

and $S\mathtt{A}_k$ is the mapping

$$[\mathbf{x}^+, \mathbf{x}^-] \mapsto [\mathbf{x}^+, \mathbf{x}^-] \begin{bmatrix} \mathbf{A}_k \\ -\mathbf{A}_k \end{bmatrix} + \mathbf{b}_k\,. \tag{4.10}$$

The structure of the network may be represented as in Fig. 4.4. Observe that the outputs of intermediate layers have twice the dimension of the output of the affine mappings $\mathtt{A}_i$.

It is important to note that the structured form of the affine mappings in Eq. (4.8)-Eq. (4.9) are for initialization only. There is no sharing parameters (such as $\mathtt{A}_i$ and

$-\mathtt{A}_i$) and layers are free to implement any affine transform during training. In fact, $\mathbf{x}$ in any activation functions satisfying

$$\sigma(\mathbf{x}) - \sigma(-\mathbf{x}) = c\mathbf{x} \,, \tag{4.11}$$

where c is a non-zero constant, can be recovered, such as SoftReLU($\cdot$) defined by $\sigma(\mathbf{x}) = \log(1 + e^{\mathbf{x}})$ and LeakyReLU($\cdot$) [Xu et al., 2015]:

$$\mathbf{x} = \frac{1}{1+\delta} \left[\sigma(\mathbf{x}),\, \sigma(-\mathbf{x})\right] \begin{bmatrix} \mathbf{I} \\ -\mathbf{I} \end{bmatrix} \,, \tag{4.12}$$

where $\delta > 0$ is the slope of the negative part of LeakyReLU($\cdot$). The substitution of Eq. (4.12) to Eq. (4.8)-Eq. (4.9) contributes to the identity mapping in our transparent initialization. One can derive it in a similar way to ReLU($\cdot$).

4.3.3 Logit Consistency with Sparse Encoder

In addition to the notations of pixel number N_p, superpixel number N_s, and subsets of pixels $\mathcal{P} = \{\mathcal{P}_0, ..., \mathcal{P}_{N_s-1}\}$ in Sec. 4.3.1, the label set is defined as $\mathcal{L} = \{0, ..., N_l - 1\}$ given N_l labels. For logit x_s^l of $\mathcal{P}_s$ for superpixel s at label l, the logit consistency follows

$$x_s^l(p) \leftarrow \frac{1}{|\mathcal{P}_s|} \sum_{p \in \mathcal{P}_s} x_s^l(p), \; \forall p \in \mathcal{P}_s \,, \tag{4.13}$$

which guarantees all pixels in $\mathcal{P}_s$ having the same logit at each label so as to be assigned with the same label.

Nevertheless, considering the high complexity of indexing $x_s^l(p)$, a dense matrix operation requires a large GPU memory, that is $N_l N_s N_p$, especially for the back-propagation in CNN learning. This makes it infeasible for training due to the limited GPU memory in our experiments.

Table 4.1: *Example of sparse property for indexing N_p pixels by N_s superpixels. Each superpixel contains only a few pixels ("1" in each row) for logit consistency. Hence, an efficient encoding is achieved by a sparse $N_s \times N_p$ matrix with N_p non-zero elements.*

		pixel index						
		0	1	2	3	4	$\cdots$	N_p-1
superpixel index	0	0	1	1	0	0	$\cdots$	0
	1	0	0	0	1	1	$\cdots$	0
	2	1	0	0	0	0	$\cdots$	1
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	N_s-1	0	0	0	0	0	$\cdots$	0
	sum	1	1	1	1	1	$\cdots$	1

Hence, we adopted a sparse encoder, including sparse encoding and decoding, with sparse matrix operations for the consistency. Let us set matrix for indexing pixels by superpixels as $M(s, p)$, logit matrix as $M(l, p)$, where $s \in \mathcal{S}$, $p \in \mathcal{P}$, and

$l \in \mathcal{L}$, sparse encoding and decoding are

$$\text{Encoding: } M(s,l) = \frac{\text{SMM}(M(s,p), M^T(l,p))}{\text{SADD}_{p\in\mathcal{P}_s}(M(s,p))} , \tag{4.14a}$$

$$\text{Decoding: } M(l,p) \leftarrow \text{SMM}(M^T(s,l), M(s,p)) , \tag{4.14b}$$

where $M(s,p) \in \mathcal{B}^{N_s N_p}$ is {0, 1} binary, shown in Table 4.1, SMM(·) is sparse matrix multiplication, and SADD(·) is sparse addition. This converts Eq. (4.13) from dense operations to sparse with reduced complexity from $N_l N_s N_p$ to $N_l N_p$. Otherwise, it is infeasible to jointly train the networks due to the limited GPU memory in our experiments.

4.4 Experiments

We first evaluated the properties of our transparent initialization including its effectiveness on data recovery and numerical stability. Then, we demonstrated its effect on jointly learning pretrained networks of semantic segmentation and superpixels together with a sparse encoder for logit consistency.

4.4.1 Properties of Transparent Initialization

4.4.1.1 Effectiveness

Table 4.2: *Our transparent initialization has high initialization and recovery rates on 3 Fully-Connected (FC) layers and supports non-square filters. "init. rate": percentage of non-zero (absolute value $> \epsilon$) parameters; "recovery rate": percentage of outputs with the same (difference $< \epsilon$) values as inputs'; "activation": ReLU(·); "non-square filter": a FC layer with different in_channels and out_channels. Inputs are in [-10, 10] and ϵ=1e-4.*

Manner	Init. Rate↑	Recovery Rate↑ w/o Activation	Recovery Rate↑ w Activation	Non-square Filter Supported
Random	98.2	0.0	0.0	✓
Xavier [Glorot and Bengio, 2010]	98.2	0.0	0.0	✓
Net2Net [Chen et al., 2016b]	2.3	100.0	50.0	✗
Ours	**99.9**	**100.0**	**100.0**	✓

Our transparent initialization aims at identically mapping the output of linear layer(s) to the input at the early stage when fine-tuning pretrained network(s). It retains the effect of learned parameters of pretrained model(s). Off-the-shelf parameter initialization methods, such as random (uniform) and Xavier [Glorot and Bengio, 2010] initialization, lead to random values of the output at the early training stage, which cannot generate effective features by the pretrained models. Also, compared with the identical initialization with identity matrices for deeper networks in Net2Net [Chen et al., 2016b], our transparent initialization has a high initialization

rate with much more non-zero parameters for dense gradients in the backpropagation.

We evaluated these methods on 3 fully-connected layers. (in_channels, out_channels) for each layer is (42, 64)→(64, 64)→(64, 42). Since Net2Net only supports square linear layer, *i.e.*, in_channels equals out_channels, to increase network depth, all layers have 42 in_channels and out_channels. Input data is normally distributed with size (4, 42, 512, 512) as (batch, in_channels, height, width).

In Table 4.2, random and Xavier initialization have ∼98% initialization rate but cannot recover the output from its input, leading to 0% recovery rate. Net2Net [Chen et al., 2016b] has only ∼2% initialization rate and 50% recovery rate with ReLU(·) for non-negative values only. In contrast, our transparent initialization has a high initialization rate and 100% recovery rate by Eq. (4.8)-Eq. (4.10) with ReLU(·).

4.4.1.2 Numerical Stability

Table 4.3: *Stability of transparent initialization with numerical orders 1, 10, 1e2, 1e3. Layer parameters are consistent with Table 4.2.*

	[-1, 1]	[-10, 10]	[-1e2, 1e2]	[-1e3, 1e3]
Max Error	∼6.8e-6	∼6.6e-5	∼6.5e-4	∼6.6e-3

Since the effect of our transparent initialization is distributed across layers and Eq. (4.4) is achieved by a pseudo-inverse matrix for a rank-deficient matrix, hidden layers have round-off errors, and thus, the matrix multiplication of layer parameters is not strictly identical. We therefore tested the numerical stability of our transparent initialization on 4 numerical orders in Table 4.3. Clearly, the max error between input and output is in proportion to the magnitude of input values. For a pretrained model, its output is usually stable in a numerical range, such as probability in [0, 1]. One can easily enforce a numerical regularization if the model output is out of range.

4.4.2 Implementation Setup

4.4.2.1 Datasets

We evaluated our proposal on 3 popular semantic segmentation datasets, ADE20K, PASCAL VOC 2012, and PASCAL Context. *ADE20K* [Zhou et al., 2017, 2016] has 150 semantic categories for indoor and outdoor objects. It contains 20,210 samples for training and 2,000 samples for validation. *PASCAL Context* [Mottaghi et al., 2014] has additional annotations for PASCAL VOC 2010 and provides annotations for the whole scene with 400+ classes. We selected the given 59 categories for semantic segmentation with 4,996 training samples and 5,104 validation samples. *PASCAL VOC 2012* [Everingham et al., 2014] and Berkeley benchmark [Hariharan et al., 2011] were used as a combined version for 21 classes segmentation. This dataset has 1,449 images from PASCAL VOC 2012 val set for validation and 10,582 images for training.

Meanwhile, *MS-COCO* [Lin et al., 2014] was used to pretrain the semantic segmentation network, *i.e.*, DeepLabV3+ in our case, for PASCAL VOC 2012. It has 92,516 images for training and 3,899 images for validation, while 20 object classes from the primary 80 classes were selected in accordance with PASCAL VOC 2012.

For superpixel networks, we used the state-of-the-art *superpixel network with FCN* from Yang et al. [2020] that was pretrained on Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500) [Arbelaez et al., 2010] containing 500 images with handcrafted ground truth edges.

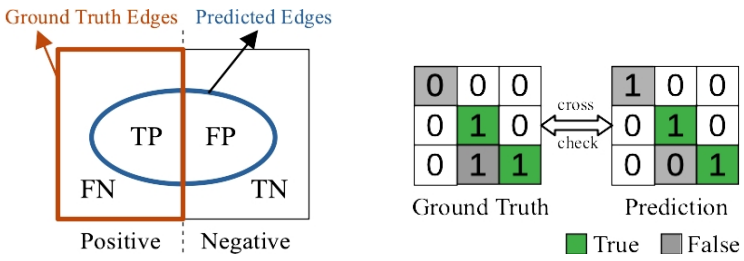
4.4.2.2 Learning Details

For the ablation study on *PASCAL VOC 2012*, we trained DeepLabV3+ from scratch on the combined dataset above with crop size 512^2. We set Learning Rate (LR) as 0.007 for ResNet and 0.07 for ASPP and decoder. To train on MS-COCO, LR was set to 0.01 for ResNet and 0.1 for the others. LR for superpixel network pretrained on BSDS500 was the same as ResNet which was pretrained on ImageNet and is accessible in PyTorch model zoo. We used SGD [Sutskever et al., 2013] with momentum 0.9, weight decay 5e-4, and "poly" scheduler for 60 epochs.

For the joint training, we set LR as 1e-6 for pretrained DeepLabV3+ and superpixel with FCN. LR for the transparent initialization module is 1e-7. These LRs were fixed in the joint training for 20 epochs.

For *ADE20K* and *PASCAL Context*, we fine-tuned the state-of-the-art pretrained ResNeSt101 and superpixel with FCN with crop size 480^2. LR for TI module is 1e-9, and 1e-6 for the others. We fine-tuned for 100 epochs by SGD with momentum 0.9 and weight decay 5e-4. Batch size was decreased from 16 to 8 due to the limited GPU memory.

4.4.2.3 Metrics for Segmentation Edges



(a) F-measure (FM) (b) Performance Ratio (PR)

Figure 4.5: *Metrics for semantic segmentation edges. "T": true, "F": false, "P": positive, "N": negative, "0": non-edge, "1": edge. F-measure is calculated by Eq. (4.15) and performance ratio by Eq. (4.16).*

For semantic segmentation, we used mean Intersection over Union (mIoU) and pixel accuracy [Long et al., 2015]. For segmentation edges, we used performance

ratio [Khaire and Thakur, 2012], in Eq. (4.16), and F-measure [Stutz et al., 2018; Flach and Kull, 2015], in Eq. (4.15). Both are illustrated in Fig. 4.5.

$$\begin{aligned} \text{F-measure:} \quad & FM = \frac{2}{R^{-1} + P^{-1}} \,, \\ \text{Recall Rate:} \quad & R = \frac{TP^{1}}{TP + FN} \,, \\ \text{Precision:} \quad & P = \frac{TP}{TP + FP} \,, \end{aligned} \tag{4.15}$$

$$\text{Performance Ratio:} \quad PR = \frac{\text{True Edges}}{\text{False Edges}} \times 100\% = \frac{TP}{FP + FN} \times 100\% \,. \tag{4.16}$$

4.4.3 Ablation Study

The ablation study was on PASCAL VOC 2012. We first reproduced DeepLabV3+ with ResNet101, resulting in 78.85% mIoU comparable to 78.43% [2]. Applying superpixel network over it by logit consistency increased the mIoU by 0.37%. We then adopted ResNet152 for a qualified baseline with 77.78% on the full size for evaluations.

Applying superpixel contours increased the mIoU from 77.78% to 78.15%, based on which fine-tuning by our transparent initialization further increased it by 0.79% (1.16% to DeepLabV3+). Pretrained on MS-COCO for PASCAL VOC 2012, our proposal has 0.61% increase of mIoU. Note that most edges in the ground truth were neglected in the evaluation, marked white in Fig. 4.6(d).

Again, our goal is to preserve sharp edges aligned with object contours by superpixels. Fig. 4.6 vividly shows the enhanced object edges, especially objects that are highly contrastive to the background, such as birds and human heads.

Table 4.4: *Ablation study: single-scale evaluation on PASCAL VOC 2012. Ours used DeepLabV3+ with ResNet101 and ResNet152 and superpixel net with a 3-layer TI module. "SP": superpixel with logit consistency; "TI": transparent initialization. "mIoU" is on* 512^2 *and full image size. Note that as an addition of each component on its previous version boosts the performance, we do not compare every possible combination of SP, TI, and MS-COCO.*

Manner	Backbone	SP	TI	MS-COCO	mIoU (512^2/full)
DeepLabV3+ [Chen et al., 2018b]	ResNet101	-	-	-	78.85 / 76.47
Ours	ResNet101	✓	-	-	**79.22** / **76.98**
DeepLabV3+ [Chen et al., 2018b]	ResNet152	-	-	-	79.32 / 77.78
Ours	ResNet152	✓	-	-	79.94 / 78.15
	ResNet152	✓	✓	-	**80.46** / **78.94**
DeepLabV3+ [Chen et al., 2018b]	ResNet152	-	-	✓	82.62 / 80.76
Ours	ResNet152	✓	✓	✓	**83.39** / **81.37**

[1] "TP": a predicted edge is an edge, "FP": a predicted edge is not an edge, "FN": a predicted non-edge is actually an edge, and "TN": a predicted non-edge is not an edge.

[2] https://github.com/jfzhang95/pytorch-deeplab-xception.git

Table 4.5: *Multiscale evaluation on ADE20K. Ours used ResNeSt101 and superpixels with a 2-layer TI module.*

Manner	pixAcc.	mIoU
PSPNet [Zhao et al., 2017]	81.39	43.29
EncNet [Zhang et al., 2018b]	81.69	44.65
ResNeSt50 [Zhang et al., 2020]	81.17	45.12
ResNeSt101 [Zhang et al., 2020]	82.07	46.91
Ours[3]	**82.37**	**47.42**

Table 4.6: *Multiscale evaluation on PASCAL Context. Ours used ResNeSt101 and superpixels with a 3-layer TI module.*

Manner	pixAcc.	mIoU
FCN (ResNet50) [Long et al., 2015]	73.40	41.00
EncNet [Zhang et al., 2018b]	80.70	54.10
ResNeSt50 [Zhang et al., 2020]	80.41	53.19
ResNeSt101 [Zhang et al., 2020]	81.91	56.49
Ours[3]	**82.43**	**57.32**

4.4.4 Evaluations

4.4.4.1 Semantic Segmentation

For ADE20K and PASCAL Context, we used the most recent state-of-the-art semantic segmentation network ResNeSt [Zhang et al., 2020] [4]. Its state-of-the-art performance on ADE20K using **ResNeSt200** is 82.45% pixel accuracy and 48.36% mIoU, and 83.06% pixel accuracy and 58.92% mIoU on PASCAL Context using **ResNeSt269**. Since we have only 4 P100 (16 GB) GPUs for our experiments while Zhang et al. [2020] has 64 V100 (16 GB) GPUs, we chose ResNeSt101 instead of ResNeSt200 or ResNeSt269 as our baseline network. Note that it is possible to enhance semantic segmentation edges on those state-of-the-art networks given sufficient GPU memory.

In Table 4.5, our method improves the pixel accuracy by 0.39% and mIoU by 0.61% on ADE20K over the baseline. In Table 4.6, the pixel accuracy is improved by 0.71% and the mIoU by 0.83% over the baseline.

More importantly, visualizations in Fig. 4.7 for ADE20K and Fig. 4.8 for PASCAL Context vividly show the enhanced object edge details. This is the core of our refinement on semantic segmentation with superpixel constraints. For a fair comparison with Zhang et al. [2020], multiscale evaluations with multiscale superpixel maps were used while the superpixel maps in Figs. 4.7-4.8 are single-scale merely for demonstration.

[3]Since Zhang et al. [2020] has 64 GPUs for state-of-the-art ResNeSt training while only 4 GPUs are accessible for ours, we used ResNeSt101 as baseline.

[4]https://github.com/zhanghang1989/PyTorch-Encoding

[5]https://au.mathworks.com/matlabcentral/fileexchange/52205-measures-of-edge-detection from Khaire and Thakur [2012].

[6]https://github.com/jyhjinghwang/SegSort.git

Table 4.7: *Evaluation of semantic segmentation edges on edge areas extended by 1 to 5 pixels*[5]*. Metric numbers are scaled by* $\times 100$*. We evaluate edge-aware SegSort [Hwang et al., 2019] on PASCAL VOC with downloaded supervised results*[6]*. Although our backbone is ResNeSt101 due to limited GPU capacity, ours outperforms both ResNeSt101 and deeper ResNeSt, that is ResNeSt200 and ResNeSt269 on related datasets.*

Dataset	Manner	Performance Ratio ↑					F-Measure ↑				
		@1pix.	@2pix.	@3pix.	@4pix.	@5pix.	@1pix.	@2pix.	@3pix.	@4pix.	@5pix.
PASCAL VOC	ResNet101 [Chen et al., 2018b]	5.19	6.79	7.62	8.00	8.17	9.10	11.51	12.77	13.40	13.73
	ResNet152 [Chen et al., 2018b]	7.10	9.65	11.05	11.51	11.29	11.74	15.27	17.31	18.10	18.00
	SegSort [Hwang et al., 2019]	10.37	14.24	15.62	15.38	14.09	15.77	20.56	22.68	22.78	21.52
	Ours	**12.53**	**18.05**	**19.15**	**17.59**	**15.26**	**18.82**	**25.10**	**26.68**	**25.34**	**22.94**
ADE20K	EncNet [Zhang et al., 2018b]	11.59	13.66	13.26	12.23	11.10	18.03	20.62	20.35	19.22	17.85
	ResNeSt101 [Zhang et al., 2020]	13.60	15.69	14.79	13.40	12.01	20.41	22.92	22.17	20.68	19.03
	ResNeSt200 [Zhang et al., 2020]	14.00	16.24	15.29	13.79	12.30	20.89	23.52	22.73	21.13	19.40
	Ours	**15.64**	**17.51**	**16.01**	**14.20**	**12.56**	**22.84**	**24.99**	**23.61**	**21.67**	**19.75**
PASCAL Context	EncNet [Zhang et al., 2018b]	12.48	15.30	15.00	13.88	12.52	18.65	22.11	22.19	21.15	19.65
	ResNeSt101 [Zhang et al., 2020]	13.52	16.31	15.70	14.38	12.87	19.67	23.11	22.96	21.73	20.09
	ResNeSt269 [Zhang et al., 2020]	13.71	16.48	15.82	14.44	12.88	19.94	23.29	23.07	21.79	20.09
	Ours	**18.08**	**20.61**	**18.66**	**16.42**	**14.32**	**24.58**	**27.67**	**26.28**	**24.14**	**21.89**

4.4.4.2 Semantic Segmentation Edges

Since our refinement of semantic segmentation mainly lies in object edge areas, we used the popular Performance Ratio (PR) [Khaire and Thakur, 2012] and F-measure (FM) [Stutz et al., 2018; Flach and Kull, 2015] to evaluate the enhanced segmentation edges. In Table 4.7, ours outperform the baselines, *i.e.*, ResNet101 on PASCAL VOC 2012 and ResNeSt101 on the others, as well as the most recent state-of-the-art [Zhang et al., 2020], *i.e.*, ResNeSt200 on ADE20K and ResNeSt269 on PASCAL Context, with higher PR and FM. Illustrations of the state-of-the-art edges compared with ours are shown in Fig. 4.1.

4.5 Conclusion

With transparent initialization and sparse encoder introduced in this chapter, the joint learning of the state-of-the-art networks for semantic segmentation and superpixels preserves object edges. The proposed transparent initialization used to fine-tune pretrained models retains the effect of learned parameters through identically mapping the network output to its input at the early learning stage. It is more robust and effective than other parameter initialization methods, such as Xavier and Net2Net. Moreover, the sparse encoder enables the feasibility of efficient matrix multiplications with largely reduced computational complexity. Evaluations on PASCAL VOC 2012, ADE20K, and PASCAL Context datasets validate the effectiveness of our proposal with enhanced object edges. Meanwhile, the quality of our semantic segmentation edges, evaluated by performance ratio and F-measure, is higher than other methods Additionally, transparent initialization is not limited to the joint learning for semantic segmentation but can also be used to initialize additional fully-connected layers for other tasks such as deep knowledge transfer.

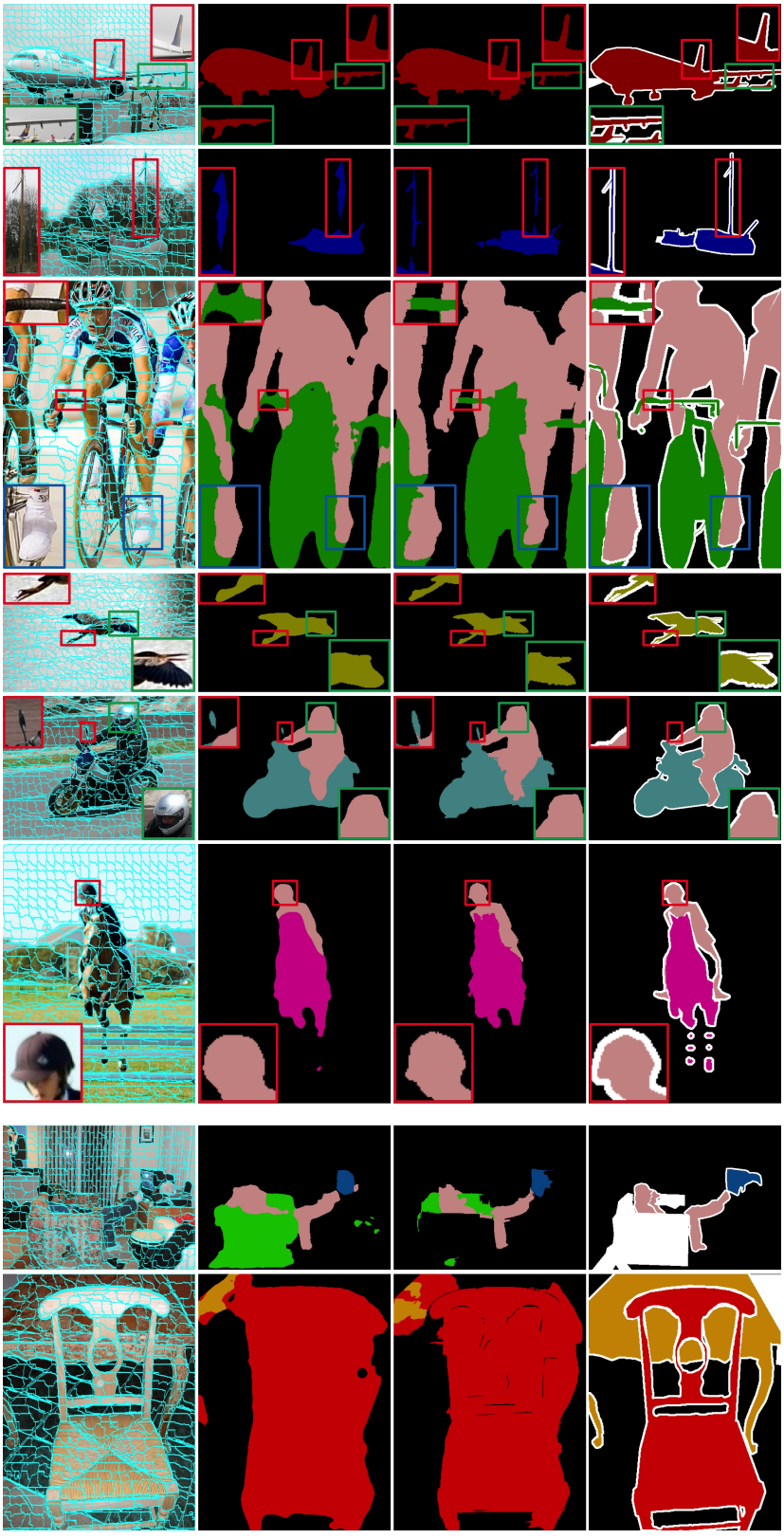


Figure 4.6: *Edge evaluation on PASCAL VOC 2012. The first 6 rows are successful cases; the last 2 rows are failed cases. Superpixel maps are single-scale. Best view by zoom-in.*

Figure 4.7: *Edge evaluation on ADE20K. Superpixel maps are single-scale for the demonstration. Best view by zoom-in.*

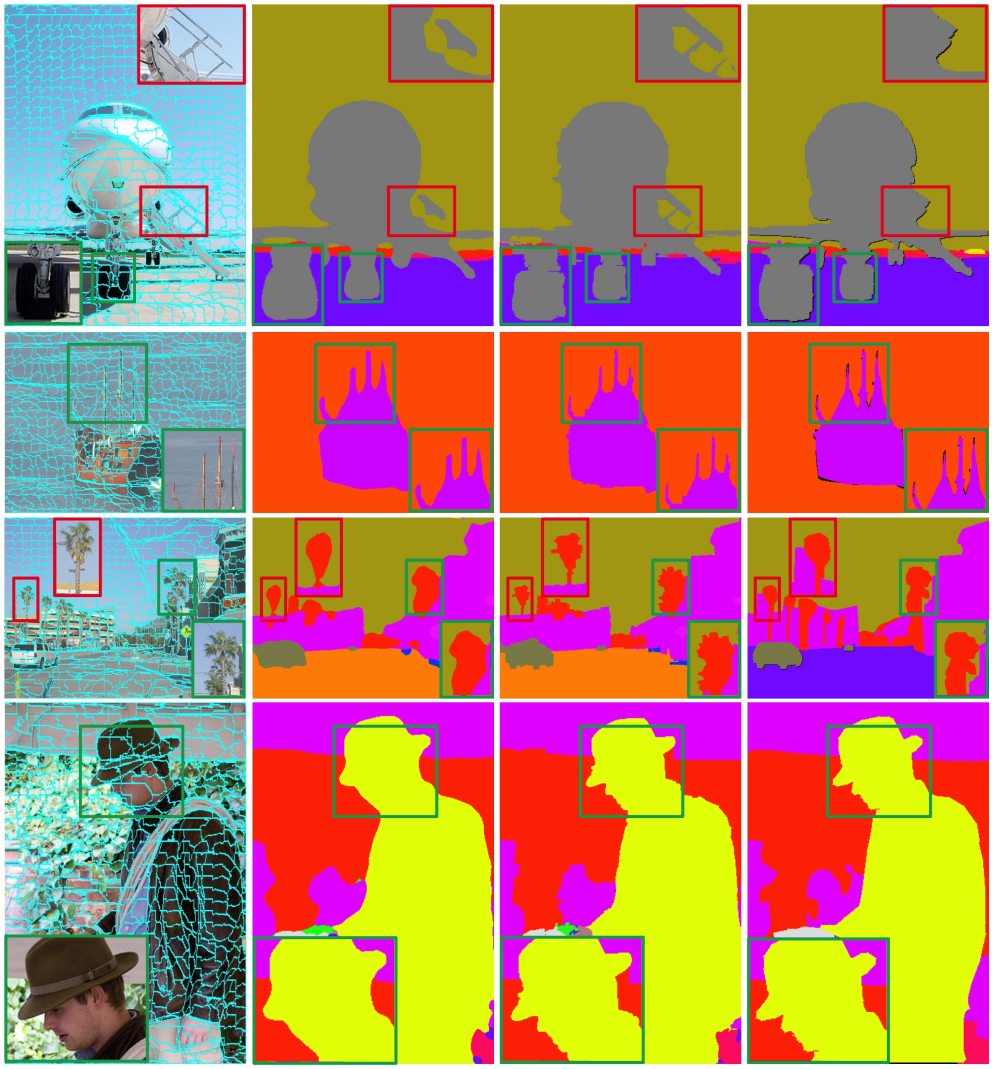


Figure 4.8: *Edge evaluation on PASCAL Context. Superpixel maps are single-scale for the demonstration. Best view by zoom-in.*

4.6 Appendix: A Booklet of Transparent Initialization

The key theoretical analysis of this section is summarized in Sec. 4.3.2. For the completion of the transparent initialization, however, we may need to restate some key parts in Sec. 4.3.2 according to the context below.

4.6.1 Introduction for Warmup

A common technique is to extend an existing network by the addition of further affine layers (classification layers) before the loss layer. These layers may then be trained separately or the whole extended network can be trained. We refer to the original (unextended) network as the base-network, and the network extended by some additional layers as the extended network. We assume that the extension layers consist of fully-connected layers, including offset, followed by an activation layer, such as a ReLU. These extension layers are inserted before the loss-layer.

Assuming that the base layer is trained to attain a low value of the loss, we wish to insert the extension layers into the network without increasing the loss of the network, so that additional training may continue to decrease the value of the loss-function. However, if the extension layers are initialized with random parameters, it will result in an increase in the loss of the network, so that the extension layers, or the whole network needs to be trained from scratch.

4.6.1.1 Simple Transparent Layers

This problem has also been addressed in previous work of Goodfellow [Chen et al., 2016b]. One very simple method to implement a transparent layer is to let the matrix of layer parameters be the identity matrix. This has the disadvantage, however, that all the parameters are either 0 or 1, which fails to introduce sufficient randomness into the system, which would perhaps be desirable. In addition, if the function f is represented by the identity mapping, then many of the parameters are equivalent, perhaps also an undesirable feature. We wish to allow the parameters of the affine transformation to be chosen randomly so as to mix things up a bit.

It is also not clear to do in the case where $f : \mathbb{R}^m \to \mathbb{R}^n$, where $m \neq n$, for then there is no such thing as an identity matrix. We wish to allow the possibility that the output and input of the extension layers are not equal.

4.6.2 Affine Layers

We are interested in layers in a Neural Network (NN) that implement an affine transformation on the input, that is, a linear transformation followed by an offset. For the present, we ignore any non-linear activatation that might be applied to the output of the layer.

The most obvious example of such a layer is a fully-connected layer, and that will be our main focus. However, the same idea could be applied to handle convolu-

tion layers. We raise this possibility and make a few comments on this later. Any development of this idea is left to further work.

An affine layer, in the following discussion, is seen as composed of a linear transform, followed by an offset and then a non-linear activation. The linear transform, followed by offset performs an affine transformation on the data, which will be written as[7]

$$f(\mathbf{x}) = \mathbf{x}\mathbf{A} + \mathbf{b} \ , \tag{4.17}$$

where $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$. In this note, the convention is that vectors, such as $\mathbf{x}$ are row vectors, which is contrary to a convention common in computer vision literature that vectors represent column vectors. The present notation is common in computer graphics literature, however.

This mapping satisfies the condition

$$f(\lambda\mathbf{x}) + f((1-\lambda)\mathbf{x}) = f(\mathbf{x}) \tag{4.18}$$

and it may be represented by matrix multiplication on the homogenized vector $\tilde{x}$, according to

$$\tilde{x} \mapsto \tilde{x}\tilde{\mathbf{A}} \ , \tag{4.19}$$

where the matrix $\tilde{\mathbf{A}}$ is given by

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{A} & \mathbf{0}^\top \\ \mathbf{b} & 1 \end{bmatrix} . \tag{4.20}$$

For future reference, the matrix that performs the inverse affine transformation is equal to

$$\tilde{\mathbf{A}}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0}^\top \\ -\mathbf{b}\mathbf{A}^{-1} & 1 \end{bmatrix} . \tag{4.21}$$

This works if $\mathbf{A}$ is a square matrix and has an inverse. More generally, in the case where $\mathbf{A}$ is of dimension $m \times n$, with $m < n$, it is possible that $\mathbf{A}$ has a right-inverse, which is a matrix, denoted by $\mathbf{A}^R$ such that $\mathbf{A}\mathbf{A}^R = \mathbf{I}$. In this case, a right inverse for $\tilde{\mathbf{A}}$ is given by

$$\tilde{\mathbf{A}}^R = \begin{bmatrix} \mathbf{A}^R & \mathbf{0}^\top \\ -\mathbf{b}\mathbf{A}^R & 1 \end{bmatrix} . \tag{4.22}$$

where $\mathbf{A}^R$ is the right-inverse of matrix $\mathbf{A}$.

An affine transform maps $\mathbb{R}^m$ onto an affine subspace of $\mathbb{R}^n$, and the *rank* of the affine transform may be defined as the dimension of the affine space that is the image of this transform. Such a transformation will be said to be of *full rank* if its rank is equal to m, the dimension of input space.

From here on, we shall use the symbol $\mathtt{A}$ to represent an affine transformation,

[7] The notation in this exposition differs slightly, mainly in choice of fonts. Here, we deonte a matrix by $\mathbf{A}$, an affine transformation by $\mathtt{A}$ and the matrix that expresses an affine transformation in terms of homogeneous coordinates by $\tilde{\mathbf{A}}$. Homogeneous quantities in general (such as homogeneous vectors are denoted with a tilde).

writing $\mathbf{x}\mathtt{A}$. The symbol $\mathtt{A}$ represents the transformation itself, not the matrix (in this case $\tilde{\mathbf{A}}$) that implements it. **Caution:** to reemphasize this, the symbols $\mathbf{A}$ and $\mathtt{A}$ do not represent the same thing. In particular $\mathbf{A}$ is a matrix and $\mathtt{A}$ is an affine transform, which is equivalent to matrix multiplication by a matrix $\mathbf{A}$ and offset by a vector $\mathbf{b}$, so that $\mathbf{x}\mathtt{A}$ is short-hand notation for $\mathbf{x}\mathbf{A} + \mathbf{b}$.

Interlude: right inverse of a matrix. Let $\mathbf{A}$ be an $m \times n$ matrix with $m \leq n$. An $n \times m$ matrix $\mathbf{A}^R$ is the right-inverse of $\mathbf{A}$ if $\mathbf{A}\mathbf{A}^R = \mathbf{I}_{m\times m}$. Such a matrix exists if and only if $\mathbf{A}$ has rank m equal to its row-dimension. This cannot happen if $m > n$, where a right-inverse cannot exist.

In this case, $\mathbf{A}^R$ is given by the formula

$$\mathbf{A}^R = \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)^{-1} , \tag{4.23}$$

where the condition $\mathbf{A}\mathbf{A}^R = \mathbf{I}$ is easily verified. This formula relies on the fact that $\mathbf{A}\mathbf{A}^\top$ has an inverse, which is ensured because the rank of $\mathbf{A}$ is m.

An alternative procedure is to take the Singular Value Decomposition (SVD) $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$, where $\mathbf{V}^\top$ has dimension $m \times n$ and $\mathbf{V}^\top\mathbf{V} = \mathbf{I}_{m\times m}$. The matrix $\mathbf{D}$ (of dimension $m \times m$) is non-singular (since $\mathbf{A}$ has rank m). Then, the right-inverse is given by

$$\mathbf{A}^R = \mathbf{V}\mathbf{D}^{-1}\mathbf{U}^\top \tag{4.24}$$

as is easily verified.

4.6.3 Sequences of Layers without Activation

For the time being, we consider the unrealistic case that the activation layer is missing from the affine layers. In this case, each affine layer performs an affine transformation. A sequence of transformations $\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_k$ may be applied to an input $\mathbf{x}$, giving

$$\mathbf{x} \mapsto \mathbf{x}\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_k . \tag{4.25}$$

If we choose $\mathtt{A}_k$ to be a right-inverse of $\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_{k-1}$ then we have

$$\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_k = \mathtt{A}_1\ldots\mathtt{A}_{k-1}(\mathtt{A}_1\ldots\mathtt{A}_{k-1})^R = I , \tag{4.26}$$

the identity mapping.

The condition for this right inverse to exist is that $\mathtt{A}_1\mathtt{A}_2\ldots\mathtt{A}_{n-1}$ should be of full rank. A necessary condition for this to happen is as follow. Let $\mathtt{A}_i$ represent an affine transformation $\mathtt{A}_i : \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}$. The first and last spaces in this sequence are $\mathbb{R}^{m_0}$ and $\mathbb{R}^{m_k}$, so $m_0 = m$ and $m_k = n$. Then $\mathtt{A}_1\ldots\mathtt{A}_{k-1}$ has a right inverse only if $m_i \geq m_0$ for all $i = 1,\ldots,k-1$. In other words, a necessary condition for the right inverse to exist is that the dimensions of all the intermediate spaces $\mathbb{R}^{m_1},\ldots,\mathbb{R}^{m_{k-1}}$ should be at least equal to m_0. It is not hard to see that *generically*, this is also a sufficient condition, meaning that it is true for almost all sequences of transformations.[8]

[8]The terms *almost all, almost always, etc.* are intended in the standard mathematical sense, meaning that the set of cases for which the relevant condition fails to be true has measure or probability zero.

Without being too formal here, we can state that this condition will hold *generically* if all the transformations are chosen at random.

Theorem 2 *Let a sequence of affine transformations* $\mathtt{A}_i : \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}$*, for* $i = 1, \ldots, k-1$ *be chosen at random. Then* $\mathtt{A}_1 \ldots \mathtt{A}_{k-1}$ *almost surely has a right inverse if and only if* $m_i \geq m_0$ *for* $i = 1, \ldots, k-1$.

For practical purposes it is safe to proceed as if this result holds always (and not just almost always), since the probability that $\mathtt{A}_1 \ldots \mathtt{A}_{k-1}$ does not have an inverse is vanishingly small (in fact zero).

Therefore, the strategy for selecting a set of parameters for a sequence of affine layers may be described. For the sequence of layers to represent an identity transform, we need that the input and output space have the same dimension, namely $m_0 = m_k$. Then

1. Select intermediate dimensions $m_1, \ldots, m_{k-1}$ such that $m_i \geq m_0$ for all $i = 1, \ldots, k-1$.

2. Define random affine transforms $\mathtt{A}_i$ by selecting the entries of matrices $\mathbf{A}_i$ and vectors $\mathbf{b}_i$ randomly, using a suitable random number generator, for instance by a zero-mean normal (Gaussian) distribution.

3. Compute the composition $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1}$ by matrix multiplication, and take its right-inverse.

4. Set $\mathtt{A}_k$ to equal $(\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1})^R$.

The resulting product $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_k$ is the identity affine transformation.

4.6.4 Gaussian Random Matrices

Since we are using matrices initialized from a normal distribution, we give some properties of such matrices. The purpose is to ensure that the concatenation of affine transformations does not cause explosion of the entries of the product $\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1}$. If this is not done properly, then this product will have very large entries and so will any value of $x_0\mathtt{A}_1\mathtt{A}_2 \ldots \mathtt{A}_{k-1}$.

The solution is to make the matrices $\mathbf{A}_i$ appearing as the linear-transform part of $\mathbf{A}_i$ as close to orthogonal as possible. As an additional advantage, the entries of $\mathtt{A}_k = (\mathtt{A}_1 \ldots \mathtt{A}_{k-1})^R$ will be approximately of the same order as the entries of each of the other $\mathtt{A}_i$. It will turn out that the entries of each $\mathtt{A}_i$ should be chosen from a Gaussian distribution with variance $1/m$ where m is its row-dimension.

For instance the set of $m \times n$, with $m \leq n$ that do not have full rank is a set of measure 0 in the set of all such matrices, so one can state that almost all such matrices have full rank.

As another example, a matrix chosen at random (which implies that numbers are chosen from some probability distribution such as a normal distribution) will be of full rank with probability 1, hence almost always.

We are assuming here that the dimensions of the matrix are large. In addition, if the matrix is not square, by saying it is orthogonal we mean that both its rows (and similarly it columns) are orthogonal vectors, all of the same length (a different length for the row and column vectors, of course). One may mistakenly assume that by randomly selecting each entry of a matrix randomly one obtains a random matrix in some vague sense. In reality, one obtains an (nearly) orthogonal matrix. To see this, we first see that two random vectors from distribution $\mathcal{D}$ are almost orthogonal. Let X and Y be two *i.i.d* random variables in $\mathcal{N}(0, \sigma^2)$, we have

$$E[XY] = E[X]E[Y] = 0\,, \quad E[(XY)^2] = E[X^2]E[Y^2]\,. \tag{4.27}$$

Then,

$$\begin{aligned} \text{var}(XY) = E[(XY)^2] - E^2[XY] &= E[X^2]\ E[Y^2] \\ &= \left(E[X^2] - E^2[X]\right)\ \left(E[Y^2] - E^2[Y]\right) \\ &= \text{var}(X)\ \text{var}(Y)\,. \end{aligned} \tag{4.28}$$

Now, given a column vector $\mathbf{v} \in \mathbb{R}^m$ with entries from distribution $\mathcal{D}$, we have

$$\text{var}\left(\sum_{i=1}^{m} X_i Y_i\right) = m\text{var}\left(X_i Y_i\right) = m\text{var}^2(X) = m\sigma^4\,, \tag{4.29}$$

where X_i and Y_i are independent entries in two of such column vectors respectively. Then, the *expected squared length* of such a vector $\mathbf{v}$ is $mE[X^2] = m\sigma^2$. If we choose $\sigma^2 = 1/m$, $\mathbf{v}$ has the expected squared length equal to 1. This satisfies the attribute of an orthogonal matrix, $\mathbf{v}^T\mathbf{v} = \mathbf{1}$. Then, Eq. (4.28) follows

$$\text{var}\left(\sum_{i=1}^{m} X_i Y_i\right) = m\sigma^4 = \frac{1}{m}\,. \tag{4.30}$$

Hence, two m-length vectors randomly chosen from $\mathcal{N}(0, 1/m)$ will have expected squared length 1 and expected inner-product 0 by Eq. (4.27) with variance of the inner product as $1/m$. Meanwhile, the *variance of the square length* of the vector is

$$\text{var}\left(\sum_{i=1}^{m} X_i^2\right) = m\text{var}(X^2) = \frac{2}{m}\,, \tag{4.31}$$

where $\text{var}(X^2) = 2\sigma^4$ is obtained by $E(X^4) - E^2(X)$ with $E^2(X^2) = \sigma^4$ and

$$\begin{aligned} E(X^4) =& \frac{\int X^4 \exp\left(-\frac{X^2}{2\sigma^2}\right) dX}{\int \exp\left(-\frac{X^2}{2\sigma^2}\right) dX} \\ =& \frac{-\sigma^2 X^3 \exp\left(-\frac{X^2}{2\sigma^2}\right)\Big|_{-\infty}^{+\infty} + \sigma^2 \int \exp\left(-\frac{X^2}{2\sigma^2}\right) dX^3}{\int \exp\left(-\frac{X^2}{2\sigma^2}\right) dX} \\ =& \frac{3\sigma^2 \int X^2 \exp\left(-\frac{X^2}{2\sigma^2}\right) dX}{\int \exp\left(-\frac{X^2}{2\sigma^2}\right) dX} \\ =& \frac{3\sigma^2(-\sigma^2)\left(X \exp\left(-\frac{X^2}{2\sigma^2}\right)\Big|_{-\infty}^{+\infty} - \int \exp\left(-\frac{X^2}{2\sigma^2}\right) dX\right)}{\int \exp\left(-\frac{X^2}{2\sigma^2}\right) dX} = 3\sigma^4 . \end{aligned} \tag{4.32}$$

To this end, it shows

Theorem 3 *If entries of an $m \times n$ matrix are chosen from a zero-mean distribution with variance $\sigma^2 = 1/m$, then the column vectors have expected squared length 1 with variance $2/m$ and the expected inner product of each two columns is 0 with variance $1/m$.*

As m increases, the matrix approximates more and more an orthogonal matrix. Corresponding to Fig. 4.4, weights A_i and bias b_i can be initialized by

1. Choose the dimension of i-th layer filter as $m_{i-1} \times m_i$, for $i = 1, ..., k-1$, where $m_i \geq m_0 = m_k$.
2. Define a random affine transformation $\mathtt{A}_i$ by selecting its weight matrix $\mathbf{A}_i$ from a $\mathcal{N}(0, 1/m_i)$ Gaussian distribution and its bias vector $\mathbf{b}_i$ from a $\mathcal{N}(0, 1)$ Gaussian distribution.
3. Initialize $\mathtt{A}_k$ with the right inverse of $\mathtt{A}_1\mathtt{A}_2...\mathtt{A}_{k-1}$.

Again, **note** that $\mathtt{A}_i$ is an affine transformation (also used as a layer) where $\mathbf{A}_i$ is the weight matrix of $\mathtt{A}_i$. Then, the above initialization will lead to an identity mapping as $\mathtt{A}_1\mathtt{A}_2...\mathtt{A}_k = \mathbf{1}$.

4.6.5 Getting Past the Activation Layer

We concentrate first on the case where each affine transform is followed by a ReLU activation. To do this, we will need to double the size of the intermediate layers, as will be seen next. We assume that affine transforms $\mathtt{A}_i$ are given, let $\mathbf{x} = \mathbf{x}_0$ be the input of $\mathtt{A}_1$ and define $\mathbf{x}_i = \mathbf{x}_{i-1}\mathtt{A}_i$ for $i = 1, \ldots, k$, the output of the i-th affine transform.

In the course of the following discussion, we shall be defining new affine transforms $\mathtt{A}'_i$ and layers represented by $\mathtt{A}'_i\sigma$, namely an affine transform followed by an

activation σ. Once more let $\mathbf{x}'_0$ be the input of the first layer, $\mathbf{x}'_0 = \mathbf{x}_0$, and define $\mathbf{x}'_i = \mathbf{x}'_{i-1}\mathtt{A}'_i\sigma$, the result of the affine transformation followed by the activation. Quantities without primes ($\mathtt{A}_i$ and $\mathbf{x}'_i$) belong to the original sequence of affine transforms, whereas those with primes ($\mathtt{A}'_i$ and $\mathbf{x}'_i$) belong to the sequence, with activation, being constructed).

The first layer will be modified as follows. Suppose that $\mathtt{A}_1$ is represented in matrix form as

$$\mathbf{x}_0\mathtt{A}_1 = (\mathbf{x}_0, 1)\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{b}_1 \end{bmatrix} = \mathbf{x}_0\mathbf{A}_1 + \mathbf{b}_1 \ . \tag{4.33}$$

This is replaced by

$$\begin{aligned} \mathbf{x}_0\mathtt{A}'_1 &= (\mathbf{x}_0, 1)\begin{bmatrix} \mathbf{A}_1 & -\mathbf{A}_1 \\ \mathbf{b}_1 & -\mathbf{b}_1 \end{bmatrix} \\ &= \mathbf{x}_0\,[\mathbf{A}_1 \ | -\mathbf{A}_1\,] + (\,\mathbf{b}_1, -\mathbf{b}_1) \\ &= (\ \mathbf{x}_0\mathbf{A}_1 + \mathbf{b}_1, -\mathbf{x}_0\mathbf{A}_1 - \mathbf{b}_1\ ) \\ &= (\mathbf{x}_0\mathtt{A}_1, -\mathbf{x}_0\mathtt{A}_1)\ . \end{aligned} \tag{4.34}$$

In other words, the $m_0 \times m_1$ matrix $\mathbf{A}$ is replaced by the $m_0 \times 2m_1$ matrix $\mathbf{A}'_1 = [\,\mathbf{A}_1 \ | -\mathbf{A}_1\,]$, and $\mathbf{b}_1$ is replaced by the $2m_1$ dimensional vector $\mathbf{b}'_1 = (\mathbf{b}_1, -\mathbf{b}_1)$.[9]

Now, when this is passed through an activation layer, represented by the activation function σ (ReLU), the result is

$$\mathbf{x}_0\mathtt{A}'_1\sigma = (\ \mathbf{x}_0\mathtt{A}_1\sigma\ , (-\mathbf{x}_0\mathtt{A}_1)\sigma) = \mathbf{x}'_1\ . \tag{4.35}$$

This is the output of our modified affine layer with activation. The point to note here is the identity $\mathbf{v}\sigma - (-\mathbf{v})\sigma = \mathbf{v}$. This gives

$$(\mathbf{x}_0\mathtt{A}_1)\sigma - (-\mathbf{x}_0\mathtt{A}_1)\sigma = \mathbf{x}_0\mathtt{A}_1\ . \tag{4.36}$$

Thus, by subtracting the two halves of $\mathbf{x}_0\mathtt{A}'_1\sigma$, one arrives back at the simple affine transform $\mathbf{x}_0\mathtt{A}_1$.

We simplify the notation as follows. Let $\mathbf{x}_i^+$ and $\mathbf{x}_i^-$ be defined as

$$\mathbf{x}_i^+ = \mathbf{x}_i\sigma\ , \tag{4.37a}$$
$$\mathbf{x}_i^- = (-\mathbf{x}_i)\sigma\ , \tag{4.37b}$$

which are the positive and negative parts of $\mathbf{x}_i$ respectively, satisfying $\mathbf{x}_i = \mathbf{x}_i^+ - \mathbf{x}_i^-$.

[9] **Important note:** The affine transformation $\mathtt{A}'$ is defined

$$\begin{bmatrix} \mathbf{A}_1 & -\mathbf{A}_1 \\ \mathbf{b}_1 & -\mathbf{b}_1 \end{bmatrix}\ .$$

This will output a vector of the form $(\mathbf{x}_0\mathtt{A}_1, -\mathbf{x}_0\mathtt{A}_1)$ with two parts that are negatives of each other. It defines the **initial** form of the affine transform only. During training, all entries of the above matrix are free to vary (and will) independently, and its output will not maintain this symmetric form. This applies to all the affine transforms $\mathtt{A}'_i$ that will be defined.

Then, we see that

$$\mathbf{x}_1' = (\mathbf{x}_1^+, \mathbf{x}_1^-) \,. \tag{4.38}$$

Now, the first thing to do at the beginning of the next layer is to subtract the two parts of the previous output, illustrated by

$$\mathbf{x}_1' \begin{bmatrix} \mathtt{I} \\ -\mathtt{I} \end{bmatrix} = (\mathbf{x}_1^+, \mathbf{x}_1^-) \begin{bmatrix} \mathtt{I} \\ -\mathtt{I} \end{bmatrix} = \mathbf{x}_1 \,. \tag{4.39}$$

This is followed by the same trick of separating the positive and negative parts as before. The affine part (without activation) of the second layer is then

$$\begin{aligned} \mathbf{x}_1' \mathtt{A}_2' &= \mathbf{x}_1' \begin{bmatrix} \mathtt{I} \\ -\mathtt{I} \end{bmatrix} [\mathbf{A}_2 \mid -\mathbf{A}_2] + (\mathbf{b}_2, -\mathbf{b}_2) \\ &= \mathbf{x}_1' \begin{bmatrix} \mathbf{A}_2 & -\mathbf{A}_2 \\ -\mathbf{A}_2 & \mathbf{A}_2 \end{bmatrix} + (\mathbf{b}_2, -\mathbf{b}_2) \\ &= \mathbf{x}_1' \mathbf{A}_2' + \mathbf{b}_2' \,, \end{aligned} \tag{4.40}$$

where $\mathbf{A}_2'$, $\mathbf{b}_2'$ and $\mathtt{A}_2'$ are defined by this equation. Noting Eq. (4.39), we arrive at

$$\mathbf{x}_1' \mathtt{A}_2' = (\mathbf{x}_1 \mathtt{A}_2, -\mathbf{x}_1 \mathtt{A}_2) \,. \tag{4.41}$$

and so

$$\mathbf{x}_1' \mathtt{A}_2' \sigma = \mathbf{x}_2' = (\mathbf{x}_1 \mathtt{A}_2 \sigma, (-\mathbf{x}_1 \mathtt{A}_2)\sigma) = (\mathbf{x}_2^+, \mathbf{x}_2^-) \,. \tag{4.42}$$

Combining this with Eq. (4.38) gives

$$\mathbf{x}_0 \, \mathtt{A}_1' \sigma \, \mathtt{A}_2' \sigma = (\mathbf{x}_2^+, \mathbf{x}_2^-) \,. \tag{4.43}$$

Continuing to define layers in this way, according to Eq. (4.40), for layers up to layer $k-1$, gives that

$$\mathbf{x}_0 \, \mathtt{A}_1' \sigma \, \mathtt{A}_2' \sigma \ldots \mathtt{A}_{k-1}' \sigma = (\mathbf{x}_{k-1}^+, \mathbf{x}_{k-1}^-) = \mathbf{x}_{k-1}' \,. \tag{4.44}$$

Finally, we define the last layer k by

$$\begin{aligned} \mathbf{x}_{k-1}' \mathtt{A}_k' &= (\mathbf{x}_{k-1}^+, \mathbf{x}_{k-1}^-) \begin{bmatrix} \mathbf{A}_k \\ -\mathbf{A}_k \end{bmatrix} + \mathbf{b}_k \\ &= (\mathbf{x}_{k-1}^+ - \mathbf{x}_{k-1}^-)\mathbf{A}_k + \mathbf{b}_k \\ &= \mathbf{x}_{k-1}\mathbf{A}_k + \mathbf{b}_k \\ &= \mathbf{x}_{k-1}\mathtt{A}_k \\ &= \mathbf{x}_k \,. \end{aligned} \tag{4.45}$$

Putting those defined $\mathtt{A}_i$ all together gives

$$\mathbf{x}_0 \, \mathtt{A}_1' \sigma \, \mathtt{A}_2 \sigma \ldots \mathtt{A}_k' = \mathbf{x}_k = \mathbf{x}_0 \,, \tag{4.46}$$

where the final step is because the sequence of affine transforms $\mathtt{A}_1 \dots \mathtt{A}_k$ is chosen to be the identity map, so $\mathbf{x}_k = \mathbf{x}_0 \mathtt{A}_1 \dots \mathtt{A}_k = \mathbf{x}_0$.

4.6.6 Exploration of Layer Initialization Effects

In Table 4.2, we compared the effectiveness of our transparent initialization with others, that is random, Xavier [Glorot and Bengio, 2010], and Net2Net [Chen et al., 2016b]. Transparent initialization can 100% recover the input data from the layers output with a high initialization rate and a small around-off error shown in Table 3 and supports non-square filters, by saying filters we mean 1^2 kernel-size convolutional layers or fully-connected layers. In contrast, random and Xavier initialization are ineffective to recover the input data from the output while Net2Net is only effective for non-negative data and is infeasible for non-square filters. Despite these quantitative experiments to analyse the attributes of transparent initialization, we directly compare their effects on joint learning pretrained networks of semantic segmentation and superpixels.

In Fig. 4.9, for the task of semantic segmentation, the cross-entropy loss function highly relies on the maximum values of logits, the network outputs, along the label dimension. In Fig. 3.9(a)-3.9(c), input data of the add-on FC layers contains positive and non-positive values while in Fig. 3.9(d) input data is non-positive by subtracting the maximum value along the label dimension. This is to explore the data recovering ability for different numerical space (non-negative and non-positive in our case).

The final mIoUs via random and Xavier initialization are nearly 0% as they totally interrupt the learned parameters, leading to a high loss. To recover **non-negative** data, in Fig. 3.9(c), Net2Net and ours have similar loss decreasing tendency. However, as the epoch increases, the loss by ours becomes lower than Net2Net because the gradients by transparent initialization are dense for parameter updates compared with the sparse ones by Net2Net. Furthermore, in Fig. 3.9(d), Net2Net is unable to recover **negative** values, as shown in Table 4.2, resulting in a high loss. In Fig. 3.9(c), the mIoU by Net2Net and ours are similar, both nearly 83.3%. Corresponding to Fig. 3.9(d), however, the mIoU by Net2Net is nearly 0% while ours is still nearly 83.3% since it is invariant to numerical space.

4.6.7 Activation Functions

Again, note that the notation of activation function $\sigma(x)$ has the same meaning as $x\sigma$ for a simplicity of sequence layers. For transparent initialization, any non-linear activation functions satisfying Eq. (4.47) can recover the input from the output.

$$\sigma(x) - \sigma(-x) = cx \,, \tag{4.47}$$

where c is a non-zero constant. This can be expressed by $xD\sigma S$ with D (duplicate) and S (subtract).

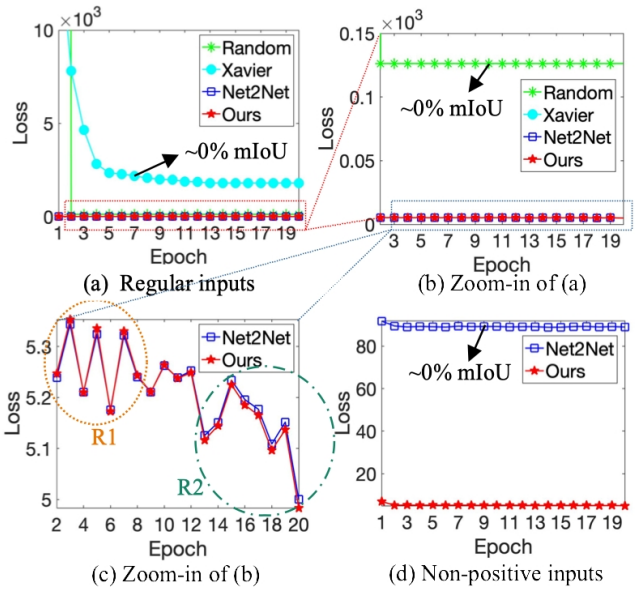


Figure 4.9: *Training loss with different layer initialization methods. "regular": data containing positive, negative, and zero values. Note that in our semantic segmentation task, a low loss is determined by a high (mostly positive) logit from the NN along the label dimension. The joint learning is for 21-label semantic segmentation using pretrained DeepLabV3+ [Chen et al., 2018b] and superpixel with FCN [Yang et al., 2020] networks with add-on 3 Fully-Connected (FC) layers. Here, in (a) and (b), random and Xavier initialization on these add-on FC layers lead to a high loss, and thus, decreasing the mIoU from ∼80% to ∼0%. It is obvious that they cannot work in our case, since they are unable to recover the pretrained results as shown in Table 4.2. On the other hand, Xavier initialization may have a worse local minima than random initialization due to the interrupted output values. In contrast, for regular data containing positive, negative, and zero values, both Net2Net and our transparent initialization have similar good effectiveness. Because, as mentioned before, negative logit values (hardly to be the highest) may not have significant effects on the joint learning as in our case positive logits always have high softmax values. So, the negative values of the input data can even be zero-out by ReLU or Net2Net initialization. For non-positive inputs in (d), however, Net2Net is unable to recover negative values, leading to ∼0% mIoU while ours has the same low loss as it is in (c). Additionally, although Net2Net and ours in (c) have similar losses, both achieving ∼83.3% mIoU, we note that the loss of ours starts to be less than Net2Net, shown in R2. In R1, ours has a high loss due to the effects of dense gradients that change network parameters more dramatically than Net2Net. This is expected as the transparent initialization should have a strong learning ability than Net2Net. Overall, ours outperforms random and Xavier initialization, both regular and non-positive data, and Net2Net for non-positive data. For regular data, ours tends to have a smaller loss than Net2Net due to its strong learning ability with dense gradients. More details are in Sec. 4.6.6.*

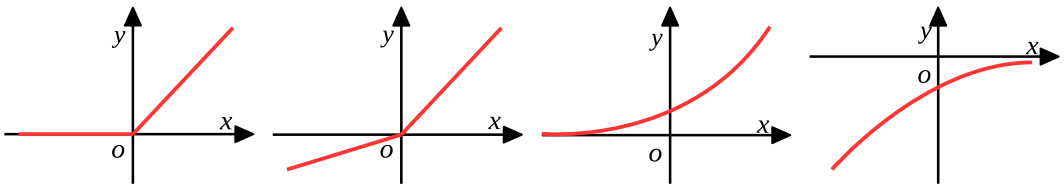
In Fig. 4.10, we give 4 examples with corresponding definitions:

$$\text{ReLU:} \quad y(x) = \begin{cases} x & \text{if } x \geq 0\,, \\ 0 & \text{otherwise}\,, \end{cases} \tag{4.48a}$$

$$\text{LeakyReLU:} \quad y(x) = \begin{cases} x & \text{if } x \geq 0\,, \\ \delta x & \text{otherwise}\,, \end{cases} \tag{4.48b}$$

$$\text{SoftReLU:} \quad y(x) = \log\left(1 + e^{x}\right)\,, \tag{4.48c}$$

$$\text{LogSigmoid:} \quad y(x) = \log\left(\frac{1}{1 + e^{-x}}\right)\,. \tag{4.48d}$$



(a) ReLU (b) LeakyReLU (c) SoftReLU (d) LogSigmoid

Figure 4.10: *Examples of non-linear active functions for transparent initialization.*

It is easy to verify Eq. (4.47) for those activation functions. Given the activation function as LogSigmoid by $\sigma(x) = \log(1/(1+e^{-x}))$, for instance, it follows

$$\begin{aligned} &\log\left(\frac{1}{1+e^{-x}}\right) - \log\left(\frac{1}{1+e^{x}}\right) \\ =&\log\left(\frac{1+e^{x}}{1+e^{-x}}\right) = \log(e^{x}) = x\ . \end{aligned} \tag{4.49}$$

Meanwhile, with c given in Eq. (4.47), $xD\sigma S$ should be $xD\sigma S/c$, where $c = 1/\delta$ is for LeakyReLU and $c = 1$ for the others. To be more general, this property holds for any functions of the form of $f(x) = cx + s(x)$, where $s(x)$ is an even function.

For other activations σ such as the sigmoid function, or arctangent, or hyperbolic tangent, it does not work directly. The trick is to observe that for the ReLU function, $(-x)\sigma = x\sigma - x$. This allows us to rewrite the sequence of operations

$$x \overset{\sigma}{\mapsto} (x\sigma, (-x)\sigma) = (x_1, x_2) \mapsto x_1 - x_2 = x\ ,$$

where σ is the ReLU function, as

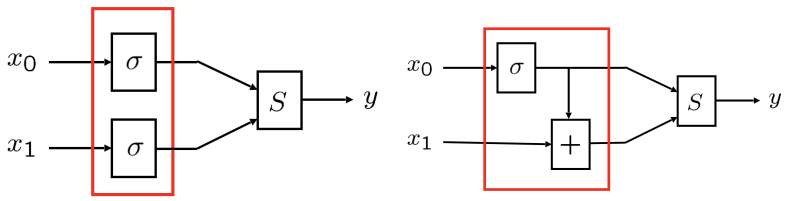
$$x \overset{\sigma}{\mapsto} (x\sigma, x\sigma - x) = (x_1, x_2) \mapsto x_1 - x_2 = x\ .$$

But now this also holds for any activation σ, ReLU or not. This requires a slight change to the architecture of the transparent affine layer. Instead of outputting $(x\sigma, (-x)\sigma)$, it must output $(x\sigma, x\sigma - x)$. This is shown in Fig. 4.11.

4.6.8 Extension to Convolutional Layers

A common practice in experimenting with modifications to neural network architectures is to add additional convolutional layers. If a base network is already trained to achieve minimal loss, then the addition of extra convolutional layers can perturb the loss, and require the network to retrain again to achieve a low loss. The loss after addition of the new layers may not be as low as the loss of the base network. However, by initializing the extra add-on layers to implement an identity transformation will ensure that the loss achieved (on the training set) by the modified network cannot be worse than that of the base network.

This section gives some ideas on the addition of transparent convolutional layers



(a) For $y = (x_0\sigma, x_1\sigma)\, S$ (b) For $y = (x_0\sigma, x_0\sigma + x_1)S$

Figure 4.11: *Two different forms of layer activation (the part in the red box) top be used as see-through activations for transparent layers. Here, σ represents any function, for instance, a commonly used non-linear function such as sigmoid, ReLU or hyperbolic tangent, and the block marked S carries out some operation on the two inputs. Thus, (a) implements $y = (x_0\sigma, x_1\sigma)\, S$ and (b) represents $y = (x_0\sigma, x_0\sigma + x_1)S$. At initialization, $x_0 = -x_1 = x$ and $(a, b)S = a - b$. In this case, the output is $y = x$ in both case. During the training of the network, x_0 and x_1 will diverge, and S will also learn to carry out a different operation, so networks will behave differently. However, using (b) to implement the affine layer activation will provide a transparent initialization, whatever function σ is chosen.*

with pseudo-random initialization that could be used for this purpose. We have not explored this topic in any great depth, leaving it rather to be the subject of future work, and is somewhat speculative at present.

A convolutional layer as usually implemented is an example of an affine layer, since without any non-linear activation it implements a linear transformation on the input and then adds bias. However, unlike fully-connected layers, which are essentially a matrix multiplication, for which an inverse (or right-inverse) can be easily found, convolution is not conveniently represented by matrix multiplication and is less easily inverted.

It is clear that convolution, being linear, can be represented as matrix multiplication on a vectorized form of the input, but this will be a sparse matrix multiplication, and not easily inverted, at least by convolutional layers. By analogy with the method implemented for fully-connected layers, we require that a final convolution will invert the effect of a sequence of previous pseudo-random convolutions.

Not all convolutions are exactly invertible. In fact (ignoring edge effects) a convolution will be invertible by another convolution if and only if its Fourier transform is everywhere non-zero. However, with this caveat, it is possible to find inverse convolutions.

Rather than carrying out a string of convolutions, followed by a single convolution to undo the previous ones, a better strategy may be to add convolutions in pairs such as a high-pass and low-pass filter that cancel each other out. Exploring this topic will be the subject of further work.

Chapter 5

3D CNN Pruning at Initialization

In this chapter, we propose Resource Aware Neuron Pruning (RANP) for but not limited to 3D CNNs to largely reduce the high resource requirements, including GPU memory and FLOPs in CNN training. The weighting and reweighting strategies in RANP keep a balance of retained neurons in each layer and further reduce the consumption of a specific resource, either GPU memory or FLOPs, by adjusting the layerwise mean values of neuron importance according to the layerwise resource consumption. This method achieves 50%-95% FLOPs reduction and 35%-80% memory reduction in our experiments of two 3D semantic segmentation tasks, two video classification tasks, and a stereo matching task.

5.1 Motivation

3D image analysis is important in various real-world applications including scene understanding [Zhang et al., 2014, 2018a], object recognition [Ji et al., 2013; Hou et al., 2019], medical image analysis [Cicek et al., 2016; Zanjani et al., 2019; Kleesiek et al., 2016], and video action recognition [Karpathy et al., 2014; Simonyan and Zisserman, 2014]. Typically, sparse 3D data can be represented by using point clouds [Yi et al., 2016] whereas volumetric representation is required by dense 3D data which arises in domains such as medical imaging [Menze et al., 2015a] and video segmentation and classification [Zhang et al., 2014; Karpathy et al., 2014; Simonyan and Zisserman, 2014]. While efficient neural network architectures can be designed for sparse point cloud data [Graham et al., 2018; Qi et al., 2017], conventional dense 3D Convolutional Neural Network (CNN) is required to represent the volumetric data. Such 3D CNNs are computationally expensive with excessive memory requirements for large-scale 3D tasks. Therefore, it is highly desirable to reduce the memory and FLOPs required to train 3D CNNs while maintaining the accuracy. This will not only enable large-scale applications but also 3D CNN training on resource-limited devices.

Network pruning is a prominent approach to compress a neural network by reducing the number of parameters or the number of neurons in each layer [Guo et al., 2016; Dong et al., 2017; He et al., 2017; Han et al., 2015]. However, most of the network pruning methods aim at 2D CNNs while pruning 3D CNNs is largely unexplored. This is mainly because pruning is typically targeted at reducing the test-time re-

source requirements while computational requirements of training time are as large as (if not more than) the unpruned network. Such pruning schemes are not suitable for 3D CNNs with dense volumetric data where training-time resource requirement is prohibitively large.

In this work, we introduce a Resource[1] Aware Neuron Pruning (RANP) that *prunes 3D CNNs at initialization*. Our method is inspired by, but superior to, SNIP [Lee et al., 2019] which prunes redundant parameters of a network at initialization and evaluates the method on small scale 2D CNNs for image classification. With the same characteristics of effectively pruning at initialization without requiring large computational resources, RANP yields better-pruned networks compared to SNIP by removing neurons that largely contribute to the high resource requirement. In our experiments on video classification and more challenging 3D semantic segmentation, with minimal accuracy loss, RANP yields 50%-95% reduction in FLOPs and 35%-80% reduction in memory while only 5%-51% reduction in FLOPs and 1%-17% reduction in memory are achieved by SNIP NP.

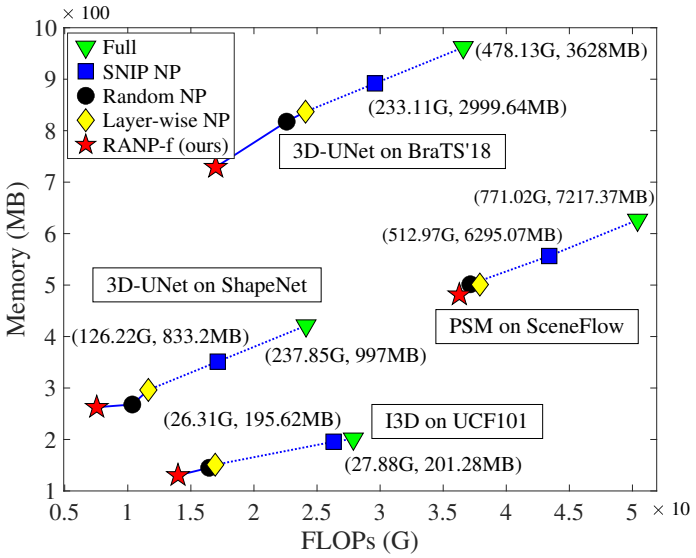


Figure 5.1: *Comparison of neuron pruning methods with the best results at bottom-left. Values of "PSM on SceneFlow" are divided by 10 for visualization. "Full" and "SNIP NP" are not drawn by scale but with their FLOPs (G) and memory (MB) values next to the markers. Our RANP-f performs best with large resource reductions while maintaining the accuracy. More details are in Table 5.2.*

The main idea of RANP is to prune based on a *neuron importance* criterion analogous to the connection sensitivity in SNIP. Note that, pruning based on such a simple criterion as SNIP has the risk of pruning the whole layer(s) at extreme sparsity levels especially on large networks [Lee et al., 2020]. Even though an orthogonal initializa-

[1]We concretely define "resource" as FLoating Point Operations per second (FLOPs) and memory required by hidden layers in a single forward passing.

tion that ensures layer-wise dynamical isometry is sufficient to mitigate this issue for parameter pruning on 2D CNNs [Lee et al., 2020], it is unclear if this could be directly applied to neuron pruning on 3D CNNs. To tackle this and improve pruning, we introduce a *resource aware reweighting scheme* that first balances the mean value of neuron importance in each layer and then reweights the neuron importance based on the resource consumption of each neuron. As evidenced by our experiments, such a reweighting scheme is crucial to obtain large reductions in memory and FLOPs while maintaining high accuracy.

We firstly evaluate our RANP on 3D semantic segmetation on a sparse point-cloud dataset, ShapeNet [Yi et al., 2016], and a dense medical image dataset, BraTS'18 [Menze et al., 2015a; Bakas et al., 2017], with widely used 3D-UNets [Cicek et al., 2016]. We also evaluate RANP on video classification using UCF101 with MobileNetV2 [Sandler et al., 2018] and I3D [Carreira and Zisserman, 2017] and two-view stereo matching using SceneFlow [Mayer et al., 2016] with PSM [Chang and Chen, 2018] which has a high combination of 2D and 3D convolution layers (roughly 3:1), see details in "3D CNNs" in Sec. 5.6.1. Our RANP-f significantly outperforms other neuron pruning methods in *resource efficiency* by yielding large reductions in computational resources (**50%-95% FLOPs reduction and 35%-80% memory reduction**) with comparable accuracy to the unpruned network (see Fig. 5.1).

Furthermore, we also perform extensive experiments to demonstrate **1) scalability** of RANP by pruning with a small input spatial size and training with a large one, **2) transferability** by pruning using ShapeNet and training on BraTS'18 and vice versa, **3) lightweight** training on a single GPU, and **4) fast** training with increased batch size on a single GPU.

5.2 Related Work

Previous works of network pruning mainly focus on pruning parameters in 2D CNNs [Lee et al., 2019; Han et al., 2015; Guo et al., 2016; Dong et al., 2017; Chen et al., 2018a] and neuron pruning [Yu et al., 2017; He et al., 2017; Li et al., 2016; Yu et al., 2018; Huang and Wang, 2018; He et al., 2018]. While a majority of the pruning methods use the traditional prune-retrain scheme with a combined loss function of pruning criteria [Han et al., 2015; He et al., 2017; Yu et al., 2018], some pruning at initialization methods are able to reduce computational complexity in training [Lee et al., 2019; Zhang and Stadie, 2019; Li et al., 2019a; Yu and Huang, 2019; Wang et al., 2020]. While very few of them can be applied to 3D CNNs [Molchanov et al., 2017; Zhang et al., 2019c; Chen et al., 2020], none of them prune networks at initialization, and thus, none of them effectively reduce the training-time computational and memory requirements of 3D CNNs.

5.2.1 2D CNN Pruning

Parameter pruning merely sparsifies filters to obtain small models without losing the high learning capacity. Han et al. [2015] adopted an iterative method of removing

parameters with values below a threshold. Lee et al. [2019] recently proposed a single-shot method with connection sensitivity by using the gradient magnitude of parameter masks to retain top-κ parameters. These filter-sparse methods, however, do not directly yield large speedup and memory reductions.

By contrast, *neuron pruning*, also known as filter pruning or channel pruning, can effectively reduce computational resources. For instance, Li et al. [2016] used L_1 normalization to remove unimportant filters by using the connecting features. He et al. [2017] adopted a LASSO regression to prune network layers and a reconstruction in least square manner. Yu et al. [2017] proposed a group-wise 2D-filter pruning in each 3D-filter by using a learning-based method and a knowledge distillation. Structure learning based MorphNet [Gordon et al., 2018] and SSL [Wen et al., 2016] aim at pruning activations with structure constraints or regularization. These approaches only reduce the test-time resource requirement while we focus on reducing the resource requirement of large 3D CNNs at training time.

5.2.2 3D CNN Pruning

To improve the efficiency of training 3D CNNs, some works like SSC [Graham et al., 2018] and OctNet [Riegler et al., 2017] use efficient data structures to reduce the memory requirement for sparse point-cloud data. However, these approaches are not applicable to dense data, *e.g.*, MRI images and videos, making the resource requirement prohibitively large and tackling efficient network training.

Hence, it is desirable to develop an efficient pruning for 3D CNNs that can handle dense 3D data which is common in real applications. Only very few works are relevant to 3D CNN pruning. Molchanov et al. [2017] proposed a greedy criteria-based method to reduce resources via backpropagation with a small 3D CNN for hand gesture recognition. Zhang et al. [2019c] used a regularization-based pruning method by assigning regularization to weight groups with 4× speedup in theory. Recently, Chen et al. [2020] converted the time dimensionality of 3D filters into frequency domain to eliminate redundancy in an iterative optimization for convergence. Being a parameter pruning method, this does not lead to large FLOPs and memory reductions, *e.g.*, merely a 2× speedup compared to our 28× (see Sec. 5.6.3). In summary, these methods embed pruning in the iterative network optimization and require extensive resources, which is inefficient for 3D CNNs.

5.2.3 Pruning at Initialization

While few works adopted pruning at initialization, some achieved impressive success. SNIP [Lee et al., 2019] is the first single-shot pruning method that presented a high possibility of pruning at initialization with minimal accuracy loss in training, followed by many recent works on single-shot pruning [Zhang and Stadie, 2019; Li et al., 2019a; Yu and Huang, 2019; Wang et al., 2020]. But none of them have been applied to 3D CNN pruning.

In addition to being a parameter pruning approach, SNIP was evaluated only on

small-scale datasets [Lee et al., 2019], such as MNIST and CIFAR-10. Therefore, it is unclear that whether it can be applied to 3D CNNs for large-scale datasets. Our experiments indicate that, while SNIP itself is not capable of yielding large resource reduction in 3D CNNs, our RANP can greatly reduce the computational resources without causing network infeasibility, that is ensuring at least one neuron in each layer. Furthermore, we show that RANP enjoys strong transferability among datasets and enables fast and lightweight training of large 3D volumetric data segmentation on a single GPU.

5.3 Preliminaries

We first briefly describe the main idea of SNIP [Lee et al., 2019] which removes redundant parameters prior to training. Given a dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{S}$ with input $\mathbf{x}_i$ and ground truth $\mathbf{y}_i$ and the sparsity level κ, the optimization problem solved by SNIP can be written as

$$\begin{aligned} \min_{\mathbf{c},\mathbf{w}} L(\mathbf{c} \odot \mathbf{w}; \mathcal{D}) = \min_{\mathbf{c},\mathbf{w}} \frac{1}{S} \sum_{i=1}^{S} \ell\left(\mathbf{c} \odot \mathbf{w}, (\mathbf{x}_i, \mathbf{y}_i)\right) \ , \\ \text{s.t.} \quad \mathbf{w} \in \mathbb{R}^m, \ \ \mathbf{c} \in \{0,1\}^m, \ \ \|\mathbf{c}\|_0 \leq \kappa \ , \end{aligned} \tag{5.1}$$

where $\mathbf{w}$ is a m-dimensional vector of parameters, $\mathbf{c}$ is the corresponding binary mask on the parameters, $\ell(\cdot)$ is the standard loss function (*e.g.*, cross-entropy loss), and $\|\cdot\|_0$ denotes L_0 norm to count the number of non-zero c_j in $\mathbf{c}$. The mask $c_j \in \{0,1\}$ for parameter w_j denotes that the parameter is retained in the compressed model if $c_j = 1$ and otherwise it is removed. In order to optimize the above problem, they first relax the binary mask constraint $\mathbf{c}$ from $0,1^m$ to $[0,1]^m$. Then an importance function for parameter w_j is calculated by the normalized magnitude of the loss gradient over mask c_j as

$$s_j = \frac{\left|g_j\left(\mathbf{w}; \mathcal{D}\right)\right|}{\sum_{k=1}^{m} \left|g_k\left(\mathbf{w}; \mathcal{D}\right)\right|} \ , \quad \text{where } g_j\left(\mathbf{w}; \mathcal{D}\right) = \left.\frac{\partial L(\mathbf{c} \odot \mathbf{w}; \mathcal{D})}{\partial c_j}\right|_{\mathbf{c}=\mathbf{1}} . \tag{5.2}$$

Then only top-κ parameters are retained based on the descending-sorted parameter importance, called connection sensitivity in Lee et al. [2019], $\mathbf{s}$ defined above. Upon pruning, the retained parameters are trained in the standard way. It is interesting to note that, even though having the mask $\mathbf{c}$ is easier to explain the intuition, SNIP can be implemented without these additional variables by noting that $g_j\left(\mathbf{w}; \mathcal{D}\right) = \left(\partial L\left(\mathbf{w}; \mathcal{D}\right) / \partial w_j\right) w_j$ [Lee et al., 2020]. This method has shown remarkable results in achieving $> 95\%$ sparsity on 2D image classification tasks with minimal loss of accuracy. Such a parameter pruning method is important, however, it cannot lead to sufficient computation and memory reductions to train a deep 3D CNN on current off-the-shelf graphics hardware. In particular, the sparse weight matrices cannot efficiently reduce memory or FLOPs, and they require specialized sparse matrix implementations for speedup. In contrast, neuron pruning directly

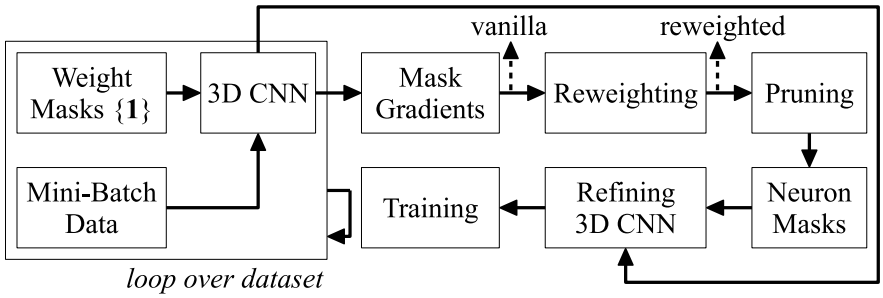


Figure 5.2: *Flowchart of RANP algorithm. The refining generates a new yet slim network for the resource-efficient training.*

translates into practical gains of reducing both memory and FLOPs. This is crucial in 3D CNNs due to their substantially higher resource requirement compared to 2D CNNs.

5.4 Resource Aware Neuron Pruning at Initialization

To explain the proposed RANP, we first extend the SNIP idea to neuron pruning at initialization. Then we discuss a resource aware reweighting strategy to further reduce the computational requirements of the pruned network. The flowchart of our RANP algorithm is shown in Fig. 5.2.

Before introducing our neuron importance, we first consider a fully-connected feed-forward neural network for the simplicity of notations. Consider weight matrices $\mathbf{W}^l \in \mathbb{R}^{N_l \times N_{l-1}}$, biases $\mathbf{b}^l \in \mathbb{R}^{N_l}$, pre-activations $\mathbf{h}^l \in \mathbb{R}^{N_l}$, and post-activations $\mathbf{x}^l \in \mathbb{R}^{N_l}$, for layer $l \in \mathcal{K} = \{1, \ldots, K\}$. Now the feed-forward dynamics is

$$\mathbf{x}^l = \phi\left(\mathbf{h}^l\right) \ , \quad \text{where } \mathbf{h}^l = \mathbf{W}^l \mathbf{x}^{l-1} + \mathbf{b}^l \tag{5.3}$$

and the activation function $\phi : \mathbb{R} \to \mathbb{R}$ is applied to every element of its input and the network input is denoted by $\mathbf{x}^0$. We then introduce binary masks on neurons (*i.e.*, post-activations). The feed-forward dynamics is then modified to include this masking operation as

$$\mathbf{x}^l = \mathbf{c}^l \odot \phi\left(\mathbf{h}^l\right) \ , \quad \text{where } \mathbf{c}^l \in \{0,1\}^{N_l} \ , \ \forall l \in \mathcal{K} \tag{5.4}$$

and neuron mask $c_u^l = 1$ indicates neuron x_u^l is retained and otherwise pruned. To this end, neuron pruning can be written as the following optimization problem

$$\begin{aligned} &\min_{\mathbf{c},\mathbf{w}} L(\mathbf{c}, \mathbf{w}; \mathcal{D}) = \min_{\mathbf{c},\mathbf{w}} \frac{1}{S} \sum_{i=1}^{S} \ell\left(\mathbf{c}, \mathbf{w}; (\mathbf{x}_i, \mathbf{y}_i)\right) \ , \\ &\text{s.t.} \quad \mathbf{w} \in \mathbb{R}^m, \ \ \mathbf{c} \in \{0,1\}^n, \ \ \|\mathbf{c}\|_0 \leq \kappa \ , \end{aligned} \tag{5.5}$$

where n is the total number of neurons and $\ell(\mathbf{c}, \cdot; \cdot)$ denotes a standard loss function

of the feed-forward mapping with neuron masks $\mathbf{c}$ defined in Eq. (5.4). This can be easily extended to convolutional and skip-concatenation operations.

As removing neurons could largely reduce memory and FLOPs compared to merely sparsifying parameters, the core of our approach is benefited by removing redundant neurons from the model. We use an influence function concept developed for parameters to establish neuron importance through the loss function, to locate redundant neurons.

5.4.1 Neuron Importance

Note that, neuron importance can be derived from the SNIP-based parameter importance discussed in Sec. 5.3. Another approach is to directly define neuron importance as the normalized magnitude of the neuron mask gradients analogous to parameter importance.

5.4.1.1 Neuron Importance with Parameter Mask Gradients

The first approach to calculate neuron importance depends on the magnitude of parameter mask gradients, denoted as Magnitude of Parameter Mask Gradients (MPMG). Thus, the importance of neuron x_u^l is

$$s_u^l = f\left(|g_{u1}^l|, \ldots, |g_{uN_{l-1}}^l|\right) , \tag{5.6}$$

where $g_{uv}^l = \partial L\left(\mathbf{c} \odot \mathbf{w}; \mathcal{D}\right) / \partial c_{uv}^l$ with c_{uv}^l as the mask of parameter w_{uv}^l. Refer to Eq. (5.2). Here, $f(\cdot) : \mathbb{R}^{N_{l-1}} \to \mathbb{R}$ is a function mapping a set of values to a scalar. We choose $f(\cdot) = \text{sum}(\cdot)$ by comparing with mean and max functions, in Table 5.1. We then set neuron masks as 1 for neurons with top-κ largest neuron importance.

5.4.1.2 Neuron Importance with Neuron Mask Gradients

Another approach is to directly compute mask gradients on neurons and treat their magnitudes as neuron importance, denoted as Magnitude of Neuron Mask Gradients (MNMG). The neuron importance of x_u^l is calculated by

$$s_u^l = \left| \frac{\partial L\left(\mathbf{c}, \mathbf{w}; \mathcal{D}\right)}{\partial c_u^l} \bigg|_{\mathbf{c}=\mathbf{1}} \right| . \tag{5.7}$$

Noting that a non-linear activation function $\phi(\cdot)$ in CNN including but not limited to ReLU can satisfy $\phi(ch) = c\phi(h)$, for all $c \geq 0$. Given such a homogeneous function, the calculation of neuron importance with neuron masks can be derived from parameter mask gradients in the form of

$$\frac{\partial L\left(\mathbf{c}, \mathbf{w}; \mathcal{D}\right)}{\partial c_u^l} \bigg|_{\mathbf{c}=\mathbf{1}} = \sum_{v=1}^{N_{l-1}} \frac{\partial L\left(\mathbf{c} \odot \mathbf{w}; \mathcal{D}\right)}{\partial c_{uv}^l} \bigg|_{\mathbf{c}=\mathbf{1}} . \tag{5.8}$$

Details of the influence of such an activation function on neuron importance are provided in Sec. 5.4.5. These two approaches for neuron importance are in a similar form that while MPMG uses the sum of magnitudes, MNMG uses the magnitude of the sum of parameter mask gradients. It can be implemented directly from parameter gradients.

The neuron importance based on MPMG or MNMG approach can be used to remove redundant neurons. However, they could lead to an imbalance of sparsity levels of each layer in 3D network architectures. As shown in Table 5.2, the computational resources required by vanilla neuron pruning are much higher than those by other sparsity enforcing methods, *e.g.*, random neuron pruning and layer-wise neuron pruning. We hypothesize that this is caused by the layer-wise imbalance of neuron importance which unilaterally emphasizes on some specific layer(s) and may lead to network infeasibility by pruning the whole layer(s). This behavior is also observed in Lee et al. [2020], and orthogonal initialization is thus recommended to solve the problem for 2D CNN pruning, which however cannot result in balanced neuron importance in our case, see results in Sec. 5.6.8.

In order to resolve this issue, we propose resource aware neuron pruning (RANP) with reweighted neuron importance, and the details are provided below.

5.4.2 Selection of Vanilla Neuron Pruning

Here, we selected sum operation in MPMG and MNMG for vanilla neuron pruning considering the trade-off between computational resources and accuracy. To give a comprehensive study, we demonstrate detailed results of mean, max, and sum operations of MPMG and MNMG in Table 5.1. We relaxed the sum operation in (5.8) to mean and max.

In Table 5.1, we aim at obtaining the maximum neuron sparsity in order to reduce the computational resources at an extreme sparsity level with minimal accuracy loss.

Vividly, for *ShapeNet*, MPMG-sum achieves the largest maximum neuron sparsity 78.24% among all with merely $\sim$0.53% accuracy loss. Differently, for *BraTS'18*, MNMG-sum has the largest maximum neuron sparsity 81.32%; however, the accuracy loss can reach up to $\sim$8.48%. In contrast, while MPMG-sum has the second-largest maximum neuron sparsity 78.17%, the accuracy loss is much smaller than MNMG-sum. For *UCF101*, it is surprising that many manners have low accuracy. As we analyse the reason in the footnote in Table 5.1, with the extreme neuron sparsity, some layers of the pruned networks have only 1 neuron retained, losing sufficient features for learning, and thus, leading to low accuracy. For *SceneFlow*, MPMG-sum

[2]For MobileNetV2 pruned by MPMG-mean, MPMG-max, MNMG-max, and MNMG-sum, the accuracy is very low because 1) the neuron sparsity here is the extreme (largest) value, a larger one will make the network infeasible by removing whole layer(s) and 2) the distribution of neuron importance is rather imbalanced possibly caused by the high mixture of 1^3 kernels and 3^3 in MobileNetV2.

In the pruned networks, we observe that, for MPMG-mean, MPMG-max, and MNMG-max, the last convolutional layer has only 1 neuron retained; for MNMG-sum, 2 convolutional layers have only 1 neuron retained. Note that, this imbalance issue can be greatly alleviated by the reweighting of our RANP, while we select MPMG-sum as vanilla NP merely according to the results in Table 5.1.

Table 5.1: *Vanilla neuron pruning method. Main resource consumption (GFLOPs and memory) are considered but not parameters whose resource consumption is much smaller than memory. Among the neuron pruning methods, we mark bold* ***the best*** *and underline the second best. "↑" and "↓" in "Metrics" denote the larger and the smaller the better respectively. "EPE": End-Point Error. Overall, we selected MPMG-sum as vanilla NP for ShapeNet, BraTS'18, and UCF101 and MNMG-sum as vanilla NP for SceneFlow; the corresponding neuron sparsity for large resource reductions with small accuracy loss.*

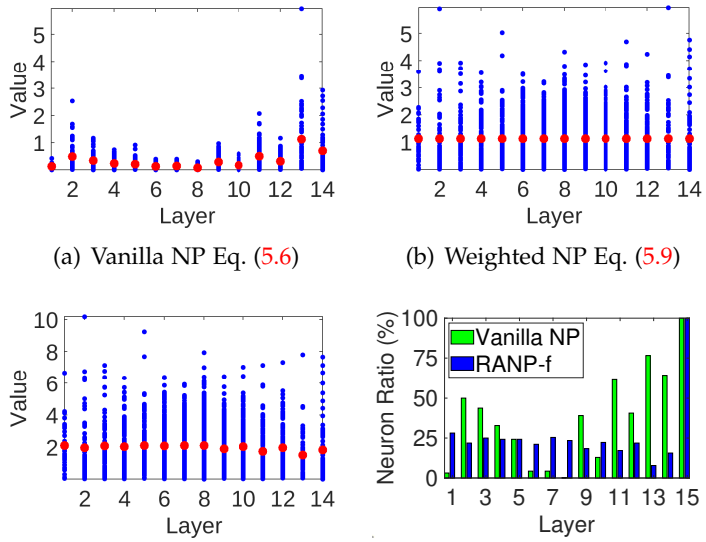
Dataset	Model	Manner	Sparsity(%)	Param(MB)	GFLOPs	Memory(MB)	Metrics(%)		
							mIoU		
ShapeNet	3D-UNet	Full	0	62.26	237.85	997.00	83.79±0.21		
		MPMG-mean	68.10	5.08	110.14	819.97	83.33±0.18		
		MPMG-max	70.24	4.54	107.38	809.88	**83.79±0.10**		
		MPMG-sum	78.24	2.54	**55.69**	**557.32**	83.26±0.14		
		MNMG-mean	63.03	4.23	112.95	834.98	83.46±0.13		
		MNMG-max	73.93	3.67	103.57	796.44	83.51±0.08		
		MNMG-sum	66.93	4.29	100.34	783.14	83.65±0.02		
							ET	TC	WT
BraTS'18	3D-UNet	Full	0	15.57	478.13	3628.00	72.96±0.60	73.51±1.54	86.79±0.35
		MPMG-mean	65.64	1.48	226.86	3038.27	73.51±0.82	73.28±1.14	87.15±0.43
		MPMG-max	75.78	0.83	189.43	2812.53	**73.67±0.98**	72.73±1.70	86.44±0.71
		MPMG-sum	78.17	0.55	104.50	1936.44	71.94±1.68	69.39±2.29	84.68±0.78
		MNMG-mean	63.85	1.08	176.76	2790.64	73.35±0.70	**73.38±0.94**	**87.21±0.38**
		MNMG-max	80.05	0.59	169.99	2676.05	72.52±1.91	72.40±1.74	84.63±0.60
		MNMG-sum	81.32	0.35	**73.50**	**1933.20**	64.48±1.10	68.47±1.59	80.71±1.07
							Top-1	Top-5	
UCF101	MobileNetV2	Full	0	9.47	0.58	157.47	47.08±0.72	76.68±0.50	
		MPMG-mean	26.31	4.39	0.55	156.00	2.98±0.14[2]	14.04±0.14	
		MPMG-max	29.48	3.96	0.54	155.38	3.49±0.12	13.64±0.10	
		MPMG-sum	33.15	6.35	0.55	155.17	**46.32±0.79**	**75.42±0.60**	
		MNMG-mean	38.91	2.79	0.50	147.69	29.13±0.92	62.93±1.37	
		MNMG-max	50.33	2.59	0.53	153.45	2.84±0.06	13.40±0.23	
		MNMG-sum	39.89	4.66	**0.43**	**120.01**	1.03±0.00	5.76±0.00	
	I3D	Full	0	47.27	27.88	201.28	51.58±1.86	77.35±0.63	
		MPMG-mean	16.47	31.57	26.50	196.51	51.88±2.00	77.98±1.46	
		MPMG-max	19.83	30.06	26.31	195.62	**52.44±1.25**	**78.08±1.27**	
		MPMG-sum	25.32	29.93	25.76	192.42	51.57±1.46	78.07±1.34	
		MNMG-mean	35.36	16.69	**15.37**	**124.85**	49.26±0.96	75.70±1.49	
		MNMG-max	40.27	17.86	23.73	184.77	44.90±1.19	74.43±1.26	
		MNMG-sum	32.87	20.00	16.03	125.17	46.90±1.26	74.02±1.25	
							Acc-1↑	Acc-3↑	EPE↓
SceneFlow	PSM	Full	0	19.93	771.02	7217.37	85.97±0.70	94.21±0.33	1.42±0.07
		MPMG-mean	21.50	13.50	575.13	6689.85	85.32±0.77	93.71±0.43	1.53±0.11
		MPMG-max	27.25	11.61	540.05	6467.20	84.84±0.80	93.65±0.42	1.56±0.11
		MPMG-sum	20.92	14.03	571.21	6663.24	**85.75±0.48**	**94.17±0.25**	**1.43±0.06**
		MNMG-mean	44.33	6.00	**449.20**	5744.31	85.04±0.50	93.07±0.23	1.54±0.06
		MNMG-max	50.07	6.85	473.74	5872.91	85.11±0.71	93.69±0.36	1.53±0.09
		MNMG-sum	46.20	6.52	452.25	**5653.44**	85.24±0.77	93.83±0.41	1.51±0.09

achieves the highest accuracy and EPE among all pruning methods. The resource reductions, *i.e.*, 25.92% FLOPs and 7.68% memory, however, are much smaller than MNMG-sum, *i.e.*, 41.34% FLOPs and 21.67% memory, due to the small maximum neuron sparsity 20.92%. This is possibly caused by the massive output addition and concatenations across multiple layers and a high combination of 2D and 3D convolution layers.

Hence, considering the comprehensive performance of reducing resources and maintaining the accuracy, MPMG-sum is selected as vanilla NP for general 3D CNNs (without massive cross-additions, cross-concatenations, and a high combination of 2D CNNs) and MNMG-sum is advised to be compared with MPMG-sum for the

trade-off between resource reductions and accuracy. Note that any neuron sparsity greater than the maximum neuron sparsity will make the pruned network infeasible by pruning the whole layer(s).

5.4.3 Resource Aware Reweighting



(a) Vanilla NP Eq. (5.6) (b) Weighted NP Eq. (5.9)

(c) RANP-f Eq. (5.10) (d) Retain ratio

Figure 5.3: *ShapeNet: neuron importance of 3D-UNet becomes balanced and resource-aware from (a) to (c) at neuron sparsity 78.24%. Blue: neuron importance; red: mean values. More illustrations are in Fig. 5.13.*

To tackle the imbalanced neuron importance issue above, we first weight the neuron importance across layers. Weighting neuron importance of x_u^l is defined as

$$\tilde{s}_u^l = \frac{\max_{k=1}^{K} \bar{s}^k}{\bar{s}^l} s_u^l \ , \ \text{where } \bar{s}^k = \frac{1}{N_k} \sum_{u=1}^{N_k} s_u^k, \ \forall\, k \in \mathcal{K} \ . \tag{5.9}$$

Here, $\bar{s}^l$ is the mean neuron importance of layer l and $\tilde{s}_u^l$ is the updated neuron importance. This helps to achieve the same mean neuron importance in each layer, which largely avoids underestimating neuron importance of specific layer(s) to prevent from pruning the whole layer(s).

To further reduce the memory and FLOPs with minimal accuracy loss, we then reweight the neuron importance $\tilde{s}_u^l$ by available resource, *i.e.*, memory or FLOPs. This reweighting counts on the addition of weighted neuron importance and the effect of the computational resource, denoted as RANP-[m|f], where "m" is for memory and "f" is for FLOPs. We represent the importance of this available resource in layer l as τ_l, refer to Sec. 5.4.4 for details.

The reweighted neuron importance of neuron x_u^l by following weighted addition variant RANP-[m|f] is

$$\hat{s}_u^l = (1 + \lambda \operatorname{softmax}(-\tau_l))\, \tilde{s}_u^l = \left(1 + \lambda \frac{e^{-\tau_l}}{\sum_{k=1}^{K} e^{-\tau_k}}\right) \tilde{s}_u^l \,, \tag{5.10}$$

where coefficient $\lambda > 0$ helps to control the effect of resource on neuron importance. This effect represented by softmax constrains the values into a controllable range [0,1], making it easy to determine λ and function a high resource influence with a small resource occupation.

We then demonstrate the effect of this reweighting strategy over vanilla pruning in Fig. 5.3. In more detail, vanilla neuron importance tends to have high values in the last few layers, making it highly possible to remove all neurons of such as the 7th and 8th layers. Weighting the importance in Fig. 5.3(b) makes the distribution of importance balanced with the same mean value in each layer. Furthermore, since some neurons have different numbers of input channels, each layer requires different FLOPs and memory. Considering the effect of computational resources on training, we embed them into neuron importance as weights.

In Fig. 5.3(c), the last few layers require larger computational resources than the others, and thus their neurons share lower weights, see the tendency of mean values. Vividly, neuron ratio in Fig. 5.3(d) indicates a more balanced distribution by RANP-f than vanilla NP. For instance, very few neurons are retained in the 8th layer by vanilla NP, resulting in low accuracy and low maximum neuron sparsity. With reweighting by RANP-f, however, more neurons can be retained in this layer. Moreover, in Table 5.2, while weighted NP achieves high accuracy, its computational resource reductions are small. In contrast, RANP-f largely decreases the computational resources with a small accuracy loss.

Then with reweighted neuron importance by Eq. (5.10) and $\ddot{s}_\kappa$ as the κth reweighted neuron importance in a descending order, the binary mask of neuron x_u^l can be obtained by

$$c_u^l = 1[\hat{s}_u^l - \ddot{s}_\kappa \geq 0] \,. \tag{5.11}$$

As mentioned in Sec. 5.2, our RANP is more effective in reducing memory and FLOPs than SNIP-based pruning which merely sparsifies parameters but needs high memory required by dense operations in training. RANP can easily remove neurons and all involved input channels at once, leading to huge reductions of input and output channels of the filter. Pseudocode is provided in Algorithm 5.

5.4.4 Resource Aware Reweighting Constraints

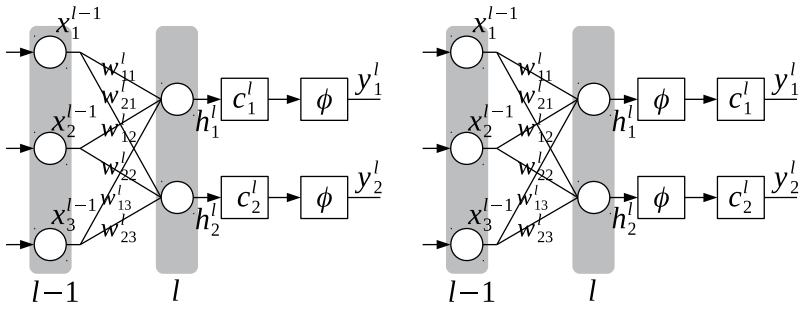
As described above, the reweighting of RANP is conducted by first balancing the layer-wise distribution of neuron importance and then adopting resource importance τ_l for layer $l \in \mathcal{K}$ to further reduce resources. Since FLOPs and memory are the main resources of 3D CNNs, τ_l is defined by FLOPs or memory as follows.

Generally, given input dimension of the lth layer $(x_{\text{in}}, x_h, x_w, x_d)$[3], neuron dimension $(f_{\text{out}}, f_{\text{in}}, f_h, f_w, f_d)$, and output dimension $(y_{\text{in}}, y_h, y_w, y_d)$ with $x_{\text{in}} = f_{\text{in}}$ and $f_{\text{out}} = y_{\text{in}}$, the resource importance in terms of FLOPs and memory is defined by

$$\begin{aligned} \text{FLOPs: } \tau_l &= [(f_h f_w f_d + f_h f_w f_d - 1) f_{\text{in}} + f_{\text{in}} - 1 + 1|_{\text{bias}}] \, y_{\text{in}} y_h y_w y_d \\ &= (2 f_h f_w f_d f_{\text{in}} - 1 + 1|_{\text{bias}}) \, y_{\text{in}} y_h y_w y_d \, , \qquad (5.12a) \\ \text{Memory: } \tau_l &= y_{\text{in}} y_h y_w y_d \, , \qquad (5.12b) \end{aligned}$$

where $(f_h f_w f_d)$ is the operation number of multiplications of filter[4] and layer input, $(f_h f_w f_d - 1)$ is for additions of values from the multiplications, (f_{in}) is for multiplications over all f_{in} filters, $(f_{\text{in}} - 1)$ is for additions of values from all these multiplications, $(1|_{\text{bias}})$ is for an addition when the neuron has a bias, and $(y_{\text{in}} y_h y_w y_d)$ is for all elements of the layer output.

5.4.5 Impacts of Activation Function



(a) Pre-activations (b) Post-activations

Figure 5.4: *Pruning pre-activations and post-activations, where* **x** *are layer inputs,* **w** *are weights,* **c** *are neuron masks,* $\phi(\cdot)$ *is an activation function,* **h** *are hidden values, and* **y** *are outputs.*

In the following, we first establish the relation between MPMG and MNMG for calculating neuron importance given a homogeneous activation function $\phi(\cdot)$ that includes but not limited to ReLU used in 3D CNNs. Then, we analyze the impact of such an activation function on the calculation of neuron importance is achieved by derivating the mask gradients on post-activations and pre-activations illustrated in Figs. 5.4(b) and 5.4(a) respectively.

Proposition 1 *For a network activation function* $\phi(w)$*:* $\mathbb{R} \to \mathbb{R}$ *being a homogeneous function of degree 1 satisfying* $\phi(cw) = c\phi(w), \forall c \geq 0$*, the neuron mask gradient equals the sum of parameter mask gradients of this neuron.*

[3]The dimension order follows that in PyTorch.
[4]Here, we refer a 3D filter with dimension (f_h, f_w, f_d).

Proof: Given a neuron mask c_1 before the activation function $\phi(\cdot)$ in Fig. 5.4(a) and the output of the 1st neuron as y_1^l, we have

$$\begin{aligned} y_1^l &= \phi(c_1^l \odot h_1^l) \\ &= \phi\left(c_1^l \odot \left(x_1^{l-1} w_{11}^l + x_2^{l-1} w_{12}^l + x_3^{l-1} w_{13}^l\right)\right) \\ &= \phi\left(c_1^l x_1^{l-1} w_{11}^l + c_1^l x_2^{l-1} w_{12}^l + c_1^l x_3^{l-1} w_{13}^l\right) . \end{aligned} \tag{5.13}$$

The gradient of loss L over the neuron mask c_1^l is

$$\frac{\partial L}{\partial c_1^l} = \frac{\partial L}{\partial y_1^l}\frac{\partial y_1^l}{\partial c_1^l} = \frac{\partial L}{\partial y_1^l}\left(x_1^{l-1} w_{11}^l + x_2^{l-1} w_{12}^l + x_3^{l-1} w_{13}^l\right) . \tag{5.14}$$

Meanwhile, if setting masks on weights of this neuron directly, we can obtain

$$y_1^l = \phi(c_{11}^l x_1^{l-1} w_{11}^l + c_{12}^l x_2^{l-1} w_{12}^l + c_{13}^l x_3^{l-1} w_{13}^l) , \tag{5.15}$$

then the gradient of weight mask, *e.g.*, c_{11}^l, from loss is

$$\frac{\partial L}{c_{11}^l} = \frac{\partial L}{\partial y_1^l}\frac{\partial y_1^l}{\partial c_{11}^l} = \frac{\partial L}{\partial y_1^l} x_1^{l-1} w_{11}^l . \tag{5.16}$$

Similarly,

$$\frac{\partial L}{\partial c_{11}^l} + \frac{\partial L}{\partial c_{12}^l} + \frac{\partial L}{\partial c_{13}^l} = \frac{\partial L}{\partial y_1^l}\left(x_1^{l-1} w_{11}^l + x_2^{l-1} w_{12}^l + x_3^{l-1} w_{13}^l\right) . \tag{5.17}$$

Clearly, Eq. (5.14) equals Eq. (5.17). Hence, the neuron mask gradients can be calculated by parameter mask gradients. To this end, the proof is done.

Furthermore, given such a homogeneous activation function in Prop. 1, the importance of a post-activation equals the importance of its pre-activation. In more detail, for post-activations in Fig. 5.4(b), output y_1^l is

$$y_1^l = c_1^l \odot \phi(h_1^l) = c_1^l \odot \phi\left(x_1^{l-1} w_{11}^l + x_2^{l-1} w_{12}^l + x_3^{l-1} w_{13}^l\right) . \tag{5.18}$$

Since the activation function satisfies $c\phi(w) = \phi(cw)$,

$$y_1^l = \phi(c_1^l x_1^{l-1} w_{11}^l + c_1^l x_2^{l-1} w_{12}^l + c_1^l x_3^{l-1} w_{13}^l) . \tag{5.19}$$

The neuron importance determined by neuron mask c_1^l is

$$\frac{\partial L}{\partial c_1^l} = \frac{\partial L}{\partial y_1^l}\frac{\partial y_1^l}{\partial c_1^l} = \frac{\partial L}{\partial y_1^l}\left(x_1^{l-1} w_{11}^l + x_2^{l-1} w_{12}^l + x_3^{l-1} w_{13}^l\right) . \tag{5.20}$$

Clearly, Eq. (5.20) equals Eq. (5.14). Now, the importance of pre-activations and post-activations is the same given such a homogeneous activation function.

5.5 Pseudocode of RANP Procedures

In Algorithm 5, we provide the pseudocode of the pruning procedures of RANP. In Algorithm 6, we used a simple half-space method to automatically search for the max neuron sparsity with network feasibility. Note that this searching cannot guarantee a small accuracy loss but merely to decide the maximum pruning capability. The relation between pruning capability and accuracy was provided in the experimental section and Table 5.1.

Algorithm 5: Pruning Procedures of RANP-[f|m].

Input: Dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{S}$ with B samples per batch, neuron sparsity κ, resource importance $\{\tau_l\}$, coefficient $\lambda > 0$, and parameter masks $\mathbf{c} = \{c_{uv}^l\}$, where layer $l \in \mathcal{K} = \{1, ..., K\}$, and neuron $u \in \mathcal{N}_l = \{1, ..., N_l\}$.

Output: Binary neuron masks $\hat{\mathbf{c}} = \{\hat{c}_u^l\}$.

for batch $t \in \{1, ..., \lfloor S/B \rfloor\}$ **do**
- $\mathcal{D}^t \leftarrow \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=(t-1)B+1}^{tB}$ ▷mini-batch
- $g_{uv}^l \leftarrow \partial L(\mathbf{c} \odot \mathbf{w}; \mathcal{D}^t)/\partial c_{uv}^l$ ▷parameter mask gradient, Eq. (5.2)
- $g_{uv}^l \leftarrow |g_{uv}^l|$, for MPMG ▷parameter mask importance, Eq. (5.6)
- $\nabla c_{uv}^l \xleftarrow{+} g_{uv}^l, \forall u \in \mathcal{N}_l, \forall v \in \mathcal{N}_{l-1}$ ▷gradient accumulation

$\nabla c_{uv}^l \leftarrow \nabla c_{uv}^l / \lfloor S/B \rfloor, \forall u \in \mathcal{N}_l, \forall v \in \mathcal{N}_{l-1}, \forall l \in \mathcal{K}$ ▷average on mini-batch

$s_u^l \leftarrow |\sum_{v=1}^{N_{l-1}} \nabla c_{uv}^l|, \forall u \in \mathcal{N}_l, \forall l \in \mathcal{K}$ ▷vanilla neuron importance, Eq. (5.7)

$\bar{s}^l \leftarrow \sum_{u \in \mathcal{N}_l} s_u^l / N_l, \forall l \in \mathcal{K}$ ▷mean neuron importance, Eq. (5.9)

$\tilde{s}_u^l \leftarrow (\max_{j \in \mathcal{K}} \bar{s}^j / \bar{s}^l) s_u^l, \forall u \in \mathcal{N}_l, \forall l \in \mathcal{K}$ ▷weighting, Eq. (5.9)

$\hat{s}_u^l \leftarrow (1 + \lambda e^{-\tau_l} / \sum_{j \in \mathcal{K}} e^{-\tau_j}) \tilde{s}_u^l, \forall u \in \mathcal{N}_l, \forall l \in \mathcal{K}$ ▷reweighting, Eq. (5.10)

$\{\ddot{s}_u\} \leftarrow \text{SortDescending}\left(\{\hat{s}_u^l\}\right), \forall u \in \mathcal{N}_l, \forall l \in \mathcal{K}$ ▷sorting in descending

$\hat{c}_u^l \leftarrow 1[\hat{s}_u^l - \ddot{s}_\kappa \geq 0], \forall u \in \mathcal{N}_l, \forall l \in \mathcal{K}$ ▷binary neuron mask, Eq. (5.11)

5.6 Experiments

We evaluated RANP on 3D-UNets for 3D semantic segmentation, MobileNetV2 and I3D for video classification, and PSM for two-view stereo matching. Experiments are in PyTorch on Nvidia Tesla P100-SXM2-16GB GPUs.

5.6.1 Experimental Setup

5.6.1.1 3D Datasets

For 3D semantic segmentation, we adopted the large-scale 3D sparse point-cloud dataset, ShapeNet [Yi et al., 2016], and dense biomedical MRI sequences, BraTS'18 [Menze et al., 2015a; Bakas et al., 2017]. *ShapeNet* consists of 50 object part classes, 14,007 training samples, and 2,874 testing samples. We split it into 6,955 training

Algorithm 6: Auto-Search for Max Neuron Sparsity .

Input: Dataset $\mathcal{D}$, layerwise resource usage $\boldsymbol{\tau}$ w.r.t. FLOPs or memory, coefficient $\lambda > 0$, lower and upper sparsity κ_{min} and κ_{max}, threshold $\delta = 1e^{-4}$. When pruning returns a successful status, a feasible network is obtained where each layer has at least one neuron.

Output: Max neuron sparsity κ^*.

Initialize $\kappa_{min} \leftarrow 0, \kappa_{max} \leftarrow 1$
while $(\kappa_{max} - \kappa_{min} > \delta)$ **do**
  $\kappa = 0.5\,(\kappa_{min} + \kappa_{max})$
  $statusy = \text{NeuronPruning}(\mathcal{D}, \boldsymbol{\tau}, \lambda, \kappa)$ ▷Algorithm 5
  if $yissuccessful$ **then**
    $\kappa_{min} \leftarrow \kappa$
  else
    $\kappa_{max} \leftarrow \kappa$
$\kappa^* = \kappa$

samples and 7,052 validation samples as Graham et al. [2018] to assign each point or voxel with a part class.

BraTS'18 includes 210 High Grade Glioma (HGG) and 75 Low Grade Glioma (LGG) cases. Each case has 4 MRI sequences, *i.e.*, T1, T1_CE, T2, and FLAIR. The task is to detect and segment brain scan images into 3 categories: Enhancing Tumor (ET), Tumor Core (TC), and Whole Tumor (WT). The spatial size is 240×240×155 in each dimension. We adopted the splitting strategy of cross-validation in Kao et al. [2018] with 228 cases for training and 57 cases for validation.

For video classification, we used video dataset, *UCF101* [Soomro et al., 2012] with 101 action categories and 13,320 videos. 2D spatial dimension from images and temporal dimension from frames are cast as dense 3D inputs. Among the 3 official train/test splits, we used split-1 which has 9,537 videos for training and 3783 videos for validation.

For two-view stereo matching, we used *SceneFlow* [Mayer et al., 2016] containing 3 synthetic videos flyingthings3D, driving, and Monkaa for 192-disparity estimation. In total, 35,454 image pairs were used for training and 4,370 image pairs for validation. The image pairs have a high resolution of 540 × 960 pixels in RGB format.

5.6.1.2 3D CNNs

For 3D semantic segmentation on ShapeNet (sparse data) and BraTS'18 (dense data), we used the standard 15-layer 3D-UNet [Cicek et al., 2016] including 4 encoders, each consists of two "3D convolution + 3D batch normalization + ReLU", a "3D max pooling", four decoders, and a confidence module by softmax. It has 14 convolution layers with 3^3 kernels and 1 layer with 1^3 kernel.

For video classification, we used the popular MobileNetV2 [Sandler et al., 2018; Kopuklu et al., 2019] and I3D (with inception as backbone) [Carreira and Zisserman, 2017] on UCF101. MobileNetV2 has a linear layer and 52 convolution layers while 18 of them are 3^3 kernels and the rest are 1^3. I3D has a linear layer and 57 convolution

layers, 19 of which are 3^3 kernels, 1 is 7^3, and the rest are 1^3.

For two-view stereo matching, we used the widespread *PSM* [Chang and Chen, 2018] network, which has multi-scale feature extraction with 2D convolution layers to calculate left-right image feature cost volumes, followed by 3 encoder-decoder modules with 3D convolution layers. PSM highly combines 2D/3D convolution layers with 61 2D layers (including linear layer) and 28 3D layers where 4,352 neurons are in the 2D layers and 1,283 neurons are in the 3D layers.

In PSM, massive cross-layer additions are adopted to preserve high-level features. Three losses from the 3D encoder-decoder modules are weighted for training. Specifically, for the massive additions across multiple layer outputs in PSM, we used the maximum retained channels as the out_channels for all associated layers. For instance, given 3 layers having 256 out_channels (also denoted as neurons) for addition, they retain 100, 50, 120 out_channels respectively after pruning. To maintain the addition, they are required to have the same number of out_channels. We thus selected the maximum number 120 by $\max\{100, 50, 120\}$ as out_channels of these 3 layers and in_channels of the next layer in the refinement for a lightweight network.

5.6.1.3 Hyper-Parameters in Learning

For *ShapeNet*, we set learning rate to 0.1 with an exponential decay rate $\gamma = 0.04$ by 100 epochs; the batch size is 12 on 2 GPUs; the spatial size for pruning and training is 64^3 while the spatial size for training is 128^3 in Sec. 5.6.7; the optimizer is SGD-Nesterov [Sutskever et al., 2013] with weight decay 0.0001 and momentum 0.9.

For *BraTS'18*, the learning rate is 0.001, decayed by 0.1 at 150th epoch with 200 epochs; the optimizer is Adam [Kingma and Ba, 2015] with weight decay 0.0001 and AMSGrad [Reddi et al., 2018]; the batch size is 2 on 2 GPUs; the spatial size for pruning is 96^3 and 128^3 for training.

For *UCF101*, we adopted similar setup from Kopuklu et al. [2019] with learning rate 0.1, decayed by 0.1 at {40, 55, 60, 70}th epoch; optimizer SGD with weight decay 0.001; batch size 8 on one GPU. The spatial size for pruning and training is 112^2 for MobileNetV2 and 224^2 for I3D; 16 frames are used for the temporal size. Note that in Kopuklu et al. [2019] networks for UCF101 had higher performance since they were pretrained on Kinetics600, while we directly trained on UCF101. A feasible train-from-scratch reference could be Soomro et al. [2012].

For *SceneFlow*, we trained PSM with 15 epochs, learning rate 0.001, 192 disparities, Adam optimizer, and 12 batch size on 3 GPUs. The crop size for training is 256×512 (height $\times$ width) from the original size 540×960.

For Eq. (5.10), we empirically set the coefficient λ as 11 for ShapeNet, 15 for BraTS'18, 80 for UCF101, and 100 for SceneFlow. Glorot initialization [Glorot and Bengio, 2010] was used for weight initialization. Note that we used orthogonal initialization [Saxe et al., 2014] to handle imbalanced layer-wise neuron importance distribution [Lee et al., 2020] but obtained a lower maximum neuron sparsity.

5.6.1.4 Loss Function and Metrics

Standard cross-entropy function was used as the loss function for ShapeNet and UCF101. For BraTS'18, the weighted function in Kao et al. [2018] is

$$L = L_{CE} + \alpha L_{DICE} = L_{CE} + \alpha \frac{1}{C} \sum_{i=1}^{C} \frac{2|\mathbf{P}_i \cap \mathbf{G}_i|}{|\mathbf{P}| + |\mathbf{G}|} , \tag{5.21}$$

where $\alpha = 0.25$ is an empiric weight for dice loss, $\mathbf{P}$ is the prediction, $\mathbf{G}$ is the ground truth, and C is the number of classes. For *SceneFlow*, the standard smooth L_1 loss was used in each of the 3 encoder-decoder modules in PSM [Chang and Chen, 2018] with weights 0.5, 0.7, and 1.0 at the training phase.

$$L(d^p, d^g) = \frac{1}{N} \sum_{i=1}^{N} \text{SmoothL}_1(d_i^p - d_i^g) , \tag{5.22}$$

with

$$\text{SmoothL}_1(x) = \begin{cases} 0.5x^2 & \text{if} |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} , \tag{5.23}$$

where d_i^p and d_i^g are predicted disparity and ground truth disparity at pixel i respectively, and N is the number of image pixels without occlusion.

Meanwhile, ShapeNet accuracy was measured by mean IoU over each part of object category [Yi et al., 2017] while IoU by $|\mathbf{P} \cap \mathbf{G}|/|\mathbf{P} \cup \mathbf{G}|$ was adopted for BraTS'18. For UCF101 classification, top-1 and top-5 recall rates were used. For *SceneFlow*, non-occluded ground truth disparity maps were used for evaluation. End-Point-Error (EPE) was calculated by the absolute pixel-wise disparity difference between the ground truth and the prediction without occlusion areas. Accuracy was measured by the absolute pixel-wise disparity difference under a given pixel threshold, 1 pixel for Acc-1 and 3 pixels for Acc-3 in our case.

5.6.2 Maximum Neuron Sparsity by Vanilla Neuron Pruning

We selected MPMG-sum and MNMG-sum for vanilla neuron importance for comparison. All neurons of the last convolutional layer are retained for the given classes.

In Table 5.1, MPMG-sum for ShapeNet achieves the largest neuron sparsity 78.24% by reducing 76.59% FLOPs, 95.92% parameters, and 44.10% memory with 0.53% accuracy loss. Meanwhile, for BraTS'18, MNMG-sum achieves the largest neuron sparsity 81.32% but has up to 8.48% accuracy loss. MPMG-sum, however, has the largest neuron sparsity 78.17% but smaller accuracy loss with decreased 78.14% FLOPs, 96.46% parameters, and 46.63% memory.

Hence, we selected MPMG-sum as vanilla NP considering the trade-off between the maximum neuron sparsity and the accuracy loss. This is applied to all methods related to weighted neuron pruning and RANP in our experiments. Results of mean and max are in Table 5.1.

5.6.3 Evaluation of RANP on Pruning Capability

Random NP retains κ neurons with neuron indices randomly shuffled. Layer-wise NP retains neurons by using the same retain rate κ in each layer. For SNIP-based parameter pruning, the parameter masks are post-processed by removing redundant parameters and then making sparse filters dense, which is denoted as SNIP NP. For a fair comparison with SNIP NP, we used the maximum parameter sparsity 98.98% for ShapeNet , 98.88% for BraTS'18, 86.26% for MobileNetV2, 81.09% for I3D, and 90.66% for PSM.

5.6.3.1 ShapeNet

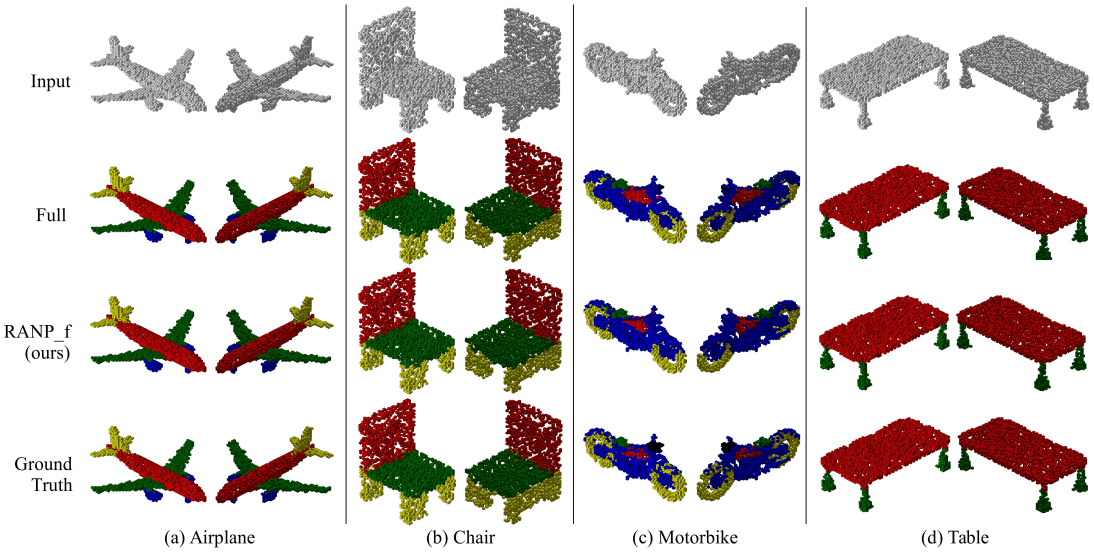


Figure 5.5: *Examples of ShapeNet for 3D semantic segmentation. Spatial size is 64^3 for the volumetric representation of sparse point clouds. We illustrate two views for each model, with (elevation angle, azimuth angle) as (30°, -45°) and (30°, 45°).*

Compared with random NP and layer-wise NP in Table 5.2, the maximum reduced resources by vanilla NP are much less due to the imbalanced layer-wise distribution of neuron importance. Weighted neuron importance by using Eq. (5.9), however, further reduces 18.3% FLOPs and 29.6% memory with 0.14% accuracy loss.

Reweighting by using RANP-f and RANP-m further reduces FLOPs and memory on the basis of weighted NP. Here, RANP-f can reduce 96.8% FLOPs, 95.3% parameters, and 73.7% memory over the unpruned networks. Furthermore, with a similar resource in Table 5.3, RANP achieves $\sim$0.5% increase in accuracy. Note that an unsuitably large λ can additionally reduce the resources but at the cost of accuracy.

Table 5.2: *Evaluation of neuron pruning capability. All models are trained from scratch for 100 epochs on ShapeNet and UCF101, 200 on BraTS'18, and 15 on SceneFlow. Metrics with ± are calculated by the last 5 epochs. "sparsity" is the maximum parameter sparsity for SNIP NP and the maximum neuron sparsity for vanilla NP for others. Among the neuron pruning methods, we mark bold* ***the best*** *and underline the second best. "↓" in resources denotes reduction in %; "↑" and "↓" in "Metrics" denote the larger and the smaller the better respectively. Overall, our RANP-f performs best with large reductions of main resource consumption (GFLOPs and memory) with negligible accuracy loss.*

Dataset	Model		Manner	Sparsity(%)	Param(MB)	GFLOPs	Memory(MB)	Metrics(%)		
								mIoU		
ShapeNet	3D-UNet		Full	0	62.26	237.85	997.00	83.79±0.21		
			SNIP NP	98.98	5.31 (91.5↓)	126.22 (46.9↓)	833.20 (16.4↓)	**83.70±0.20**		
			Random NP	78.24	3.05 (95.1↓)	10.36 (95.6↓)	267.95 (73.1↓)	82.90±0.19		
			Layer-wise NP		2.99 (95.2↓)	11.63 (95.1↓)	296.22 (70.3↓)	83.25±0.14		
			Vanilla NP		2.54 (95.9↓)	55.69 (76.6↓)	557.32 (44.1↓)	83.26±0.14		
		ours	Weighted NP		2.97 (95.2↓)	12.06 (94.9↓)	301.56 (69.8↓)	83.12±0.09		
		ours	RANP-m		3.39 (94.6↓)	**6.68 (97.2↓)**	**214.95 (78.4↓)**	82.35±0.24		
		ours	RANP-f		2.94 (95.3↓)	7.54 (96.8↓)	262.66 (73.7↓)	83.07±0.22		
								ET	TC	WT
BraTS'18	3D-UNet		Full	0	15.57	478.13	3628.00	72.96±0.60	73.51±1.54	86.79±0.35
			SNIP NP	98.88	1.09 (93.0↓)	233.11 (51.2↓)	2999.64 (17.3↓)	**73.33±1.89**	71.98±2.15	**86.44±0.39**
			Random NP	78.17	0.75 (95.2↓)	22.59 (95.3↓)	817.59 (77.5↓)	67.27±0.99	71.62±1.20	74.16±1.33
			Layer-wise NP		0.75 (95.2↓)	24.09 (95.0↓)	836.88 (77.0↓)	69.74±1.33	71.49±1.62	86.38±0.39
			Vanilla NP		0.55 (96.5↓)	104.50 (78.1↓)	1936.44 (46.6↓)	71.94±1.68	69.39±2.29	84.68±0.78
		ours	Weighted NP		0.79 (95.0↓)	22.40 (95.3↓)	860.64 (76.3↓)	71.50±0.63	**75.05±1.19**	84.05±0.65
		ours	RANP-m		0.87 (94.4↓)	**13.47 (97.2↓)**	**506.97 (86.0↓)**	66.70±2.94	62.99±2.38	82.90±0.41
		ours	RANP-f		0.76 (95.1↓)	16.97 (96.5↓)	729.11 (80.0↓)	70.73±0.66	74.50±1.05	85.45±1.06
								Top-1	Top-5	
UCF101	MobileNetV2		Full	0	9.47	0.58	157.47	47.08±0.72	76.68±0.50	
			SNIP NP	86.26	3.67 (61.3↓)	0.54 (6.9↓)	155.35 (1.3↓)	45.78±0.04	75.08±0.17	
			Random NP	33.15	4.58 (51.6↓)	0.34 (41.4↓)	106.68 (32.3↓)	44.74±0.36	74.69±0.58	
			Layer-wise NP		4.56 (51.8↓)	0.33 (43.1↓)	106.92 (32.1↓)	44.90±0.36	75.54±0.34	
			Vanilla NP		6.35 (32.9↓)	0.55 (5.2↓)	155.17 (1.5↓)	**46.32±0.79**	75.42±0.60	
		ours	Weighted NP		4.82 (49.1↓)	0.30 (48.3↓)	100.33 (36.3↓)	46.19±0.51	75.72±0.30	
		ours	RANP-m		4.87 (48.6↓)	0.27 (53.4↓)	**84.51 (46.3↓)**	45.11±0.41	75.53±0.37	
		ours	RANP-f		4.83 (49.0↓)	**0.26 (55.2↓)**	88.01 (44.1↓)	45.87±0.41	**75.75±0.30**	
	I3D		Full	0	47.27	27.88	201.28	51.58±1.86	77.35±0.63	
			SNIP NP	81.09	30.06 (36.4↓)	26.31 (5.6↓)	195.62 (2.8↓)	52.38±3.55	78.32±3.24	
			Random NP	25.32	26.36 (44.2↓)	16.45 (41.0↓)	145.07 (27.9↓)	52.42±2.52	79.05±2.06	
			Layer-wise NP		26.67 (43.6↓)	16.93 (39.3↓)	150.95 (25.0↓)	52.77±1.99	78.41±1.07	
			Vanilla NP		29.93 (36.7↓)	25.76 (7.6↓)	192.42 (4.4↓)	51.57±1.46	78.07±1.34	
		ours	Weighted NP		26.57 (43.8↓)	15.56 (44.2↓)	142.57 (29.2↓)	54.09±0.82	79.26±0.61	
		ours	RANP-m		26.75 (43.4↓)	14.08 (49.5↓)	130.44 (35.2↓)	52.11±3.05	77.54±2.64	
		ours	RANP-f		26.69 (43.5↓)	**13.98 (49.9↓)**	**130.22 (35.3↓)**	**54.27±2.88**	**79.27±2.13**	
								Acc-1↑	Acc-2↑	EPE↓
SceneFlow	PSM		Full	0	19.93	771.02	7217.37	85.97±0.70	94.21±0.33	1.42±0.07
			SNIP NP	90.66	11.09 (44.4↓)	512.97 (33.5↓)	6295.07 (12.8↓)	84.98±0.57	93.71±0.27	1.53±0.08
			Random NP	46.20	8.13 (59.2↓)	371.48 (51.8↓)	5014.83 (30.5↓)	84.09±0.54	92.84±0.28	1.62±0.07
			Layerwise NP		7.71 (61.3↓)	379.05 (50.8↓)	5009.78 (30.6↓)	84.16±0.70	93.82±0.30	1.57±0.07
			Vanilla NP		6.52 (67.3↓)	452.25 (41.3↓)	5653.44 (21.7↓)	**85.24±0.77**	**93.83±0.41**	**1.51±0.09**
		ours	Weighted NP		7.90 (60.4↓)	386.66 (49.9↓)	5002.16 (30.7↓)	84.86±0.74	93.67±0.34	1.56±0.09
		ours	RANP-m		7.96 (60.1↓)	**353.78 (54.1↓)**	**4693.55 (35.0↓)**	84.26±0.73	92.66±0.30	1.62±0.07
		ours	RANP-f		7.93 (60.2↓)	362.77 (53.0↓)	4813.24 (33.3↓)	84.51±0.56	93.47±0.27	1.58±0.07

Table 5.3: *In addition to Table 5.2, with similar GFLOPs or memory on 3D-UNets, our RANP-f achieves the highest accuracy.*

Manner	ShapeNet					BraTS'18						
	Sparsity	Param	GFLOPs	Mem	mIoU	Sparsity	Param	GFLOPs	Mem	ET	TC	WT
Random NP	81.01	2.27	∼7.54	253.12	82.66±0.23	81.08	0.56	∼16.97	685.77	61.09±1.87	68.94±2.44	78.89±2.47
Layer-wise NP	82.82	1.84	∼7.54	255.67	82.82±0.26	83.50	0.46	∼16.97	700.64	70.50±0.63	74.27±0.95	83.63±0.92
Random NP	78.83	2.87	9.57	∼262.66	82.86±0.45	80.90	0.57	17.95	∼729.11	68.45±1.11	70.67±1.21	75.02±0.79
Layer-wise NP	82.81	1.94	8.14	∼262.66	82.52±0.13	82.45	0.51	17.31	∼729.11	70.45±1.03	69.27±1.95	82.42±0.68
RANP-f(ours)	78.24	2.94	7.54	262.66	**83.07±0.22**	78.17	0.76	16.97	729.11	**70.73±0.66**	**74.50±1.05**	**85.45±1.06**

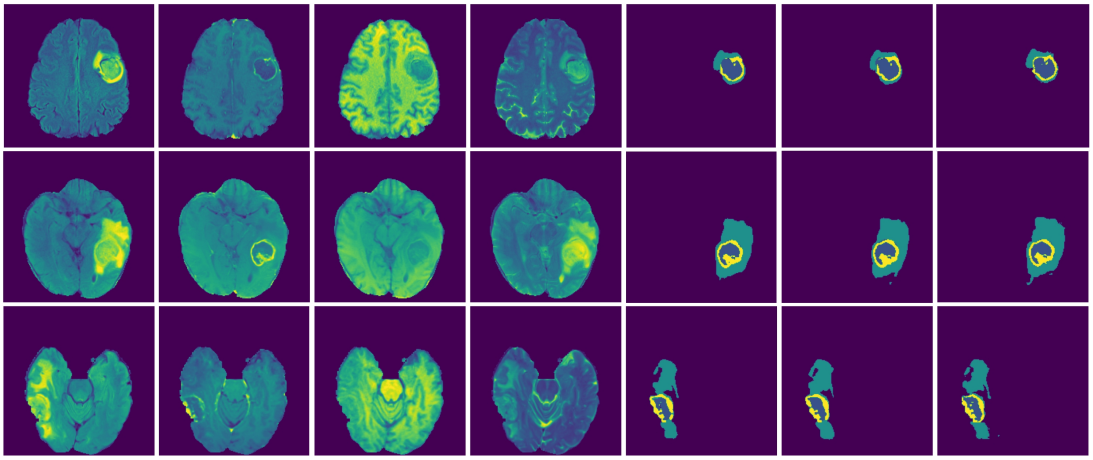


Figure 5.6: *Examples of BraTS'18 for medical image segmentation. Spatial size is* 192^3 *for the MRI image sequences. Slices with distinguishing segmentation are illustrated. (a)-(d) are input slices, (e)-(g) are segmentation with semantics: whole tumor, tumor core, and enhancing tumor (from large size to small one).*

5.6.3.2 BraTS'18

In Table 5.2, RANP-f achieves 96.5% FLOPs, 95.1% parameters, and 80% memory reductions. It further reduces 18.3% FLOPs and 33.3% memory over vanilla NP while increasing -1.21% ET, 5.11% TC, and 0.77% WT. With a similar resource in Table 5.3, RANP achieves higher accuracy than random NP and layer-wise NP.

Additionally, Chen et al. [2020] achieved 2× speedups on BraTS'18 with 3D-UNet. In comparison, our RANP-f has roughly 28× speedups, which is theoretically evidenced by the reduced FLOPs from 478.13G to 16.97G in Table 5.2. We illustrated 2 MRI cases in Fig. 5.6.

5.6.3.3 UCF101

In Table 5.2, for MobileNetV2, RANP-f reduces 55.2% FLOPs, 49% parameters, and 44.1% memory with around 1% accuracy loss. Meanwhile, for I3D, it reduces 49.9% FLOPs, 43.5% parameters, and 35.3% memory with around 2% accuracy increase. The RANP-based methods can reduce much more resources than other methods. We illustrated 2 video categories in Fig. 5.7.

5.6.3.4 SceneFlow.

In Table 5.2, the proposed RANP_f reduces 60.2% parameters, 53.0% FLOPs, and 33.3% memory over the unpruned network with 0.74%-1.46% accuracy loss and 0.16 pixel EPE loss. Specifically, the reweighting scheme in RANP_f boosts the pruning ability of vanilla NP by roughly 10% reductions in FLOPs and memory. Our RANP_f achieves the highest resource reductions than SNIP NP, random NP, and layerwise

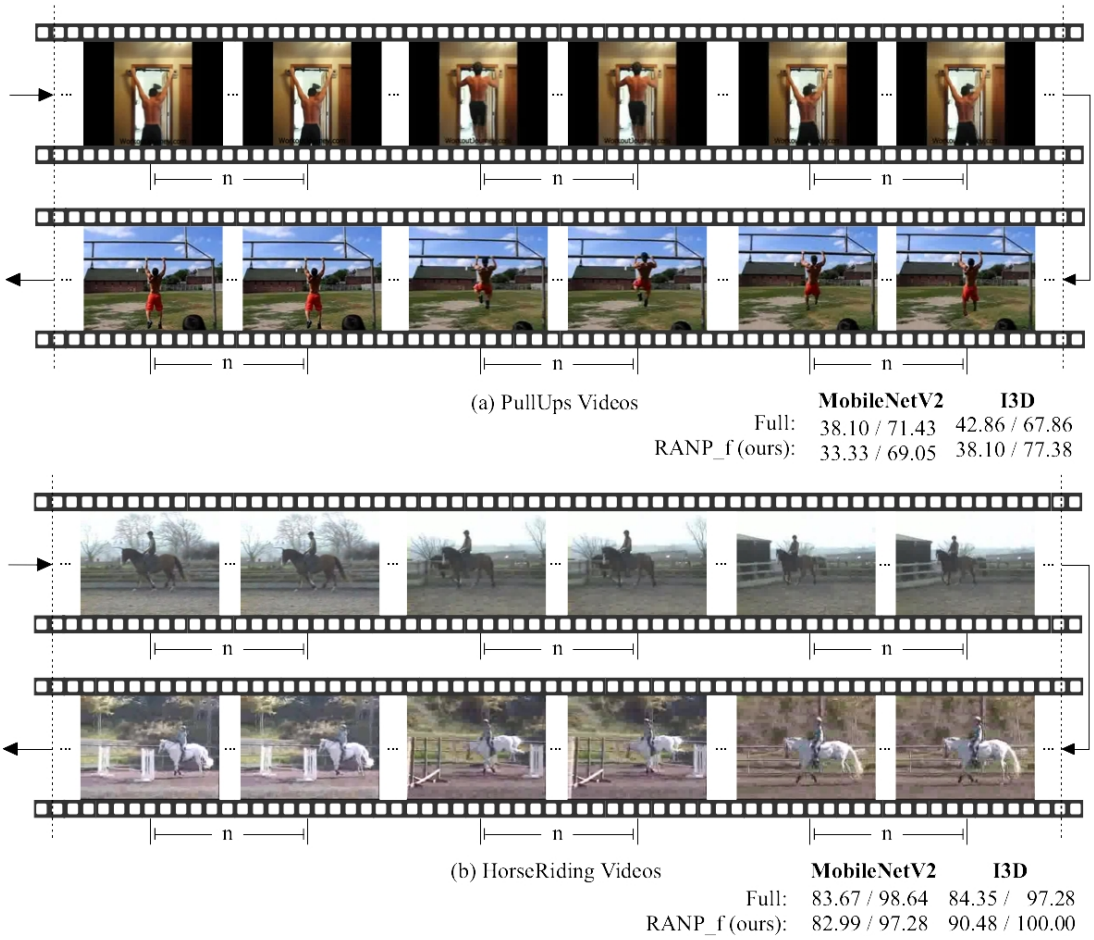


Figure 5.7: *Examples of UCF101 for video classification using MobileNetV2 and I3D models. Illustrations are on 2 video categories, PullUps and HorseRiding. For every different video, 3 samples with each from $n = 16$ frames are used to calculate accuracy. Metrics "* / *" are "Top-1 / Top-5". Pruned networks have different accuracy of different video categories from those of the full networks while the overall accuracy of the whole dataset is in Table 5.2.*

Table 5.4: *Pruning on PSM with 2D-3D CNNs. We mark bold* ***the best*** *and underline the second best. Unit of "Param" and "Mem" is MB. Here, 2D and 3D CNN refer to 2D and 3D convolution layers, pooling layers, and batch normalization layers respectively. Resource consumption caused by activation layers is not counted. Overall, RANP-f has a high ability to reduce resources, mainly FLOPs and memory, on both 2D and 3D CNNs without losing accuracy.*

Manner	Sparsity	2D CNNs			3D CNNs		
		Param	GFLOPs	Mem	Param	GFLOPs	Mem
Full [Chang and Chen, 2018]	0	12.74	174.94	1902.75	7.19	596.08	3595.50
SNIP NP [Lee et al., 2019]	90.66	8.47	119.89	1674.26	**2.63**	393.08	3198.23
Random NP		4.38	60.62	1135.57	3.75	310.86	2791.13
Layerwise NP	46.20	**3.84**	**53.06**	**1051.16**	3.87	325.99	2843.16
RANP-f (ours)		4.09	56.28	1079.32	3.83	**306.49**	**2697.61**

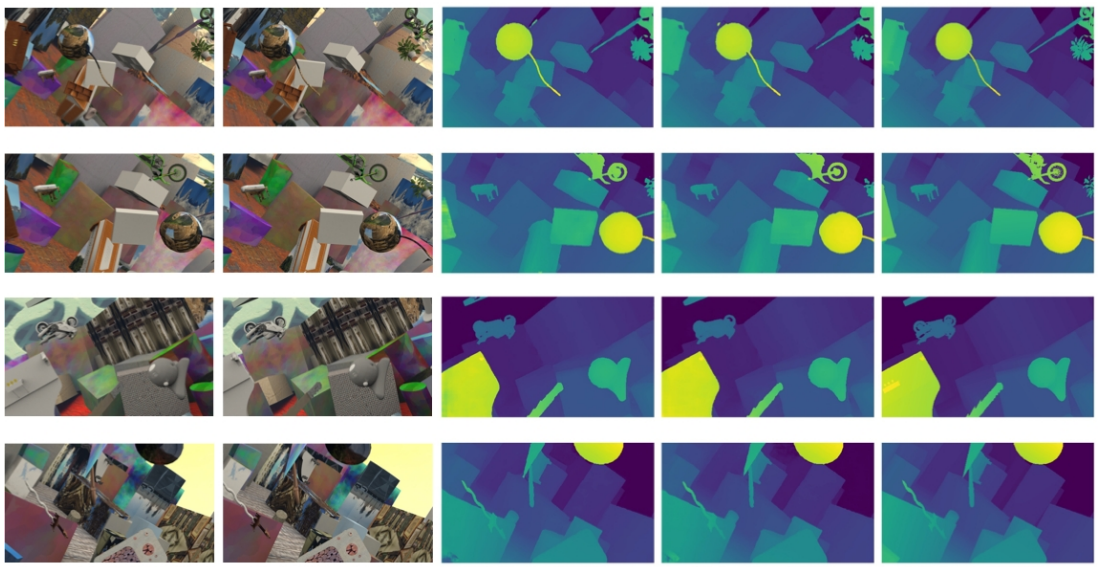


Figure 5.8: *Examples of SceneFlow for two-view stereo matching using PSM. Each raw is for an example. Predicted disparity maps are in (c)-(d) corresponding to the left images in (a). End-point error and accuracy are shown in Table 5.2.*

NP with relatively small accuracy loss. We illustrated 4 examples of SceneFlow validation set in Fig. 5.8.

Moreover, in Table 5.4, we compare the resource consumption of 2D and 3D CNNs in PSM which has a high combination of 2D/3D layers mentioned in "3D CNNs" in Sec. 5.6.1. Although in 2D CNNs, our RANP_f reduces less FLOPs and memory than layerwise NP, these reduced resources in 2D CNNs are much smaller than those in the 3D CNNs while our RANP_f reduces more resources than layerwise NP. Meanwhile, ours can reduce much more resources than SNIP NP and random NP. Table 5.4 indices the generalization of our proposal in both 2D and 3D CNNs.

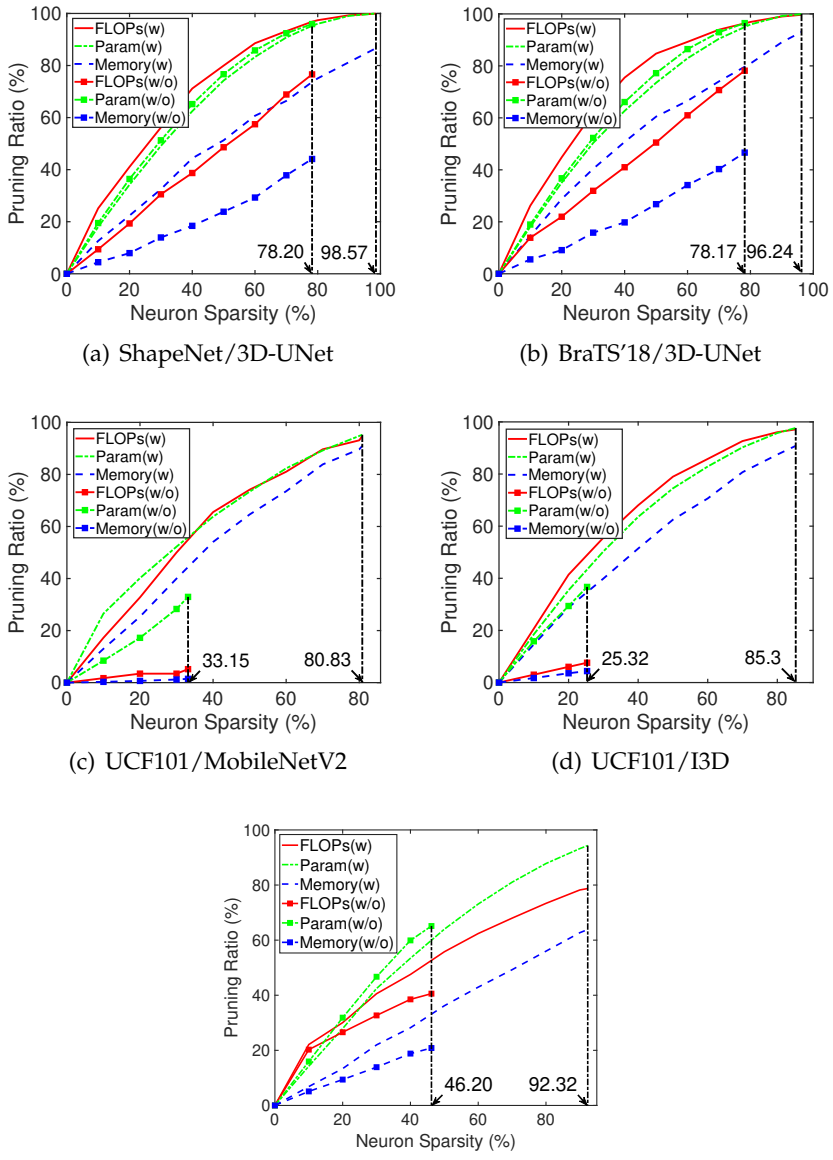
5.6.4 Resources and Accuracy with Neuron Sparsity

Here, we further studied the tendencies of resources and accuracy with an increasing neuron sparsity level from 0 to the maximum one with network feasibility.

5.6.4.1 Resource Reductions

In Figs. 5.9(a)-5.9(d), RANP, marked with (w), achieves much larger FLOPs and memory reductions than vanilla NP, marked with (w/o), due to the balanced distribution of neuron importance by reweighting.

Specifically, for *ShapeNet*, RANP prunes up to 98.57% neurons while only up to 78.24% by vanilla NP in Fig. 5.9(a). For *BraTS'18*, RANP can prune up to 96.24% neurons while only up to 78.17% neurons can be pruned by using vanilla NP in Fig. 5.9(b). For *UCF101*, RANP can prune up to 80.83% neurons compared to 33.15%



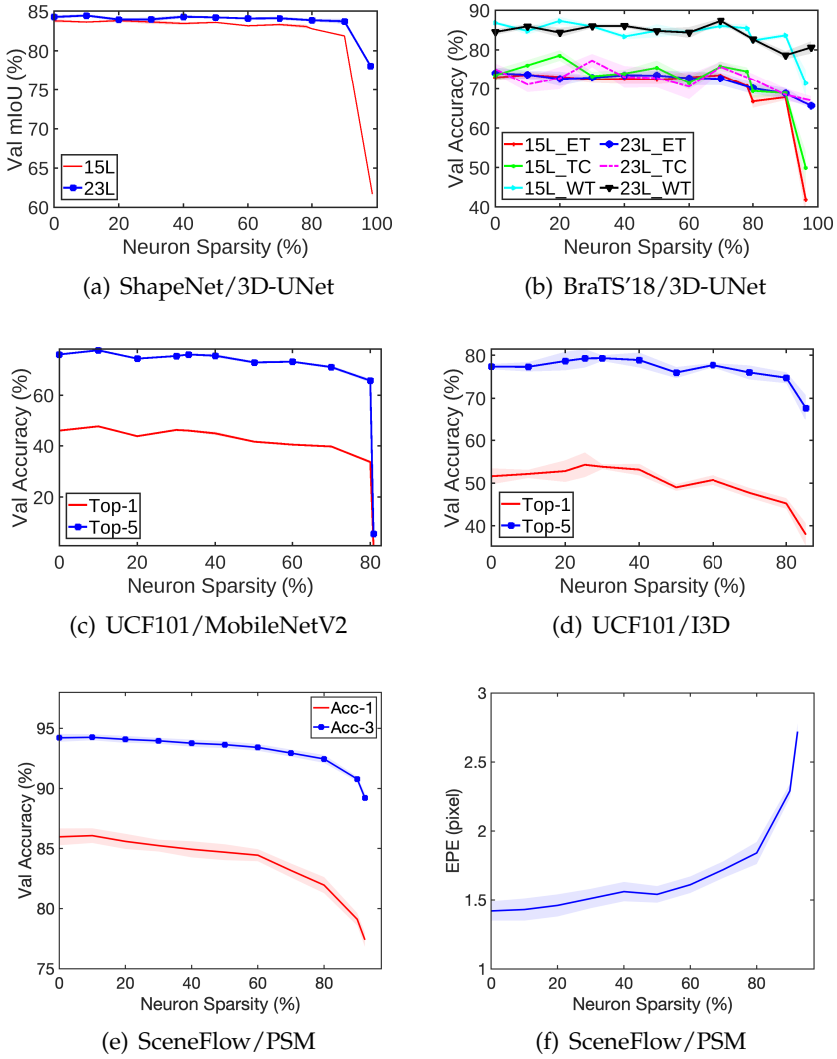
(a) ShapeNet/3D-UNet (b) BraTS'18/3D-UNet

(c) UCF101/MobileNetV2 (d) UCF101/I3D

(e) SceneFlow/PSM

Figure 5.9: *With minimal accuracy loss, more resources are reduced with (w) reweighting by using RANP-f than without (w/o) by using vanilla NP. Best view in color.*

on MobileNetV2 in Fig. 5.9(c), and 85.3% neurons compared to 25.32% on I3D in Fig. 5.9(d). For *SceneFlow* in Fig. 5.9(e), the reweighting scheme induced in RANP_f greatly reduces the resource consumptions with the maximum neuron sparsity 92.32% while only up to 46.20% neuron sparsity can be achieved by vanilla NP.



(a) ShapeNet/3D-UNet (b) BraTS'18/3D-UNet

(c) UCF101/MobileNetV2 (d) UCF101/I3D

(e) SceneFlow/PSM (f) SceneFlow/PSM

Figure 5.10: *Accuracy with sparsity. Best view in color.*

5.6.4.2 Accuracy with Pruning Sparsity

For *ShapeNet* in Fig. 5.10(a), the 23-layer 3D-UNet achieves a higher mIoU than the 15-layer one. Extremely, when pruned with the maximum neuron sparsity 97.99%, it can achieve 78.10% mIoU. With the maximum neuron sparsity 98.57%, however, the 15-layer 3D-UNet achieves only 61.42%.

For *BraTS'18* in Fig. 5.10(b), the 23-layer 3D-UNet does not always outperform the 15-layer one and has a larger fluctuation which could be caused by the limited training samples. Nevertheless, even in the extreme case, the 23-layer 3D-UNet has a small accuracy loss. Clearly, RANP makes it feasible to use deeper 3D-UNets without the memory issue.

For *UCF101* in Figs. 5.10(c)-5.10(d), RANP-f achieves <3% accuracy loss at 70% neuron sparsity, indicating its effectiveness of greatly reducing resources with small

accuracy loss.

For *SceneFlow* in Figs. 5.10(e) and 5.10(f), pruning at roughly 70% neuron sparsity achieves 2.8% decrease in Acc-1, 1.27% decrease in Acc-3, and 0.3 pixels decrease in EPE while reducing 68.0% FLOPs and 49.3% memory.

5.6.5 Transferability with Interactive Model

Table 5.5: *Transfer learning by 23-layer 3D-UNets interactively pruned and trained between ShapeNet and BraTS'18. The accuracy loss from RANP-f to T-RANP-f is negligible. "T": transferred.*

Manner	ShapeNet	BraTS'18		
	mIoU(%)	ET(%)	TC(%)	WT(%)
Full [Cicek et al., 2016]	**84.27±0.21**	**74.04±1.45**	**75.11±2.43**	84.49±0.74
RANP-f(ours)	83.86±0.15	71.13±1.43	72.40±1.48	83.32±0.62
T-RANP-f(ours)	83.25±0.17	72.74±0.69	73.25±1.69	**85.22±0.57**

In this experiment, we trained on ShapeNet with a transferred 3D-UNet by RANP on BraTS'18 with 80% neuron sparsity. Interactively, with the same neuron sparsity, a transferred 3D-UNet by RANP on ShapeNet was applied to training on BraTS'18. Results in Table 5.5 demonstrate that training with transferred models across different datasets can largely maintain high or higher accuracy.

5.6.6 Lightweight Training on a Single GPU

Table 5.6: *ShapeNet: a deeper 23-layer 3D-UNet is achievable on a single GPU with 80% neuron pruning.*

Manner	Layer	Batch	GPU(s)	Sparsity(%)	mIoU(%)
Full [Cicek et al., 2016]	15	12	2	0	83.79±0.21
Full [Cicek et al., 2016]	23	12	2	0	84.27±0.21
RANP-f(ours)	23	12	**1**	80	**84.34±0.21**

RANP with high neuron sparsity makes it feasible to train with large data size on a single GPU due to the largely reduced resources. We trained on ShapeNet with the same batch size 12 and spatial size 64^3 in Sec. 5.6.1 by using a 23-layer 3D-UNet with 80% neuron sparsity on a single GPU. With this setup, RANP-f reduces $\sim 35\times$ GFLOPs (from 259.59 to 7.39) and $\sim 3.9\times$ memory (from 1005.96MB to 255.57MB), making it feasible to train on a single GPU instead of 2 GPUs. It achieves a higher mIoU, 84.34±0.21%, than the 15-layer and 23-layer full 3D-UNets in Table 5.6.

The accuracy increase is due to the enlarged batch size on each GPU. With limited memory, however, training a 23-layer full 3D-UNet on a single GPU is infeasible.

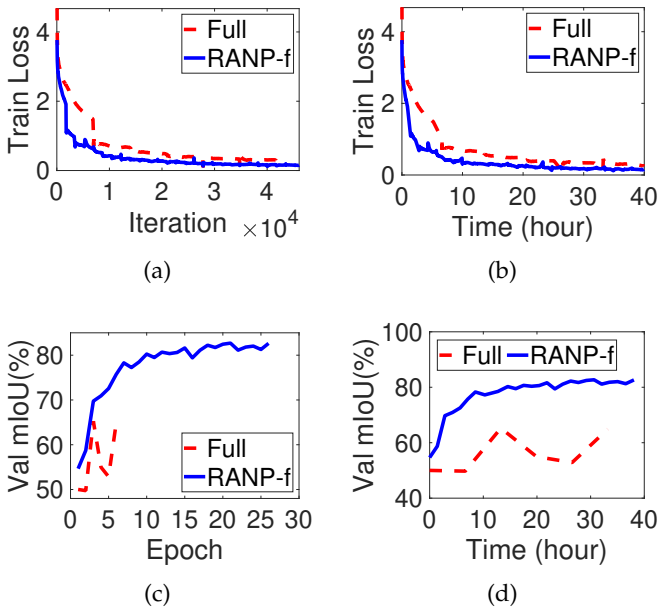


Figure 5.11: *ShapeNet: a faster convergence on a single GPU with a 23-layer 3D-UNet and an increased batch size due to the largely reduced resources by using RANP-f. The batch size is 1 for "Full" and 4 for "RANP-f". Experiments run for 40 hours.*

5.6.7 Fast Training with Increased Batch Size

Here, we used the largest spatial size 128^3 of one sample on a single GPU and then extended it to RANP with an increased batch size from 1 to 4 to fully fill the GPU capacity. The initial learning rate was reduced from 0.1 to 0.01 due to the batch size decreased from 12 in Table 5.6. This is to avoid an immediate increase in the training loss right after 1st epoch due to the unsuitably large learning space.

In Fig. 5.11(a), RANP-f enables an increased batch size 4 and achieves a faster loss convergence than the full network. In Fig. 5.11(c), the full network executed 6 epochs while RANP-f reached 26 epochs. Vividly shown by training time in Figs. 5.11(b) and 5.11(d), RANP-f has much lower loss and higher accuracy than the full one. This greatly indicates the practical advantage of RANP on fastening training convergence.

5.6.8 Solving Layer-wise Neuron Imbalance Issue

The imbalanced layer-wise distribution of neuron importance hinders pruning at a high sparsity level due to the pruning of the whole layer(s). For 2D classification tasks in Lee et al. [2020], the orthogonal initialization is used to effectively solve this problem for balancing the importance of parameters; but it does not improve our neuron pruning results in 3D tasks and even leads to a poor pruning capability with a lower maximum neuron sparsity than the Glorot initialization [Glorot and Bengio, 2010]. Here, we compare the resource reducing capability by using the Glorot initialization and the orthogonal initialization.

Table 5.7: *Impact of parameter initialization on neuron pruning. "ort": orthogonal initialization; "xn": Glorot initialization; "f": FLOPs. "Sparsity" is the least maximum neuron sparsity among all manners to ensure the network feasibility. RANP-f with the Glorot initialization achieves the least FLOPs and memory consumptions.*

Dataset(Model)	Manner	Sparsity(%)	Param(MB)	GFLOPs	Mem(MB)
ShapeNet (3D-UNet)	Full [Cicek et al., 2016]	0	62.26	237.85	997.00
	Vanilla-ort [Lee et al., 2020]	70.53	4.40	72.65	630.00
	Vanilla-xn		4.56	73.22	618.35
	RANP-f-ort		5.40	21.73	366.29
	RANP-f-xn		5.52	**15.06**	**328.66**
BraTS'18 (3D-UNet)	Full [Cicek et al., 2016]	0	15.57	478.13	3628.00
	Vanilla-ort [Lee et al., 2020]	72.20	0.95	159.91	2240.33
	Vanilla-xn		0.92	130.28	2109.19
	RANP-f-ort		1.24	33.28	967.56
	RANP-f-xn		1.29	**23.31**	**850.56**
UCF101 (MobileNetV2)	Full [Sandler et al., 2018]	0	9.47	0.58	157.47
	Vanilla-ort [Lee et al., 2020]	30.21	6.80	0.56	155.71
	Vanilla-xn		6.77	0.55	155.48
	RANP-f-ort		5.12	0.32	105.88
	RANP-f-xn		5.19	**0.28**	**94.50**
UCF101 (I3D)	Full [Carreira and Zisserman, 2017]	0	47.27	27.88	201.28
	Vanilla-ort [Lee et al., 2020]	24.24	30.56	25.83	192.70
	Vanilla-xn		30.64	25.85	192.88
	RANP-f-ort		27.39	15.94	144.10
	RANP-f-xn		27.38	**14.63**	**133.80**
SceneFlow (PSM)	Full [Chang and Chen, 2018]	0	19.93	771.02	7217.37
	Vanilla-ort [Lee et al., 2020]	36.76	8.66	485.98	5913.52
	Vanilla-xn		8.75	486.92	5947.86
	RANP-f-ort		10.03	422.34	5332.72
	RANP-f-xn		9.96	**420.77**	**5313.30**

Resource reductions. In Table 5.7, vanilla neuron pruning (*i.e.*, MPMG-sum) with the Glorot initialization, *i.e.*, vanilla-xn, achieves smaller FLOPs and memory consumption than those with the orthogonal initialization, *i.e.*, vanilla-ort, except FLOPs with 3D-UNet on ShapeNet and I3D on UCF101. This exception of I3D on UCF101 is possibly caused by the high ratio of 1^3 kernel size filters in I3D, *i.e.*, 37 out of 57 convolutional layers, because those 1^3 kernel size filters can be regarded as 2D filters on which the orthogonal initialization can effectively deal with Lee et al. [2020]. While this ratio is also high in MobileNetV2, *i.e.*, 34 out of 52 convolutional layers, it is unnecessary to have the same problem as I3D since it is also affected by the number of neurons in each layer. Note that since 3D-UNets used are all with 3^3 kernel size filters, the orthogonal initialization for 3D-UNet in most cases is inferior to the Glorot initialization according to our experiments.

Meanwhile, the exception of PSM on SceneFlow is possibly caused by the massive cross-layer additions, which breaks the independence of nonadjacent layers such that retained neurons in each layer have a strong connection with multiple layers, as described in "3D CNNs" in Sec. 5.6.1.

Furthermore, in Table 5.7, the gap between vanilla-ort and vanilla-xn is very small

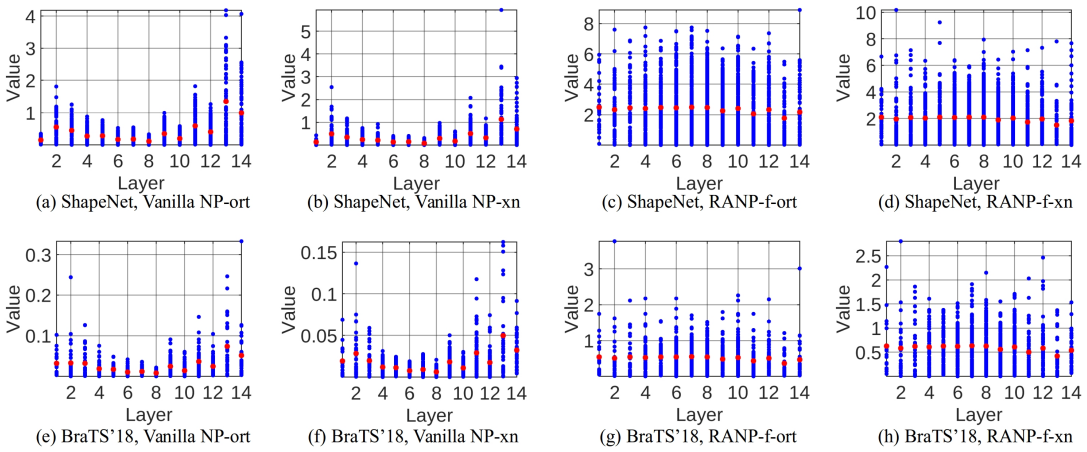


Figure 5.12: *Neuron importance of a 15-layer 3D-UNet by using MPMG-sum with the orthogonal and the Glorot initialization. Blue: neuron values; red: mean values. By using the vanilla NP, the orthogonal initialization does not result in a balanced neuron importance distribution compared to the Glorot initialization whereas by using our RANP-f, the values are more balanced and resource-aware on FLOPs, enabling a pruning at the extreme sparsity.*

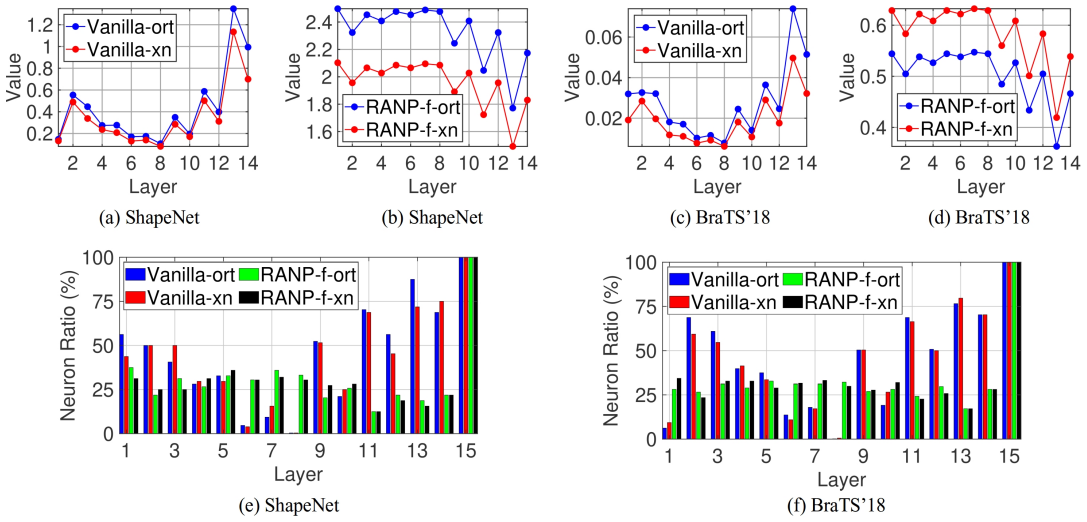


Figure 5.13: *Comparison of neuron distribution with the orthogonal and the Glorot initialization before and after the reweighting. (a)-(d) are neuron importance values. (e)-(f) are neuron retained ratios. Vanilla versions (both the orthogonal and the Glorot initialization) prune all the neurons in the 8th layer, leading to network infeasibility while our RANP-f versions have a balanced distribution of retained neurons.*

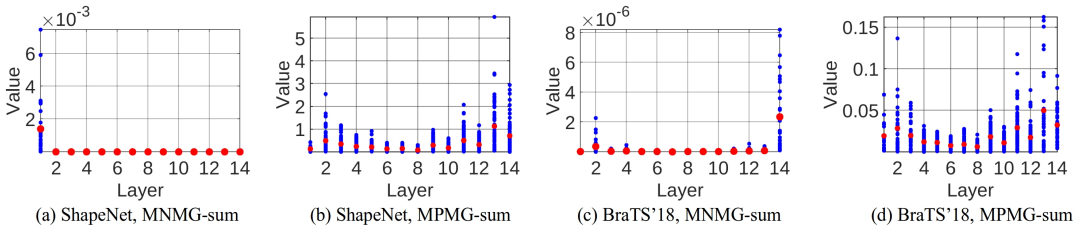
on MobileNetV2 and I3D. Nevertheless, with RANP-f and the Glorot initialization, *i.e.*, RANP-f-xn, more FLOPs and memory can be reduced than using the orthogonal initialization, *i.e.*, RANP-f-ort.

Balance of Neuron Importance Distribution. More importantly, by using the

reweighting scheme of RANP in Fig. 5.12, the values of neuron importance are more balanced and stable than those of vanilla neuron importance. This can largely avoid network infeasibility without pruning the whole layer(s).

Now, we analyse the neuron distribution from the observation of neuron importance values and network structures. *Fig. 5.13 illustrates a detailed comparison between the orthogonal and the Glorot initialization by each two subfigures in column of Fig. 5.12.* In Figs. 5.13(a)-5.13(c), vanilla neuron importance by using the Glorot initialization is more stable and compact than that by using the orthogonal initialization. After applying the reweighting scheme of RANP-f, the importance tends to be in a similar tendency, shown in Figs. 5.13(b)-5.13(d). Consequently, in Figs. 5.13(c)-5.13(f), the neuron ratios are more balanced after the reweighting than without reweighting, especially the 8th layer. Thus, we choose the Glorot initialization as network initialization. Note that we adopt the same neuron sparsity for these two initialization experiments in Table 5.7 and Fig. 5.13.

5.6.9 Visualization of Balanced Neuron Distribution with RANP



(a) ShapeNet, MNMG-sum (b) ShapeNet, MPMG-sum (c) BraTS'18, MNMG-sum (d) BraTS'18, MPMG-sum

Figure 5.14: *MNMG-sum and MPMG-sum on ShapeNet and BraTS'18 with the maximum neuron sparsity in Table 5.1. Blue: neuron values; red: mean values. Clearly, neuron importance distribution with MPMG-sum is more balanced than that with MNMG-sum.*

In Fig. 5.14, neuron importance by using MPMG-sum is more balanced than by using MNMG-sum, which avoids pruned networks by using MPMG-sum to be infeasible, that is at least 1 neuron will be retained in each layer.

In addition to the distribution of retained neuron ratios in Fig. 5.3, which is also shown in the first row of Fig. 5.15, the last row of Fig. 5.15 is for BraTS'18. Moreover, Fig. 5.16 illustrates the distribution of neurons retained in each layer by using vanilla neuron pruning (*i.e.*, vanilla NP) and RANP-f compare to the full network.

Clearly, upon pruning, neurons in each layer are largely reduced except the last layer where all neurons are retained for the number of segmentation classes. In Fig. 5.16, vanilla NP has very few neurons in, *e.g.*, the 8th layer, resulting in low accuracy or network infeasibility. By contrast, the neuron distribution with RANP-f is more balanced to improve the pruning capability.

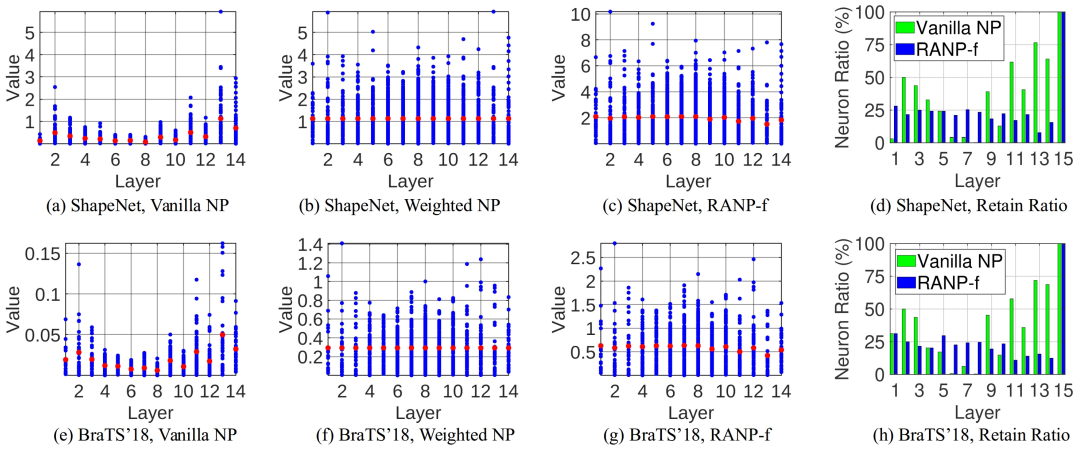


Figure 5.15: *Balanced neuron importance distribution with MPMG-sum on ShapeNet and BraTS'18. Neuron sparsity is 78.24% on ShapeNet and 78.17% on BraTS'18. Blue: neuron values; red: mean values.*

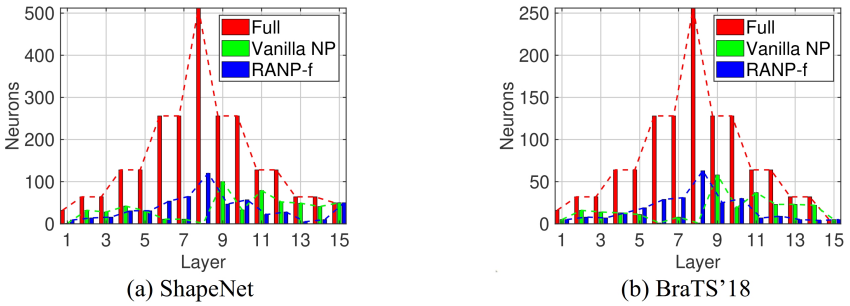


Figure 5.16: *Layer-wise neuron distribution of 3D-UNets.*

5.7 Conclusion

In this chapter, we propose an effective resource aware neuron pruning method, RANP, for 3D CNNs. RANP prunes a network at initialization by greatly reducing resources with negligible loss of accuracy. Its resource aware reweighting scheme balances the neuron importance distribution in each layer and enhances the pruning capability of removing a high ratio, say 80% on 3D-UNet, of neurons with minimal accuracy loss. This advantage enables training deep 3D CNNs with a large batch size to improve accuracy and achieving lightweight training on one GPU.

Our experiments on 3D semantic segmentation using ShapeNet and BraTS'18, video classification using UCF101, and two-view stereo matching using SceneFlow demonstrate the effectiveness of RANP by pruning 70%-80% neurons with minimal loss of accuracy. Moreover, the transferred models pruned on a dataset and trained on another one are succeeded in maintaining high accuracy, indicating the high transferability of RANP. Meanwhile, the largely reduced computational resources enable lightweight and fast training on one GPU with increased batch size.

Chapter 6

Conclusion

In this thesis, we first introduce the background of MRF optimization algorithms and CNN pruning in Chapter 2. Then, we propose solutions to three problems: inefficient and ineffective MRF inference algorithms in deep learning, inferior joint learning methods for multiple CNNs, and high resource requirements of CNNs, forming Chapter 3, Chapter 4, and Chapter 5 respectively.

6.1 Summary

We summarise our works in Chapters 3-5 according to the proposed method corresponding to "Problem Setup" in Chapter 1.

In *Chapter 3*, we propose two message passing layers, namely ISGMR and TRWP, with advantages of their higher energy minimization ability and higher efficiency using CUDA programming on massive parallel trees over comparable MRF optimization algorithms, that is mean-field, SGM, and TRWS.

Specifically, ISGMR achieves much lower energies than the baseline SGM due to the solution of removing the overcounted unary potentials and the iterations in the energy minimization. Meanwhile, TRWP has a much higher efficiency than the baseline TRWS by tree decomposition without losing the energy minimization ability given sufficient iterations, such as 50 iterations.

The proposed ISGMR and TRWP are implemented in CUDA on parallel trees, and thus, are at least 7 and 700 times faster than our CPU single and multiple versions and PyTorch GPU versions in the forward and backward propagation respectively.

Our experiments of MRF energy minimization on stereo vision and image denoising and inpainting validate that our methods outperform mean-field and SGM, and comparable to TRWS. Moreover, applying our methods into deep semantic segmentation with Canny edges improves the accuracy with accurate object edges.

In *Chapter 4*, we propose a transparent initialization and a sparse encoder for joint learning a superpixel CNN and a semantic segmentation CNN. The transparent initialization method identically maps the outputs of additional CNN layers, fully-connected layers in our case, to the input at the initialization stage. Then, a small learning rate is used to finetune the additional layers and pretrained CNNs. Meanwhile, a sparse encoder is used to index the pixels with the indices of superpixels to

reduce the GPU memory and computational complexity using sparse matrix operations, where most of the values are zero, instead of dense matrix operations.

Experiments on semantic segmentation using PASCAL VOC 2012, PASCAL Context, and ADE20K datasets validated the effectiveness of our method. The evaluation of segmented object edges using performance ratio and F-measure further indicates that our method outperforms the current state-of-the-art methods.

In *Chapter 5*, we propose a single-shot neuron pruning with resource constraints for 3D CNNs, also applicable to 2D CNNs, to largely reduce the high requirements of training resources, mainly GPU memory and FLOPs. This method firstly calculates the neuron importance using learned neuron connection sensitivity. Then, the neuron importance in each layer is balanced using a multiplication factor from the maximum mean value of neuron importance among all layers. To embed the constraint of training resources into our pruning to further reduce resource consumption, either to reduce more GPU memory or FLOPs, a reweighting strategy based on GPU memory or FLOPs is introduced.

Our experiments on 3D semantic segmentation, video classification, and two-view stereo matching validate the high pruning ability and resource reduction capability of our method, decreasing 35%-80% GPU memory and 50%-90% FLOPs, without losing the task accuracy. This method can also be generalized to 2D CNNs for any tasks.

The overall contribution of this thesis starts from the introduction of differentiable message passing algorithms in Chapter 3 and transparent initialization module in Chapter 4 for fusion learning. While both could cause the increase of resources, mainly FLOPs and GPU memory, required at the training phase, especially in the fusion learning, one can refer to RANP in Chapter 5 to prune unimportant network channels to largely reduce the resource consumption.

6.2 Future Works

We extend the aforementioned works to three possible future works according to the current limitations of their application or setting.

For the message passing layers in *Chapter 3*, we currently validate them on deep semantic segmentation using Canny edges. Nevertheless, traditional MRF optimization algorithms are not limited to a specific task. Hence, it would be promising to apply our methods to other deep learning tasks, such as optical flow and stereo vision, with different types of edges, such as learnable edges from CNNs.

For the transparent initialization in *Chapter 4*, the currently supported layers are fully-connected layers and convolutional layers with 1^3 kernel size. It could be extended to general convolutional layers with an arbitrary kernel size such that the proposed transparent initialization is more generalized.

For the resource aware neuron pruning in *Chapter 5*, the neuron sparsity used in the CNNs are manually selected from the observation of the relation between different neuron sparsity and their validation accuracy. However, running all possible

settings for the neuron sparsity involves multiple training. Therefore, it would be desirable to automatically estimate a suitable neuron sparsity for the trade-off between largely reduced training resources and a small loss of task accuracy.

Bibliography

Achanta, R.; Shaji, A.; Smith, K.; Lucchi, A.; Fua, P.; and Susstrunk, S., 2012. SLIC superpixels compared to state-of-the-art superpixel methods. *TPAMI*, 34, 11 (2012), 2274–2282. (cited on pages 59, 60, 61, 62, and 63)

Achanta, R. and Susstrunk, S., 2017. Superpixels and polygons using simple non-iterative clustering. *CVPR*, (2017). (cited on page 61)

Ajanthan, T.; Desmaison, A.; Rudy, R.; Salzmann, M.; Torr, P.; and Kumar, M., 2017. Efficient linear programming for dense CRFs. *CVPR*, (2017). (cited on page 61)

Ajanthan, T.; Hartley, R.; and Salzmann, M., 2016. Memory efficient max-flow for multi-label submodular MRFs. *CVPR*, (2016). (cited on page 29)

Ajanthan, T.; Hartley, R.; Salzmann, M.; and Li, H., 2015. Iteratively reweighted graph cut for multi-label MRFs with non-convex priors. *CVPR*, (2015). (cited on pages 29 and 46)

Andres, B., 2015. Lifting of multicuts. *CoRR abs/1503.03791*, (2015). (cited on page 15)

Arbelaez, P.; Maire, M.; Fowlkes, C.; and Malik, J., 2010. Contour detection and hierarchical image segmentation. *TPAMI*, 33, 5 (2010), 898–916. (cited on pages 63 and 70)

Badrinarayanan, V.; Kendall, A.; and Cipolla, R., 2017. SegNet: A deep convolutional encoder-decoder architecture for image segmentation. *TPAMI*, 39, 12 (2017), 2481–2495. (cited on pages 6 and 61)

Bakas, S.; Akbari, H.; Sotiras, A.; Bilello, M.; Rozycki, M.; Kirby, J.; Freymann, J.; Farahani, K.; and Davatzikos, C., 2017. Advancing the cancer genome atlas glioma MRI collections with expert segmentation labels and radiomic features. *Nature Scientific Data*, (2017). (cited on pages 91 and 102)

Battaglia1, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; Gulcehre, C.; Song, F.; Ballard, A.; Gilmer, J.; Dahl, G.; Vaswani, A.; Allen, K.; Nash, C.; Langston, V.; Dyer, C.; Heess, N.; Wierstra, D.; Kohli, P.; Botvinick, M.; Vinyals, O.; Li, Y.; and Pascanu, R., 2018. Relational inductive biases, deep learning, and graph networks. *arXiv:1806.01261*, (2018). (cited on pages 23 and 29)

Beier, T.; Andres, B.; Kothe, U.; and Hamprecht, F., 2016. An efficient fusion move algorithm for theminimum cost lifted multicut problem. *ECCV*, (2016). (cited on pages 14 and 16)

Bergh, M.; Boix, X.; Roig, G.; and Gool, L., 2015. SEEDS: Superpixels extracted via energydriven sampling. *IJCV*, (2015). (cited on page 61)

Besag, J., 1986. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society, Series B*, 48, 3 (1986), 259—302. (cited on page 17)

Boykov, Y. and Jolly, M., 2011. Interactive graph cuts for optimal boundary and region segmentation of objects in N-D images. *ICCV*, (2011). (cited on page 27)

Boykov, Y.; Veksler, O.; and Zabih, R., 1998. Markov random fields with efficient approximations. *CVPR*, (1998). (cited on page 11)

Boykov, Y.; Veksler, O.; and Zabih, R., 2001a. Fast approximate energy minimization via graph cuts. *TPAMI*, 23, 11 (2001), 1222–1239. (cited on page 11)

Boykov, Y.; Veksler, O.; and Zabih, R., 2001b. Fast approximate energy minimization via graph cuts. *TPAMI*, (2001). (cited on page 29)

Carr, P. and Hartley, R., 2009. Solving multilabel graph cut problems with multilabel swap. *DICTA*, (2009). (cited on page 29)

Carreira, J. and Zisserman, A., 2017. Quo vadis, action recognition? a new model and the Kinetics dataset. *CVPR*, (2017). (cited on pages 91, 103, and 115)

Chang, J. and Chen, Y., 2018. Pyramid stereo matching network. *CVPR*, (2018). (cited on pages 91, 104, 105, 109, and 115)

Chen, C.; Tung, F.; Vedula, N.; and Mori, G., 2018a. Constraint-aware deep neural network compression. *ECCV*, (2018). (cited on page 91)

Chen, H.; Wang, Y.; Shu, H.; Tang, Y.; Xu, C.; Shi, B.; Xu, C.; Tian, Q.; and Xu, C., 2020. Frequency domain compact 3D convolutional neural networks. *CVPR*, (2020). (cited on pages 91, 92, and 108)

Chen, L.; Barron, J.; Papandreou, G.; Murphy, K.; and Yuille, A., 2016a. Semantic image segmentation with task-specific edge detection using CNNs and a discriminatively trained domain transform. *CVPR*, (2016). (cited on pages 59 and 61)

Chen, L.; Papandreou, G.; Kokkinos, I.; Murphy, K.; and Yuille, A., 2017. DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution,and fully connected CRFs. *TPAMI*, 40, 4 (2017), 834–848. (cited on page 59)

Chen, L.; Zhu, Y.; Papandreou, G.; Schroff, F.; and Adam, H., 2018b. Encoder-decoder with atrous separable convolution for semantic image segmentation. *ECCV*, (2018). (cited on pages xviii, 49, 51, 59, 61, 62, 63, 71, 73, and 86)

Chen, T.; Goodfellow, I.; and Shlens, J., 2016b. Net2Net: Accelerating learning via knowledge transfer. *ICLR*, (2016). (cited on pages x, 7, 64, 68, 69, 77, and 85)

Chen, Y.; Kalantidis, Y.; Li, J.; Yan, S.; and Feng, J., 2018c. A2-nets: Double attention networks. *NeurIPS*, (2018). (cited on page 61)

Chopra, S. and Rao, M., 1993. The partition problem. *Math. Program.*, 59, 1–3 (1993), 87–115. (cited on page 14)

Cicek, O.; Abdulkadir, A.; Lienkamp, S.; Brox, T.; and Ronneberger, O., 2016. 3D U-Net: Learning dense volumetric segmentation from sparse annotation. *MICCAI*, (2016). (cited on pages 89, 91, 103, 113, and 115)

Dagum, L. and Menon, R., 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science and Engineering*, (1998). (cited on page 35)

Deng, J.; Dong, W.; Socher, R.; Li, L.; Li, K.; and Li, F., 2009. ImageNet: A large-scale hierarchical image database. *CVPR*, (2009). (cited on page 61)

Domke, J., 2013. Learning graphical model parameters with approximate marginal inference. *TPAMI*, (2013). (cited on page 29)

Dong, X.; Chen, S.; and Pan, S., 2017. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *NeurIPS*, (2017). (cited on pages 89 and 91)

Drory, A.; Haubold, C.; Avidan, S.; and Hamprecht, F. A., 2014. Semi-global matching: a principled derivation in terms of message passing. *Pattern Recognition*, (2014). (cited on pages 28 and 31)

Everingham, M.; Eslami, S.; Gool, L.; Williams, C.; Winn, J.; and Zisserman, A., 2014. The pascal visual object classes challenge a retrospective. *IJCV*, (2014). (cited on pages 49, 63, and 69)

Facciolo, G.; Franchis, C.; and Meinhardt, E., 2015. MGM: A significantly more global matching for stereo vision. *BMVC*, (2015). (cited on pages 31 and 47)

Feng, D.; Schutz, C.; Rosenbaum, L.; Hertlein, H.; Glaser, C.; Tim, F.; Wiesbeck, W.; and Dietmayer, K., 2020. Deep multi-model object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges. *IEEE Transactions on Intelligent Transportation systems*, (2020), 1–20. (cited on page 59)

Flach, P. and Kull, M., 2015. Precision-recall-gain curves: Pr analysis done right. *NeurIPS*, (2015). (cited on pages 60, 71, and 73)

Fu, J.; Liu, J.; Tian, H.; Li, Y.; Bao, Y.; Fang, Z.; and Lu, H., 2019. Dual attention network for scene segmentation. *CVPR*, (2019). (cited on page 61)

Gadde, R.; Jampani, V.; Kiefel, M.; Kappler, D.; and Gehler, P., 2016. Superpixel convolutional networks using bilateral inceptions. *ECCV*, (2016). (cited on pages 59 and 62)

Glorot, X. and Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, (2010). (cited on pages x, 7, 68, 85, 104, and 114)

Gordon, A.; Eban, E.; Nachum, O.; Chen, B.; Wu, H.; Yang, T.; and Choi, E., 2018. Morphnet: Fast & simple resource-constrained structure learning of deep networks. *CVPR*, (2018). (cited on page 92)

Gould, S.; Zhao, J.; He, X.; and Zhang, Y., 2014. Superpixel graph label transfer with learned distance metric. *ECCV*, (2014). (cited on page 62)

Graham, B.; Engelcke, M.; and Maaten, L., 2018. 3D semantic segmentation with submanifold sparse convolutional networks. *CVPR*, (2018). (cited on pages 89, 92, and 103)

Grotschel, M. and Wakabayashi, Y., 1989. A cutting plane algorithm for a clustering problem. *Math. Program.*, 45, 1 (1989), 59–96. (cited on page 14)

Guo, Y.; Yao, A.; and Chen, Y., 2016. Dynamic network surgery for efficient dnns. *NeurIPS*, (2016). (cited on pages 89 and 91)

Han, S.; Pool, J.; Tran, J.; and Dally, W., 2015. Learning both weights and connections for efficient neural network. *NeurIPS*, (2015). (cited on pages 89 and 91)

Hariharan, B.; Arbelaez, P.; Bourdev, L.; Maji, S.; and Malik, J., 2011. Semantic contours from inverse detectors. *ICCV*, (2011). (cited on pages 49, 59, 63, and 69)

Hartley, R. and Ajanthan, T., 2018. Generalized range moves. *arXiv:1811.09171*, (2018). (cited on page 29)

Hartley, R. and Zisserman, A., 2003. Multiple view geometry in computer vision. *Cambridge university press*, (2003). (cited on page 46)

Hassner, M. and Sklansky, J., 1980. The use of Markov random fields as models of texture. *Computer Graphics and Image Processing*, (1980). (cited on page 27)

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2016. Deep residual learning for image recognition. *CVPR*, (2016). (cited on page 61)

He, Y.; Lin, J.; Liu, Z.; Wang, H.; Li, L.; and Han, S., 2018. Amc: Automl for model compression and acceleration on mobile devices. *ECCV*, (2018). (cited on page 91)

He, Y.; Zhang, X.; and Sun, J., 2017. Channel pruning for accelerating very deep neural networks. *ICCV*, (2017). (cited on pages 89, 91, and 92)

Hernandez-Juare, D.; Chacon, A.; Espinosa, A.; Vazquez, D.; Moure, J.; and opez, A. M. L., 2016. Embedded real-time stereo estimation via semi-global matching on the GPU. *International Conference on Computational Sciences*, (2016). (cited on page 32)

Hirschmuller, H., 2008. Stereo processing by semiglobal matching and mutual information. *TPAMI*, (2008). (cited on pages 11, 17, 27, 28, 29, 31, and 51)

Hofmarcher, M.; Unterthiner, T.; Medina, J.; Klambauer, G.; Hochreiter, S.; and Nessler, B., 2019. Visual scene understanding for autonomous driving using semantic segmentation. *Explainable AI. LNCS*, 11700 (2019), 285–296. (cited on page 59)

Hong, S.; Yan, X.; Huang, T.; and Lee, H., 2018. Learning hierarchical semantic image manipulation through structured representations. *NeurIPS*, (2018). (cited on page 59)

Hou, R.; Chen, C.; Sukthankar, R.; and Shah, M., 2019. An efficient 3D CNN for action/object segmentation in video. *BMVC*, (2019). (cited on page 89)

Huang, G.; Liu, Z.; Maaten, L.; and Weinberger, K., 2017. Densely connected convolutional networks. *CVPR*, (2017). (cited on page 61)

Huang, Z. and Wang, N., 2018. Data-driven sparse structure selection for deep neural networks. *ECCV*, (2018). (cited on page 91)

Hwang, J. J.; Yu, S.; Shi, J.; Collins, M. D.; Yang, T. J.; Zhan, X.; and Chen, L. C., 2019. SegSort: Segmentation by discriminative sorting of segments. *ICCV*, (2019). (cited on pages xxii and 73)

Isola, P.; Zoran, D.; Krishnan, D.; and Adelson, E., 2014. Crisp boundary detection using pointwise mutual information. *ECCV*, (2014). (cited on page 60)

Jampani, V.; Sun, D.; Liu, M.; Yang, M.; and Kautz, J., 2018. Superpixel sampling networks. *ECCV*, (2018). (cited on pages 59, 60, and 61)

Ji, S.; Xu, W.; Yang, M.; and Yu, K., 2013. 3D convolutional neural networks for human action recognition. *TPAMI*, (2013). (cited on page 89)

Jordan, M., 1998. Learning in graphical models. *MIT Press*, (1998). (cited on pages 1, 11, 19, 29, and 61)

Kao, P.; Ngo, T.; Zhang, A.; Chen, J.; and Manjunath, B., 2018. Brain tumor segmentation and tractographic feature extraction from structural MR images for overall survival prediction. *Workshop on MICCAI*, (2018). (cited on pages 103 and 105)

Kappes, J.; Andres, B.; Hamprecht, F.; Schnörr, C.; Nowozin, S.; Batra, D.; Kim, S.; Kroeger, T.; Kausler, B.; Lellmann, J.; Savchynskyy, B.; Komodakis, N.; and Rother, C., 2013. A comparative study of modern inference techniques for discrete energy minimization problems. *CVPR*, (2013). (cited on pages 29 and 45)

Karpathy, A.; Toderici, G.; Shetty, S.; Leung, T.; Sukthankar, R.; and Fei-Fei, L., 2014. Large-scale video classification with convolutional neural networks. *CVPR*, (2014). (cited on page 89)

Khaire, P. and Thakur, D., 2012. A fuzzy set approach for edge detection. *International Journal of Image Processing*, 6, 6 (2012), 403–412. (cited on pages 60, 71, 72, and 73)

Kingma, D. and Ba, J., 2015. Adam: A method for stochastic optimization. *ICLR*, (2015). (cited on page 104)

Kleesiek, J.; Urban, G.; Hubert, A.; Schwarz, D.; Hein, K.; Bendszus, M.; and Biller, A., 2016. Deep MRI brain extraction: A 3D convolutional neural network for skull stripping. *NeuroImage*, (2016). (cited on page 89)

Knobelreiter, P.; Reinbacher, C.; Shekhovtsov, A.; and Pock, T., 2017. End-to-end training of hybrid CNN-CRF models for stereo. *CVPR*, (2017). (cited on page 29)

Koller, D. and Friedman, N., 2009. Probabilistic graphical models: Principles andtechniques. *MIT Press*, (2009). (cited on page 1)

Kolmogorov, V., 2006. Convergent tree-reweighted message passing for energy minimization. *TPAMI*, (2006). (cited on pages 11, 19, 20, 28, 29, and 33)

Kopuklu, O.; Kose, N.; Gunduz, A.; and Rigoll, G., 2019. Resource efficient 3D convolutional neural networks. *ICCVW*, (2019). (cited on pages 103 and 104)

Krähenbühl, P. and Koltun, V., 2011. Efficient inference in fully connected CRFs with gaussian edge potentials. *NeurIPS*, (2011). (cited on pages 27, 45, 48, 51, 59, and 61)

Kwon, D.; Lee, K.; Yun, I.; and Lee, S., 2010. Solving MRFs with higher-order smoothness priors using hierarchical gradient nodes. *ACCV*, (2010). (cited on page 29)

LeCun, Y.; Bengio, Y.; and Hinton, G., 2015. Deep learning. *Nature*, (2015). (cited on page 1)

Lee, N.; Ajanthan, T.; Gould, S.; and Torr, P., 2020. A signal propagation perspective for pruning neural networks at initialization. *ICLR*, (2020). (cited on pages 90, 91, 93, 96, 104, 114, and 115)

Lee, N.; Ajanthan, T.; and Torr, P., 2019. SNIP: Single-shot network pruning based on connection sensitivity. *ICLR*, (2019). (cited on pages 7, 90, 91, 92, 93, and 109)

Li, C.; Wang, Z.; Wang, X.; and Qi, H., 2019a. Single-shot channel pruning based on alternating direction method of multipliers. *arXiv:1902.06382*, (2019). (cited on pages 91 and 92)

Li, H.; Kadav, A.; Samet, I. D. H.; and Graf, H., 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, (2016). (cited on pages 91 and 92)

Li, X.; Zhong, Z.; Wu, J.; Yang, Y.; Lin, Z.; and Liu, H., 2019b. Expectation-maximization attention networks for semantic segmentation. *ICCV*, (2019). (cited on page 61)

Li, Z. and Chen, J., 2015. Superpixel segmentation using linear spectral clustering. *CVPR*, (2015). (cited on pages 59, 60, and 61)

Lin, G.; Milan, A.; Shen, C.; and Reid, I., 2017. RefineNet: Multi-path refinement networks for high resolution semantic segmentation. *CVPR*, (2017). (cited on page 61)

Lin, G.; Shen, C.; Hengel, A.; and Reid, I., 2016. Efficient piecewise training of deep structured models for semantic segmentation. *CVPR*, (2016). (cited on page 29)

Lin, T.; Maire, M.; Belongie, S.; Bourdev, L.; Girshick, R.; Hays, J.; Perona, P.; Ramanan, D.; Zitnick, C.; and Dollar, P., 2014. Microsoft COCO: Common objects in context. *ECCV*, (2014). (cited on pages 59 and 70)

Liu, Y.; Yu, C.; Yu, M.; and He, Y., 2015a. Manifold SLIC: A fast method to compute content-sensitive superpixels. *CVPR*, (2015). (cited on page 61)

Liu, Z.; Li, X.; Luo, P.; Loy, C. C.; and Tang, X., 2015b. Semantic image segmentation via deep parsing network. *ICCV*, (2015). (cited on page 29)

Long, J.; Shelhamer, E.; and Darrell, T., 2015. Fully convolutional networks for semantic segmentation. *CVPR*, (2015). (cited on pages 59, 61, 70, and 72)

Mayer, N.; Ilg, E.; Hausser, P.; Fischer, P.; Cremers, D.; Dosovitskiy, A.; and Brox, T., 2016. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. *CVPR*, (2016). (cited on pages 91 and 103)

Menze, B.; Jakab, A.; and *et al*, S. B., 2015a. The multimodal brain tumor image segmentation benchmark (brats). *IEEE Transactions on Medical Imaging*, (2015). (cited on pages 89, 91, and 102)

Menze, M.; Heipke, C.; and Geiger, A., 2015b. Joint 3D estimation of vehicles and scene flow. *ISPRS Workshop on Image Sequence Analysis*, (2015). (cited on page 46)

Menze, M.; Heipke, C.; and Geiger, A., 2018. Object scene flow. *ISPRS Journal of Photogrammetry and Remote Sensing*, (2018). (cited on page 46)

Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; and Kautz, J., 2017. Pruning convolutional neural networks for resource efficient inference. *ICLR*, (2017). (cited on pages 91 and 92)

Mottaghi, R.; Chen, X.; Liu, X.; Cho, N.; Lee, S.; Fidler, S.; Urtasun, R.; and Yuille, A., 2014. The role of context for object detection and semantic segmentation in the wild. *CVPR*, (2014). (cited on pages 59 and 69)

Murphy, K.; Weiss, Y.; and Jordan, M., 1999. Loopy belief propagation for approximate inference: an empirical study. *UAI*, (1999). (cited on page 29)

Park, H.; Jeong, J.; Yoo, Y.; and Kwak, N., 2017. Superpixel-based semantic segmentation rrained by statistical process control. *BMVC*, (2017). (cited on page 62)

Pearl, J., 1988. Probabilistic reasoning in intelligent systems. *Morgan Kaufmann*, (1988). (cited on page 29)

Qi, C.; Su, H.; Mo, K.; and Guibas, L., 2017. Pointnet: Deep learning on point sets for 3D classification and segmentation. *CVPR*, (2017). (cited on page 89)

Reddi, S.; Kale, S.; and Kumar, S., 2018. On the convergence of adam and beyond. *ICLR*, (2018). (cited on page 104)

Riegler, G.; Ulusoy, A.; and Geiger, A., 2017. OctNet: Learning deep 3D representations at high resolutions. *CVPR*, (2017). (cited on page 92)

Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; Berg, A.; and Fei-Fei, L., 2014. Image net large scale visual recognition challenge. *arXiv:1409.0575v3*, (2014). (cited on page 1)

Salscheider, N., 2019. Simultaneous object detectin and semantic segmentation. *International conference on pattern recognition applications and methods (ICPRAM)*, (2019). (cited on page 59)

Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; and Chen, L., 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. *CVPR*, (2018). (cited on pages 91, 103, and 115)

Saxe, A.; McClelland, J.; and Ganguli, S., 2014. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *ICLR*, (2014). (cited on page 104)

Scharstein, D. and Szeliski, R., 2002. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *IJCV*, (2002). (cited on page 45)

Scharstein, D. and Szeliski, R., 2003. High-accuracy stereo depth maps using structured light. *CVPR*, (2003). (cited on page 45)

Schops, T.; Schonberger, J.; Galliani, S.; Sattler, T.; Schindler, K.; Pollefeys, M.; and Geiger, A., 2017. A multi-view stereo benchmark with high-resolution images and multi-camera videos. *CVPR*, (2017). (cited on page 46)

Schroff, F.; Criminisi, A.; and Zisserman, A., 2008. Object class segmentation using random forests. *BMVC*, (2008). (cited on page 61)

Schuurmans, M.; Berman, M.; and Blaschko, M., 2018. Efficient semantic image segmentation with superpixel pooling. *arXiv preprint arXiv:1806.02705*, (2018). (cited on page 62)

Seki, A. and Pollefeys, M., 2017. SGM-nets: Semi-global matching with neural networks. *CVPR*, (2017). (cited on pages 1, 27, 29, 32, and 35)

Simonyan, K. and Zisserman, A., 2014. Two-stream convolutional networks for action recognition in videos. *NeurIPS*, (2014). (cited on page 89)

Simonyan, K. and Zisserman, A., 2015. Very deep convolutional networks for large-scale image recognition. *ICLR*, (2015). (cited on page 61)

Soomro, K.; Zamir, A.; and Shah, M., 2012. UCF101: A dataset of 101 human action classes from videos in the wild. *CRCV-Techinal Report*, (2012). (cited on pages 103 and 104)

Stutz, D.; Hermans, A.; and Leibe, B., 2018. Superpixels: An evaluation of the state-of-the-art. *Computer Vision and Image Understanding*, 166 (2018), 1–27. (cited on pages 60, 61, 71, and 73)

Sutskever, I.; Martens, J.; Dahl, G.; and Hinton, G., 2013. On the importance of initialization and momentum in deep learning. *ICML*, (2013). (cited on pages 70 and 104)

Szeliski, R.; Zabih, R.; Scharstein, D.; Veksler, O.; Kolmogorov, V.; Agarwala, A.; Tappen, M.; and Rother, C., 2008. A comparative study of energy minimization methods for Markov random fields with smoothness-based priors. *TPAMI*, (2008). (cited on pages 27, 29, 45, and 46)

Taskar, B.; Guestrin, C.; and Koller, D., 2003. Max-margin Markov networks. *MIT Press*, (2003). (cited on page 29)

Thoma, M., 2016. A survey of semantic segmentation. *arXiv:1602.06541*, (2016). (cited on page 61)

Tsochantaridis, I.; Joachims, T.; Hofmann, T.; and Altun, Y., 2005. Large margin methods for structured and interdependent output variables. *JMLR*, (2005). (cited on page 29)

Tu, W.; Liu, M.; Jampani, V.; Sun, D.; Chien, S.; Yang, M.; and Kautz, J., 2018. Learning superpixels with segmentation-aware affinity loss. *CVPR*, (2018). (cited on page 60)

Uziel, R.; Ronen, M.; and Freifeld, O., 2019. Bayesian adaptive superpixel segmentation. *ICCV*, (2019). (cited on page 60)

Veksler, O., 2012. Multi-label moves for MRFs with truncated convex priors. *IJCV*, (2012). (cited on page 29)

Wainwright, M.; Jaakkola, T.; and Willsk, A., 2005a. Map estimation via agreement on (hyper)trees: Message-passing and linear-programming approaches. *IEEE Transactions on Information Theory*, 51, 11 (2005), 3697—-3717. (cited on page 19)

Wainwright, M.; Jaakkola, T.; and Willsky, A., 2005b. MAP estimation via agreement on (hyper) trees: Message-passing and linear-programming approaches. *Transactions on Information Theory*, (2005). (cited on page 33)

Wainwright, M. J. and Jordan, M. I., 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, (2008). (cited on page 29)

Wang, C.; Zhang, G.; and Grosse, R., 2020. Picking winning tickets before training by preserving gradient flow. *ICLR*, (2020). (cited on pages 91 and 92)

Wang, P.; Zeng, G.; Gan, R.; Wang, J.; and Zha, H., 2013. Structure-sensitive superpixels via geodesic distance. *IJCV*, (2013). (cited on page 61)

Wang, X.; Wang, T.; and Bu, J., 2011. Color image segmentation using pixel wise support vector machine classification. *pattern recognition*, (2011). (cited on page 61)

Wang, Z.; Zhang, Z.; and Geng, N., 2014. A message passing algorithm for MRF inference with unknown graphs and its applications. *ACCV*, (2014). (cited on page 29)

Wen, W.; Wu, C.; Wang, Y.; Chen, Y.; and Li, H., 2016. Learning structured sparsity in deep neural networks. *NeurIPS*, (2016). (cited on page 92)

Xing, F.; Cambria, E.; Huang, W.; and Xu, Y., 2016. Weakly supervised semantic segmentation with superpixel embedding. *ICIP*, (2016). (cited on page 62)

Xu, B.; Wang, N.; Chen, T.; and Li, M., 2015. Empirical evaluation of rectified activations in convolutional network. *arXiv:1505.00853*, (2015). (cited on page 67)

Xu, Z.; Ajanthan, T.; and Hartley, R., 2020a. Fast and differentiable message passing on pairwise markov random fields. *ACCV*, (2020). (cited on page 7)

Xu, Z.; Ajanthan, T.; and Hartley, R., 2020b. Refining semantic segmentation with superpixel by transparent initialization and sparse encoder. *arXiv:2010.04363*, (2020). (cited on page 8)

Xu, Z.; Ajanthan, T.; Vineet, V.; and Hartley, R., 2020c. Ranp: resource aware neuron pruning at initialization for 3d cnns. *3DV*, (2020). (cited on page 8)

Yang, F.; Sun, Q.; Jin, H.; and Zhou, Z., 2020. Superpixel segmentation with fully convolutional networks. *CVPR*, (2020). (cited on pages xvii, xviii, 60, 61, 62, 63, 64, 70, and 86)

Yi, L.; Kim, V.; Ceylan, D.; Shen, I.; Yan, M.; Su, H.; Lu, C.; Huang, Q.; Sheffer, A.; and Guibas, L., 2016. A scalable active framework for region annotation in 3D shape collections. *SIGGRAPH Asia*, (2016). (cited on pages 89, 91, and 102)

Yi, L.; Shao, L.; and Savva, M., 2017. Large-scale 3D shape reconstruction and segmentation from shapenet core55. *arXiv preprint arXiv:1710.06104*, (2017). (cited on page 105)

Yu, J. and Huang, T., 2019. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv:1903.11728*, (2019). (cited on pages 91 and 92)

Yu, N.; Qiu, S.; Hu, X.; and Li, J., 2017. Accelerating convolutional neural networks by group-wise 2D-filter pruning. *IJCNN*, (2017). (cited on pages 91 and 92)

Yu, R.; Li, A.; Chen, C.; Lai, J.; Morariu, V.; Han, X.; Gao, M.; Lin, C.; and Davis, L., 2018. NISP: Pruning networks using neuron importance score propagation. *CVPR*, (2018). (cited on page 91)

Zanjani, F.; Moin, D.; Verheij, B.; Claessen, F.; Cherici, T.; Tan, T.; and With, P., 2019. Deep learning approach to semantic segmentation in 3D point cloud intra-oral scans of teeth. *Proceedings of Machine Learning Research*, (2019). (cited on page 89)

Zhang, C.; Luo, W.; and Urtasun, R., 2018a. Efficient convolutions for real-time semantic segmentation of 3D point cloud. *3DV*, (2018). (cited on page 89)

Zhang, F.; Prisacariu, V.; Yang, R.; and Torr, P. H., 2019a. GA-Net: Guided aggregation net for end-to-end stereo matching. *CVPR*, (2019). (cited on page 29)

Zhang, H.; Dana, K.; Ping, J.; Wang, Z. Z. X.; Tyagi, A.; and Agrawal, A., 2018b. Context encoding for semantic segmentation. *CVPR*, (2018). (cited on pages 59, 72, and 73)

Zhang, H.; Jiang, K.; Zhang, Y.; Li, Q.; Xia, C.; and Chen, X., 2014. Discriminative feature learning for video semantic segmentation. *International Conference on Virtual Reality and Visualization*, (2014). (cited on page 89)

Zhang, H.; Wu, C.; Zhang, Z.; Zhu, Y.; Zhang, Z.; Lin, H.; Sun, Y.; He, T.; Mueller, J.; Manmatha, R.; Li, M.; and Smola, A., 2020. ResNeSt: Split-attention networks. *arXiv:2004.08955*, (2020). (cited on pages 59, 61, 62, 63, 72, and 73)

Zhang, M. and Stadie, B., 2019. One-shot pruning of recurrent neural networks by jacobian spectrum evaluation. *arXiv:1912.00120*, (2019). (cited on pages 91 and 92)

Zhang, S.; Tong, H.; Xu, J.; and Maciejewski, R., 2019b. Graph convolutional networks: a comprehensive review. *computational social networks*, (2019). (cited on pages 23 and 29)

Zhang, Y.; Wang, H.; Luo, Y.; Yu, L.; Hu, H.; Shan, H.; and Quek, T., 2019c. Three dimensional convolutional neural network pruning with regularization-based method. *ICIP*, (2019). (cited on pages 91 and 92)

Zhao, H.; Shi, J.; Qi, X.; Wang, X.; and Jia, J., 2017. Pyramid scene parsing network. *CVPR*, (2017). (cited on pages 59 and 72)

Zhao, H.; Zhang, Y.; Liu, S.; Shi, J.; Loy, C.; Lin, D.; and Jia, J., 2018. PSANet: Point-wise spatial attention network for scene parsing. *ECCV*, (2018). (cited on page 61)

Zhao, S.; Wang, Y.; Yang, Z.; and Cai, D., 2019. Region mutual information loss for semantic segmentation. *NeurIPS*, (2019). (cited on page 61)

Zheng, S.; Jayasumana, S.; Romera-Paredes, B.; Vineet, V.; Su, Z.; Du, D.; Huang, C.; and Torr, P. H., 2015. Conditional random fields as recurrent neural networks. *CVPR*, (2015). (cited on pages 1, 6, 27, 29, 35, and 59)

Zhou, B.; Zhao, H.; Puig, X.; Fidler, S.; Barriuso, A.; and Torralba, A., 2017. Scene parsing through ade20k dataset. *CVPR*, (2017). (cited on page 69)

Zhou, B.; Zhao, H.; Puig, X.; Xiao, T.; Fidler, S.; Barriuso, A.; and Torralba, A., 2016. Semantic understanding of scenes through ade20k dataset. *IJCV*, (2016). (cited on page 69)