國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

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

Master Thesis

牛頓法於卷積神經網路之應用

Newton Methods For Convolutional Neural Networks

陳勁龍

Tan Kent Loong

指導教授：林智仁 博士

Advisor: Chih-Jen Lin, Ph.D.

中華民國 107 年 7 月

July, 2018

doi:10.6342/NTU201802108

國立臺灣大學碩士學位論文
口試委員會審定書

牛頓法於卷積神經網路之應用

Newton Methods for Convolutional Neural Networks

本論文係陳勁龍君（學號 R04944005）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國一百零七年七月十七日承下列考試委員審查通過及口試及格，特此證明

口試委員：

林智仁（簽名）

（指導教授）

李育杰　林軒田

所　長：楊佳玲

doi:10.6342/NTU201802108

中文摘要

深度學習包含困難的非凸優化問題。大多數研究經常使用隨機梯度演算法（SG）來優化這類模型。使用SG通常很有效，但有時並不那麼強大。近代的研究探討了利用牛頓法作爲替代的優化方法，但絕大部分研究只將其應用於全連接神經網路。他們沒有探討諸如卷積神經網路等更爲廣泛使用的深度學習模型。其中一個原因是應用牛頓法於卷積神經網路的過程中牽涉到多個複雜的運算，因此目前未有仔細的相關研究。在這篇論文中，我們給出詳細的建構模組，當中包括函數、梯度及賈可比矩陣的運算和高斯-牛頓矩陣向量的乘積。這些基本的模組非常重要。因爲沒有它們，任何牛頓法於全連接神經網路的進步沒辦法在卷積神經網路上嘗試。因此我們的研究將可能推動更多牛頓法於卷積神經網路上的發展。我們完成一個簡單的MATLAB實作。實驗結果顯示這個方法具有競爭力。

關鍵詞: 卷積神經網路, 多類別分類, 大規模學習, 抽樣海森矩陣。

doi:10.6342/NTU201802108

ABSTRACT

Deep learning involves a difficult non-convex optimization problem, which is often solved by stochastic gradient (SG) methods. While SG is usually effective, it is sometimes not very robust. Recently, Newton methods have been investigated as an alternative optimization technique, but nearly all existing studies consider only fully-connected feedforward neural networks. They do not investigate other types of networks such as Convolutional Neural Networks (CNN), which are more commonly used in deep-learning applications. One reason is that Newton methods for CNN involve complicated operations, and so far no works have conducted a thorough investigation. In this thesis, we give details of building blocks including function, gradient, and Jacobian evaluation, and Gauss-Newton matrix-vector products. These basic components are very important because without them none of any recent improvement of Newton methods for fully-connected networks can even be tried. Thus we will enable possible further developments of Newton methods for CNN. We finish a simple *MATLAB* implementation and show that it gives competitive test accuracy.

KEYWORDS: Convolutional neural networks, multi-class classification, large-scale classification, subsampled Hessian.

doi:10.6342/NTU201802108

TABLE OF CONTENTS

口試委員會審定書 i

中文摘要 ii

ABSTRACT iii

LIST OF FIGURES vi

LIST OF TABLES vii

CHAPTER

I. Introduction 1

II. Optimization Problem of Feedforward CNN 3

2.1 Convolutional Layer 3
2.1.1 Zero Padding 9
2.1.2 Pooling Layer 9
2.1.3 Summary of a Convolutional Layer 11
2.2 Fully-Connected Layer 12
2.3 The Overall Optimization Problem 13

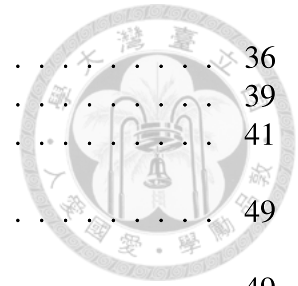
III. Hessian-free Newton Methods for Training CNN 14

3.1 Procedure of the Newton Method 14
3.2 Gradient Evaluation 17
3.2.1 Padding, Pooling, and the Overall Procedure 21
3.3 Jacobian Evaluation 22
3.4 Gauss-Newton Matrix-Vector Products 24

IV. Implementation Details 27

4.1 Generation of $\phi(Z^{m-1,i})$ 27
4.2 Construction of $P_{\text{pool}}^{m-1,i}$ 32
4.3 Details of Padding Operation 34

doi:10.6342/NTU201802108



4.4 Evaluation of $\boldsymbol{v}^T P_{\phi}^{m-1}$ 36
4.5 Gauss-Newton Matrix-Vector Products 39
4.6 Mini-Batch Function and Gradient Evaluation 41

V. Analysis of Newton Methods for CNN 49

5.1 Memory Requirement 49
5.2 Computational Cost 51

VI. Experiments 54

6.1 Comparison Between Newton and Stochastic Gradient Methods 56

VII. Conclusions 59

APPENDICES **60**

BIBLIOGRAPHY **66**

doi:10.6342/NTU201802108

LIST OF FIGURES

Figure

2.1 An padding example with $s^m = 1$ in order to set $a^m = a^{m-1}$. 10
2.2 An illustration of max pooling to extract translational invariance features. The image B is derived from shifting A by 1 pixel in the horizontal direction. 10

doi:10.6342/NTU201802108

LIST OF TABLES

Table

6.1 Summary of the data sets, where $a^0 \times b^0 \times d^0$ represents the (height, width, channel) of the input image, l is the number of training data, l_t is the number of test data, and n_L is the number of classes. 55

6.2 Structure of convolutional neural networks. "conv" indicates a convolutional layer, "pool" indicates a pooling layer, and "full" indicates a fully-connected layer. 57

6.3 Test accuracy for Newton method and SG. For Newton method, we trained for 250 iterations; for SG, we trained for 1000 epochs. 58

doi:10.6342/NTU201802108

CHAPTER I

Introduction

Deep learning is now widely used in many applications. To apply this technique, a difficult non-convex optimization problem must be solved. Currently, stochastic gradient (SG) methods and their variants are the major optimization technique used for deep learning (e.g., Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). This situation is different from some application domains, where other types of optimization methods are more frequently used. One interesting research question is thus to study if other optimization methods can be extended to be viable alternatives for deep learning. In this thesis, we aim to address this issue by developing a practical Newton method for deep learning.

Some past works have studied Newton methods for training deep neural networks (e.g., Botev et al. 2017; He et al. 2016; Kiros 2013; Martens 2010; Vinyals and Povey 2012; Wang et al. 2015, 2018a). Almost all of them consider fully-connected feedforward neural networks and some have shown the potential of Newton methods for being more robust than SG. Unfortunately, these works have not fully established Newton methods as a practical technique for deep learning because other types of networks such as Convolutional Neural Networks (CNN) are more commonly used in deep-learning applications. One important reason why CNN was not considered is because of the very complicated operations in implementing Newton methods. Up to now no works

doi:10.6342/NTU201802108

have shown details of all the building blocks such as function, gradient, and Jacobian evaluation, and Hessian-vector products. In particular, because interpreted-type languages such as *Python* or *MATLAB* have been popular for deep learning, how to easily implement efficient operations by these languages is an important research issue.

In this thesis, we aim at a thorough investigation on the implementation of Newton methods for CNN. We focus on basic components because without them none of any recent improvement of Newton methods for fully-connected networks can be even tried. We will enable many further developments of Newton methods for CNN and maybe even other types of networks.

This thesis is organized as follows. In Chapter II, we begin with introducing CNN. In Chapter III, Newton methods for CNN are introduced and the detailed mathematical formulations of all operations are dervied. In Chapter IV, we provide details for an efficient *MATLAB* implementation. Experiments to demonstrate the viability of Newton methods for CNN are in Chapter VI. In the same chapter, we also investigate the efficiency of our implementation. Chapter VII concludes this work.

A *MATLAB* package of implementing a Newton method for CNN is available at

`https://https://www.csie.ntu.edu.tw/~cjlin/papers/cnn/`

Programs used for experiments can be found at the same page.

This thesis is based on the paper by Wang et al. (2018b). We acknowledge the support from Ministry of Science and Technology of Taiwan.

doi:10.6342/NTU201802108

CHAPTER II

Optimization Problem of Feedforward CNN

Consider a K-class problem, where the training data set consists of l input pairs $(Z^{0,i}, \boldsymbol{y}^i),\ i = 1, \ldots, l$. Here $Z^{0,i}$ is the ith input image with dimension $a^0 \times b^0 \times d^0$, where a^0 denotes the height of the input images, b^0 represents the width of the input images and d^0 is the number of color channels. If $Z^{0,i}$ belongs to the kth class, then the label vector

$$\boldsymbol{y}^i = [\underbrace{0, \ldots, 0}_{k-1}, 1, 0, \ldots, 0]^T \in R^K.$$

A CNN utilizes a stack of convolutional and pooling layers followed by fully-connected layers to predict the target vector. Let L be the number of layers, L^c be the number of convolutional layers, and L^f be the number of fully-connected layers. The images

$$Z^{0,i},\ i = 1, \ldots, l,$$

are the input of layer 1. Subsequently we describe operations of convolutional layers, pooling layers, and fully-connected layers.

2.1 Convolutional Layer

A hallmark of CNN is that both input and output of a convolutional layer are explicitly assumed to be images. We discuss the details between layers $m-1$ and m. Let the

doi:10.6342/NTU201802108

input be an image of dimensions

$$a^{m-1} \times b^{m-1} \times d^{m-1},$$

where a^{m-1} is the height, b^{m-1} is the width, and d^{m-1} is the depth (or the number of channels). Thus for every given channel, we have a matrix of $a^{m-1} \times b^{m-1}$ pixels. Specifically, the input contains the following matrices.

$$\begin{bmatrix} z^{m-1,i}_{1,1,1} & & \\ & \ddots & \\ & & z^{m-1,i}_{a^{m-1},b^{m-1},1} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} z^{m-1,i}_{1,1,d^{m-1}} & & \\ & \ddots & \\ & & z^{m-1,i}_{a^{m-1},b^{m-1},d^{m-1}} \end{bmatrix} \tag{2.1}$$

For example, at layer 1, usually $d^0 = 3$ because of three color channels (red, green, blue). For each channel, the matrix of size $a^0 \times b^0$ contains raw pixel values of the image.

To generate the output, we consider d^m filters, each of which is a 3-D weight matrix of size

$$h^m \times h^m \times d^{m-1}.$$

Specifically, the jth filter includes the following matrices

$$\begin{bmatrix} w^{m,j}_{1,1,1} & & w^{m,j}_{1,h^m,1} \\ & \ddots & \\ w^{m,j}_{h^m,1,1} & & w^{m,j}_{h^m,h^m,1} \end{bmatrix}, \quad \cdots \quad, \begin{bmatrix} w^{m,j}_{1,1,d^{m-1}} & & w^{m,j}_{1,h^m,d^{m-1}} \\ & \ddots & \\ w^{m,j}_{h^m,1,d^{m-1}} & & w^{m,j}_{h^m,h^m,d^{m-1}} \end{bmatrix}$$

and a bias term b^m_j. The main idea of CNN is to conduct convolutional operations, each of which is the inner product between a small sub-image and a filter. We now describe the details. Specifically, for the jth filter, we scan the entire input image to obtain small regions of size (h^m, h^m) and calculate the inner product between each region and the filter. For example, if we start from the upper left corner of the input image, the first

doi:10.6342/NTU201802108

sub-image of channel d is

$$\begin{bmatrix} z^{m-1,i}_{1,1,d} & \dots & z^{m-1,i}_{1,h^m,d} \\ & \ddots & \\ z^{m-1,i}_{h^m,1,d} & \dots & z^{m-1,i}_{h^m,h^m,d} \end{bmatrix}.$$

We then calculate the following value.

$$\sum_{d=1}^{d^{m-1}} \left\langle \begin{bmatrix} z^{m-1,i}_{1,1,d} & \dots & z^{m-1,i}_{1,h^m,d} \\ & \ddots & \\ z^{m-1,i}_{h^m,1,d} & \dots & z^{m-1,i}_{h^m,h^m,d} \end{bmatrix}, \begin{bmatrix} w^{m,j}_{1,1,d} & \dots & w^{m,j}_{1,h^m,d} \\ & \ddots & \\ w^{m,j}_{h^m,1,d} & \dots & w^{m,j}_{h^m,h^m,d} \end{bmatrix} \right\rangle + b^m_j, \tag{2.2}$$

where $\langle \cdot, \cdot \rangle$ means the sum of component-wise products between two matrices. This value becomes the $(1, 1)$ position of the channel j of the output image.

Next, we must obtain other sub-images to produce values in other positions of the output image. We specify the stride s^m for sliding the filter. That is, we move s^m pixels vertically or horizontally to get sub-images. For the $(2, 1)$ position of the output image, we move down s^m pixels vertically to obtain the following sub-image:

$$\begin{bmatrix} z^{m-1,i}_{1+s^m,1,d} & \dots & z^{m-1,i}_{1+s^m,h^m,d} \\ & \ddots & \\ z^{m-1,i}_{h^m+s^m,1,d} & \dots & z^{m-1,i}_{h^m+s^m,h^m,d} \end{bmatrix}, d = 1, \dots, d^{m-1}.$$

Then the $(2, 1)$ position of the channel j of the output image is

$$\sum_{d=1}^{d^{m-1}} \left\langle \begin{bmatrix} z^{m-1,i}_{1+s^m,1,d} & \dots & z^{m-1,i}_{1+s^m,h^m,d} \\ & \ddots & \\ z^{m-1,i}_{h^m+s^m,1,d} & \dots & z^{m-1,i}_{h^m+s^m,h^m,d} \end{bmatrix}, \begin{bmatrix} w^{m,j}_{1,1,d} & \dots & w^{m,j}_{1,h^m,d} \\ & \ddots & \\ w^{m,j}_{h^m,1,d} & \dots & w^{m,j}_{h^m,h^m,d} \end{bmatrix} \right\rangle + b^m_j. \tag{2.3}$$

Assume that vertically and horizontally we can move the filter a^m and b^m times, respectively. Therefore,

$$a^{m-1} = h^m + (a^m - 1) \times s^m,$$

doi:10.6342/NTU201802108

$$b^{m-1} = h^m + (b^m - 1) \times s^m. \tag{2.4}$$

By our notation, the output image has the size

$$a^m \times b^m \times d^m.$$

For efficient implementations, we can conduct all operations including (2.2) and (2.3) by matrix operations. To begin, we concatenate the matrices of the different channels in (2.1) to

$$Z^{m-1,i} = \begin{bmatrix} z^{m-1,i}_{1,1,1} & \dots & z^{m-1,i}_{a^{m-1},1,1} & z^{m-1,i}_{1,2,1} & \dots & z^{m-1,i}_{a^{m-1},b^{m-1},1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ z^{m-1,i}_{1,1,d^{m-1}} & \dots & z^{m-1,i}_{a^{m-1},1,d^{m-1}} & z^{m-1,i}_{1,2,d^{m-1}} & \dots & z^{m-1,i}_{a^{m-1},b^{m-1},d^{m-1}} \end{bmatrix}, \; i = 1, \dots, l. \tag{2.5}$$

We note that (2.2) is the inner product between the following two vectors

$$\begin{bmatrix} w^{m,j}_{1,1,1} & \dots & w^{m,j}_{h^m,1,1} & w^{m,j}_{1,2,1} & \dots & w^{m,j}_{h^m,h^m,1} & \dots & w^{m,j}_{h^m,h^m,d^{m-1}} & b^m_j \end{bmatrix}^T$$

and

$$\begin{bmatrix} z^{m-1,i}_{1,1,1} & \dots & z^{m-1,i}_{h^m,1,1} & z^{m-1,i}_{1,2,1} & \dots & z^{m-1,i}_{h^m,h^m,1} & \dots & z^{m-1,i}_{h^m,h^m,d^{m-1}} & 1 \end{bmatrix}^T.$$

Therefore, based on Vedaldi and Lenc (2015), we define the following two operators

$$\text{vec}(M) = \begin{bmatrix} (M_{:,1})^T \dots (M_{:,n})^T \end{bmatrix}^T \in R^{mn \times 1}, \text{ where } M \in R^{m \times n}, \tag{2.6}$$

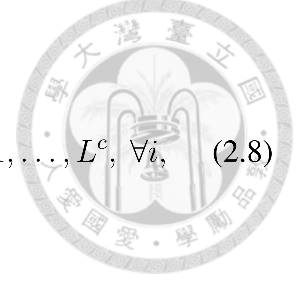
$$\text{mat}(\boldsymbol{v})_{m \times n} = \begin{bmatrix} v_1 & & v_{(n-1)m+1} \\ \vdots & \dots & \vdots \\ v_m & & v_{nm} \end{bmatrix} \in R^{m \times n}, \text{ where } \boldsymbol{v} \in R^{mn \times 1}, \tag{2.7}$$

and the operator

$$\phi : R^{d^{m-1} \times a^{m-1} b^{m-1}} \to R^{h^m h^m d^{m-1} \times a^m b^m}$$

doi:10.6342/NTU201802108

in order to collect all sub-images in $Z^{m-1,i}$:



$$\phi(Z^{m-1,i}) \equiv \text{mat}\left(P_{\phi}^{m-1}\text{vec}(Z^{m-1,i})\right)_{h^m h^m d^{m-1} \times a^m b^m},\ m = 1,\ldots,L^c,\ \forall i, \tag{2.8}$$

where

$$P_{\phi}^{m-1} \in R^{h^m h^m d^{m-1} a^m b^m \times d^{m-1} a^{m-1} b^{m-1}}.$$

We discuss the implementation details of (2.8) in Chapter 4.1. Then, we have $\phi(Z^{m-1,i})$ derived as follows.

$$\begin{bmatrix}
z_{1,1,1}^{m-1,i} & \dots & z_{1+(a^m-1)\times s^m,1,1}^{m-1,i} & z_{1,1+s^m,1}^{m-1,i} & \dots & z_{1+(a^m-1)\times s^m,1+(b^m-1)\times s^m,1}^{m-1,i} \\
z_{2,1,1}^{m-1,i} & \dots & z_{2+(a^m-1)\times s^m,1,1}^{m-1,i} & z_{2,1+s^m,1}^{m-1,i} & \dots & z_{2+(a^m-1)\times s^m,1+(b^m-1)\times s^m,1}^{m-1,i} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
z_{h^m,h^m,1}^{m-1,i} & \dots & z_{h^m+(a^m-1)\times s^m,h^m,1}^{m-1,i} & z_{h^m,h^m+s^m,1}^{m-1,i} & \dots & z_{h^m+(a^m-1)\times s^m,h^m+(b^m-1)\times s^m,1}^{m-1,i} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
z_{1,1,d^{m-1}}^{m-1,i} & \dots & z_{1+(a^m-1)\times s^m,1,d^{m-1}}^{m-1,i} & z_{1,1+s^m,d^{m-1}}^{m-1,i} & \dots & z_{1+(a^m-1)\times s^m,1+(b^m-1)\times s^m,d^{m-1}}^{m-1,i} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
z_{h^m,h^m,d^{m-1}}^{m-1,i} & \dots & z_{h^m+(a^m-1)\times s^m,h^m,d^{m-1}}^{m-1,i} & z_{h^m,h^m+s^m,d^{m-1}}^{m-1,i} & \dots & z_{h^m+(a^m-1)\times s^m,h^m+(b^m-1)\times s^m,d^{m-1}}^{m-1,i}
\end{bmatrix}. \tag{2.9}$$

By considering

$$W^m = \begin{bmatrix} w_{1,1,1}^{m,1} & w_{2,1,1}^{m,1} & \dots & w_{h^m,h^m,d^{m-1}}^{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,1,1}^{m,d^m} & w_{2,1,1}^{m,d^m} & \dots & w_{h^m,h^m,d^{m-1}}^{m,d^m} \end{bmatrix} \in R^{d^m \times h^m h^m d^{m-1}} \text{ and } \boldsymbol{b}^m = \begin{bmatrix} b_1^m \\ \vdots \\ b_{d^m}^m \end{bmatrix} \in R^{d^m \times 1}, \tag{2.10}$$

the following operations are conducted.

$$S^{m,i} = W^m \phi(Z^{m-1,i}) + \boldsymbol{b}^m \mathbb{1}_{a^m b^m}^T \in R^{d^m \times a^m b^m}, \tag{2.11}$$

doi:10.6342/NTU201802108

where

$$S^{m,i} = \begin{bmatrix} s^{m,i}_{1,1,1} & \dots & s^{m,i}_{a^m,1,1} & s^{m,i}_{1,2,1} & \dots & s^{m,i}_{a^m,b^m,1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ s^{m,i}_{1,1,d^m} & \dots & s^{m,i}_{a^m,1,d^m} & s^{m,i}_{1,2,d^m} & \dots & s^{m,i}_{a^m,b^m,d^m} \end{bmatrix} \text{ and } \mathbb{1}_{a^m b^m} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \in R^{a^m b^m \times 1}.$$

Next, an activation function is applied to scale the value.

$$z^{m,i}_{a,b,d} = \sigma(s^{m,i}_{a,b,d}), \tag{2.12}$$

where $a = 1, \dots, a^m$, $b = 1, \dots, b^m$, and $d = 1, \dots, d^m$. For CNN, commonly the following RELU activation function

$$\sigma(x) = \max(x, 0) \tag{2.13}$$

is used and we consider it in this work. The output becomes the following matrix

$$Z^{m,i} = \begin{bmatrix} z^{m,i}_{1,1,1} & z^{m,i}_{2,1,1} & \dots & z^{m,i}_{a^m,b^m,1} \\ & & \ddots & \\ z^{m,i}_{1,1,d^m} & z^{m,i}_{2,1,d^m} & \dots & z^{m,i}_{a^m,b^m,d^m} \end{bmatrix}. \tag{2.14}$$

We then apply (2.8) to expand the output to form the matrix $\phi(Z^{m,i})$ and then substitute $\phi(Z^{m,i})$ into (2.11), so we can continue the operations between layers m and $m+1$.

Note that by the matrix representation, the storage is increased from

$$a^{m-1} \times b^{m-1} \times d^{m-1}$$

in (2.1) to

$$(h^m h^m d^{m-1}) \times a^m \times b^m.$$

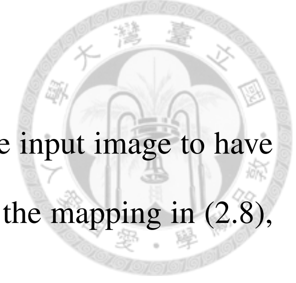
From (2.4), roughly

$$\left(\frac{h^m}{s^m}\right)^2$$

folds increase of the memory occurs. However, we gain efficiency by using fast matrix-matrix multiplications in optimized BLAS (Dongarra et al., 1990).

doi:10.6342/NTU201802108

2.1.1 Zero Padding



To make (2.4) hold or a^m be larger, sometimes we enlarge the input image to have zero values around the border. This technique, conducted before the mapping in (2.8), is called zero-padding in CNN training. For example, we may set

$$a^m = a^{m-1}$$

in order to prevent the decrease of the image size. When

$$s^m = 1,$$

we can pad the input image with

$$h^m - 1$$

lines of zeros around every border. See Figure 2.1.

For our derivation, we represent the padding operation as the following linear operation:

$$Z^{m,i} = \text{mat}(P_{\text{padding}}^{m-1,i}\text{vec}(Z^{m-1,i}))_{d^m \times a^m b^m}. \tag{2.15}$$

2.1.2 Pooling Layer

For CNN, to reduce the computational cost, a dimension reduction is often applied by using a pooling layer after each convolutional layer. Usually we consider an operation that can (approximately) extract rotational or translational invariance features. There are various types of pooling methods such as average pooling, max pooling, and stochastic pooling. We consider max pooling in this chapter because it is the most used setting for CNN. Here we show an example of max pooling by considering two 4×4 images, A and B, in Figure 2.2. The image B is derived by shifting A by 1 pixel in the horizontal direction. We split two images into four 2×2 sub-images and choose the

doi:10.6342/NTU201802108

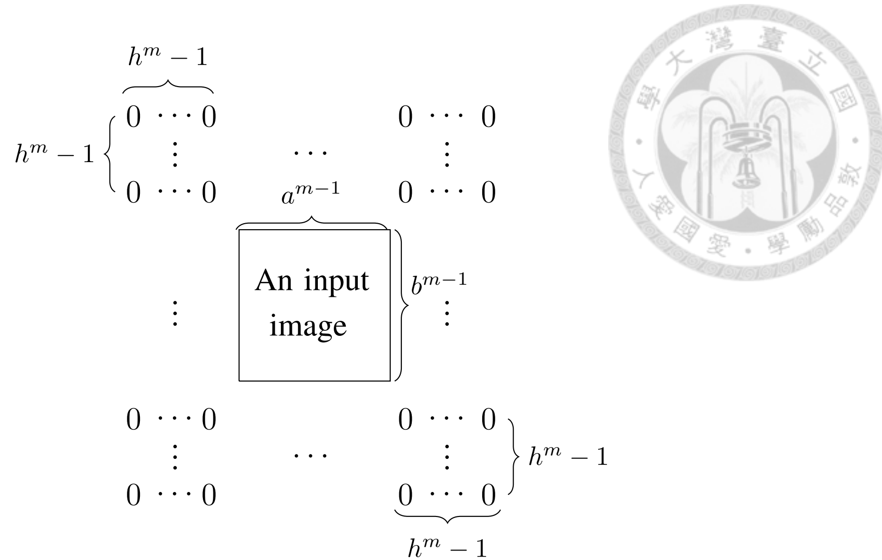


Figure 2.1: An padding example with $s^m = 1$ in order to set $a^m = a^{m-1}$.

(a) Image A

(b) Image B

Figure 2.2: An illustration of max pooling to extract translational invariance features. The image B is derived from shifting A by 1 pixel in the horizontal direction.

max value from every sub-image. In each sub-image because only some elements are changed, the maximal value is likely the same or similar. This is called translational invariance and for our example the two output images from A and B are the same.

Now we discuss the mathematical operation of the pooling layer. They are in fact special cases of convolutional operations. Assume $Z^{m-1,i}$ is the input image (i.e., output image of the previous convolutional layer). We partition every channel of $Z^{m-1,i}$ into non-overlapping sub-regions by $h^m \times h^m$ filters with the stride $s^m = h^m$.[1] By the same definition as (2.8) we can generate the matrix

$$\phi(Z^{m-1,i}) = \text{mat}(P_{\phi}^{m-1}\text{vec}(Z^{m-1,i}))_{h^m h^m \times d^{m-1} a^m b^m}, \tag{2.16}$$

[1] Because of the disjoint sub-regions, the stride s^m for sliding the filters is equal to h^m.

doi:10.6342/NTU201802108

where

$$a^m = \frac{a^{m-1}}{h^m},\; b^m = \frac{b^{m-1}}{h^m}. \tag{2.17}$$

To select the largest element of each sub-region, there exists a matrix

$$W^{m,i} \in R^{d^m a^m b^m \times h^m h^m d^{m-1} a^m b^m}$$

so that each row of $W^{m,i}$ selects a single element from $\text{vec}(\phi(Z^{m-1,i}))$. Therefore,

$$Z^{m,i} = \text{mat}\left(W^{m,i}\text{vec}(\phi(Z^{m-1,i}))\right)_{d^m \times a^m b^m}. \tag{2.18}$$

Note that different from (2.11) of the convolutional layer, $W^{m,i}$ is a constant matrix rather than a weight matrix. Further, because from (2.8)

$$\text{vec}(\phi(Z^{m-1,i})) = P_\phi^{m-1}\text{vec}(Z^{m-1,i}),$$

we have

$$Z^{m,i} = \text{mat}\left(P_{\text{pool}}^{m-1,i}\text{vec}(Z^{m-1,i})\right)_{d^m \times a^m b^m}, \tag{2.19}$$

where

$$P_{\text{pool}}^{m-1,i} = W^{m,i} P_\phi^{m-1} \in R^{d^m a^m b^m \times d^{m-1} a^{m-1} b^{m-1}}.$$

We provide implementation details in Chapter 4.2. Note that pooling operations are often considered as an (optional) part of the convolutional layer. Here we treat them as a separate layer for the easier description of the procedure.

2.1.3 Summary of a Convolutional Layer

From the input $Z^{m-1,i}$, we consider the following flow as one convolutional layer:

$$Z^{m-1,i} \rightarrow \text{padding} \rightarrow \phi(Z^{m-1,i}) \rightarrow \bar{S}^{m,i} \rightarrow \bar{Z}^{m,i} \rightarrow \text{pooling} \rightarrow Z^{m,i},$$

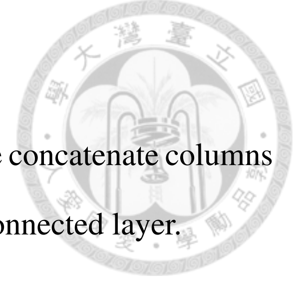
where

$$\bar{S}^{m,i} \quad \text{and} \quad \bar{Z}^{m,i}$$

are $S^{m,i}$ and $Z^{m,i}$ in (2.11) and (2.14), respectively, if pooling is not applied.

doi:10.6342/NTU201802108

2.2 Fully-Connected Layer



After passing through the convolutional and pooling layers, we concatenate columns in the matrix in (2.14) to form the input vector of the first fully-connected layer.

$$\boldsymbol{z}^{m,i} = \text{vec}(Z^{m,i}),\ i = 1, \dots, l,\ m = L^c.$$

In the fully-connected layers ($L^c < m \leq L$), we consider the following weight matrix and bias vector between layers $m-1$ and m.

$$W^m = \begin{bmatrix} w_{11}^m & w_{21}^m & \cdots & w_{n_{m-1}1}^m \\ w_{12}^m & w_{22}^m & \cdots & w_{n_{m-1}2}^m \\ \vdots & \vdots & \vdots & \vdots \\ w_{1n_m}^m & w_{2n_m}^m & \cdots & w_{n_{m-1}n_m}^m \end{bmatrix}_{n_m \times n_{m-1}} \text{ and } \boldsymbol{b}^m = \begin{bmatrix} b_1^m \\ b_2^m \\ \vdots \\ b_{n_m}^m \end{bmatrix}_{n_m \times 1}, \tag{2.20}$$

where n_{m-1} and n_m are the numbers of neurons in layers $m-1$ and m, respectively.[2] If $\boldsymbol{z}^{m-1,i} \in R^{n_{m-1}}$ is the input vector, the following operations are applied to generate the output vector $\boldsymbol{z}^{m,i} \in R^{n_m}$.

$$\boldsymbol{s}^{m,i} = W^m \boldsymbol{z}^{m-1,i} + \boldsymbol{b}^m, \tag{2.21}$$

$$z_j^{m,i} = \sigma(s_j^{m,i}),\ j = 1, \dots, n_m. \tag{2.22}$$

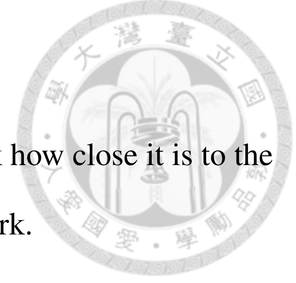
For the activation function in fully-connected layers, except the last layer, we also consider the RELU function defined in (2.13). For the last layer, we use the following linear function.

$$\sigma(x) = x. \tag{2.23}$$

[2] $n_{L^c} = d^{L^c} a^{L^c} b^{L^c}$ and $n_L = K$ is the number of classes.

doi:10.6342/NTU201802108

2.3 The Overall Optimization Problem



At the last layer, the output $\boldsymbol{z}^{L,i}, \forall i$ is obtained. We can check how close it is to the label vector $\boldsymbol{y}^i$ and consider the following squared loss in this work.

$$\xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i) = ||\boldsymbol{z}^{L,i} - \boldsymbol{y}^i||^2. \tag{2.24}$$

Furthermore, we can collect all model parameters such as filters of convolutional layers in (2.10) and weights/biases in (2.20) for fully-connected layers into a long vector $\boldsymbol{\theta} \in R^n$, where n becomes the total number of variables from the discussion in this chapter.

$$n = \sum_{m=1}^{L^c} d^m \times (h^m \times h^m \times d^{m-1} + 1) + \sum_{m=L^c+1}^{L} n_m \times (n_{m-1} + 1).$$

The output $\boldsymbol{z}^{L,i}$ of the last layer is a function of $\boldsymbol{\theta}$. The optimization problem to train a CNN is

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}), \tag{2.25}$$

where

$$f(\boldsymbol{\theta}) = \frac{1}{2C}\boldsymbol{\theta}^T\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i) \tag{2.26}$$

and the first term with the parameter $C > 0$ is used to avoid overfitting.

doi:10.6342/NTU201802108

CHAPTER III

Hessian-free Newton Methods for Training CNN

To solve an unconstrained minimization problem such as (2.25), a Newton method iteratively finds a search direction $\boldsymbol{d}$ by solving the following second-order approximation.

$$\min_{\boldsymbol{d}} \nabla f(\boldsymbol{\theta})^T \boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T \nabla^2 f(\boldsymbol{\theta})\boldsymbol{d}, \tag{3.1}$$

where $\nabla f(\boldsymbol{\theta})$ and $\nabla^2 f(\boldsymbol{\theta})$ are the gradient vector and the Hessian matrix, respectively. In this chapter we present details of applying a Newton method to solve the CNN problem (2.25).

3.1 Procedure of the Newton Method

For CNN, the gradient of $f(\boldsymbol{\theta})$ is

$$\nabla f(\boldsymbol{\theta}) = \frac{1}{C}\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l}(J^i)^T \nabla_{\boldsymbol{z}^{L,i}} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i), \tag{3.2}$$

where

$$J^i = \begin{bmatrix} \frac{\partial z_1^{L,i}}{\partial \theta_1} & \dots & \frac{\partial z_1^{L,i}}{\partial \theta_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial z_{n_L}^{L,i}}{\partial \theta_1} & \dots & \frac{\partial z_{n_L}^{L,i}}{\partial \theta_n} \end{bmatrix}_{n_L \times n}, \; i = 1, \dots, l, \tag{3.3}$$

is the Jacobian of $\boldsymbol{z}^{L,i}$. The Hessian matrix of $f(\boldsymbol{\theta})$ is

$$\nabla^2 f(\boldsymbol{\theta}) = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l}(J^i)^T B^i J^i$$

doi:10.6342/NTU201802108

$$+\frac{1}{l}\sum_{i=1}^{l}\sum_{j=1}^{n_L}\frac{\partial \xi(\boldsymbol{z}^{L,i};\boldsymbol{y}^i)}{\partial z_j^{L,i}}\begin{bmatrix}\frac{\partial^2 z_j^{L,i}}{\partial\theta_1\partial\theta_1} & \cdots & \frac{\partial^2 z_j^{L,i}}{\partial\theta_1\partial\theta_n}\\ \vdots & \ddots & \vdots\\ \frac{\partial^2 z_j^{L,i}}{\partial\theta_n\partial\theta_1} & \cdots & \frac{\partial^2 z_j^{L,i}}{\partial\theta_n\partial\theta_n}\end{bmatrix}, \tag{3.4}$$

where $\mathcal{I}$ is the identity matrix and

$$B_{ts}^i = \frac{\partial^2 \xi(\boldsymbol{z}^{L,i};\boldsymbol{y}^i)}{\partial z_t^{L,i}\partial z_s^{L,i}},\ t=1,\ldots,n_L,\ s=1,\ldots,n_L. \tag{3.5}$$

From now on for simplicity we let

$$\xi_i \equiv \xi_i(\boldsymbol{z}^{L,i};\boldsymbol{y}^i).$$

If $f(\boldsymbol{\theta})$ is non-convex as in the case of deep learning, (3.1) is difficult to solve and the resulting direction may not lead to the decrease of the function value. Thus the Gauss-Newton approximation (Schraudolph, 2002)

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l}(J^i)^T B^i J^i \approx \nabla^2 f(\boldsymbol{\theta}) \tag{3.6}$$

is often used. In particular, if G is positive definite, then (3.1) becomes the same as solving the following linear system.

$$G\boldsymbol{d} = -\nabla f(\boldsymbol{\theta}). \tag{3.7}$$

After a Newton direction $\boldsymbol{d}$ is obtained, to ensure the convergence, we update $\boldsymbol{\theta}$ by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{d},$$

where α is the largest element in $\{1, \frac{1}{2}, \frac{1}{4}, \ldots\}$ which can satisfy

$$f(\boldsymbol{\theta}+\alpha\boldsymbol{d}) \leq f(\boldsymbol{\theta}) + \eta\alpha\nabla f(\boldsymbol{\theta})^T\boldsymbol{d}, \tag{3.8}$$

where $\eta \in (0,1)$ is a pre-defined constant. The procedure to find α is called a back-tracking line search.

doi:10.6342/NTU201802108

Past works (e.g., Martens, 2010; Wang et al., 2018a) have shown that sometimes (3.7) is too aggressive, so a direction closer to the negative gradient is better. To this end, in recent works of applying Newton methods on fully-connected networks, the Levenberg-Marquardt method (Levenberg, 1944; Marquardt, 1963) is used to solve the following linear system rather than (3.7).

$$(G + \lambda \mathcal{I})\boldsymbol{d} = -\nabla f(\boldsymbol{\theta}), \tag{3.9}$$

where λ is a parameter decided by how good the function reduction is. Specifically, we define

$$\rho = \frac{f(\boldsymbol{\theta} + \boldsymbol{d}) - f(\boldsymbol{\theta})}{\nabla f(\boldsymbol{\theta})^T \boldsymbol{d} + \frac{1}{2}(\boldsymbol{d})^T G \boldsymbol{d}}$$

as the ratio between the actual function reduction and the predicted reduction. By using ρ, the parameter λ_{next} for the next iteration is decided by

$$\lambda_{\text{next}} = \begin{cases} \lambda \times \text{drop} & \rho > \rho_{\text{upper}}, \\ \lambda & \rho_{\text{lower}} \leq \rho \leq \rho_{\text{upper}}, \\ \lambda \times \text{boost} & \text{otherwise}, \end{cases} \tag{3.10}$$

where (drop,boost) are given constants. From (3.10) we can clearly see that if the function-value reduction is not satisfactory, then λ is enlarged and the resulting direction is closer to the negative gradient.

Next, we discuss how to solve the linear system (3.9). When the number of variables n is large, the matrix G is too large to be stored. For some optimization problems including neural networks, without explicitly storing G it is possible to calculate the product between G and any vector $\boldsymbol{v}$ (Le et al., 2011; Martens, 2010; Wang et al., 2018a). For example, from (3.6),

$$(G + \lambda \mathcal{I})\boldsymbol{v} = (\frac{1}{C} + \lambda)\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l}\left((J^i)^T \left(B^i(J^i\boldsymbol{v})\right)\right). \tag{3.11}$$

doi:10.6342/NTU201802108

If the product between J^i and a vector can be easily calculated, then G does not need to be explicitly formed. Therefore, we can apply the conjugate gradient (CG) method to solve (3.7) by a sequence of matrix-vector products. This technique is called Hessian-free methods in optimization. Details of CG methods in a Hessian-free Newton framework can be found in, for example, Algorithm 2 of Lin et al. (2007).

Because the computational cost in (3.11) is proportional to the number of instances, subsampled Hessian Newton methods have been proposed (Byrd et al., 2011; Martens, 2010; Wang et al., 2015) to reduce the cost in solving the linear system (3.9). They observe that the second term in (3.6) is the average training loss. If the large number of data points are assumed to be from the same distribution, (3.6) can be reasonablely approximated by selecting a subset $S \subset \{1, \dots, l\}$ and having

$$G^S = \frac{1}{C}\mathcal{I} + \frac{1}{|S|}\sum_{i \in S}(J^i)^T B^i J^i \approx G.$$

Then (3.11) becomes

$$(G^S + \lambda\mathcal{I})\boldsymbol{v} = (\frac{1}{C} + \lambda)\boldsymbol{v} + \frac{1}{|S|}\sum_{i \in S}\left((J^i)^T\left(B^i(J^i\boldsymbol{v})\right)\right) \approx (G + \lambda\mathcal{I})\boldsymbol{v}.$$

A summary of the Newton method is in Algorithm 1.

3.2 Gradient Evaluation

In order to solve (3.7), $\nabla f(\boldsymbol{\theta})$ is needed. It can be obtained by (3.2) if the Jacobian matrix $J^i,\ i = 1, \dots, l$ is available; see the discussion on Jacobian calculation in Chapter 3.3. However, we have mentioned in Chapter 3.1 that in practice, only a subset of J^i may be calculated. Thus here we present a direct calculation by a backward process.

Consider two layers $m-1$ and m. The variables between them are W^m and $\boldsymbol{b}^m$, so we aim to calculate the following gradient components.

$$\frac{\partial f}{\partial W^m} = \frac{1}{C}W^m + \frac{1}{l}\sum_{i=1}^{l}\frac{\partial \xi_i}{\partial W^m}, \tag{3.12}$$

doi:10.6342/NTU201802108

Algorithm 1 A standard subsampled Hessian Newton method for CNN.

1: Compute $f(\boldsymbol{\theta}^1)$.
2: **for** $k = 1, \ldots,$ **do**
3: Choose a set $S_k \subset \{1, \ldots, l\}$.
4: Compute $\nabla f(\boldsymbol{\theta}^k)$ and $J^i, \forall i \in S_k$.
5: Approximately solve the linear system in (3.9) by CG to obtain a direction $\boldsymbol{d}^k$
6: $\alpha = 1$.
7: **while** true **do**
8: Update $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \alpha \boldsymbol{d}^k$ and compute $f(\boldsymbol{\theta}^{k+1})$
9: **if** (3.8) is satisfied **then**
10: break
11: **end if**
12: $\alpha \leftarrow \alpha/2$.
13: **end while**
14: Calculate λ_{k+1} based on (3.10).
15: **end for**

$$\frac{\partial f}{\partial \boldsymbol{b}^m} = \frac{1}{C}\boldsymbol{b}^m + \frac{1}{l}\sum_{i=1}^{l} \frac{\partial \xi_i}{\partial \boldsymbol{b}^m}. \tag{3.13}$$

Because (3.12) is in a matrix form, following past developments such as Vedaldi and Lenc (2015), it is easier to transform them to a vector form for the derivation. To begin, we list the following properties of the vec$(\cdot)$ function, in which $\otimes$ is the kronecker product.

$$\text{vec}(AB) = (\mathcal{I} \otimes A)\text{vec}(B), \tag{3.14}$$

$$= (B^T \otimes \mathcal{I})\text{vec}(A), \tag{3.15}$$

$$\text{vec}(AB)^T = \text{vec}(B)^T(\mathcal{I} \otimes A^T), \tag{3.16}$$

$$= \text{vec}(A)^T(B \otimes \mathcal{I}). \tag{3.17}$$

doi:10.6342/NTU201802108

We further define

$$\frac{\partial \boldsymbol{y}}{\partial (\boldsymbol{x})^T} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_{|\boldsymbol{x}|}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{|\boldsymbol{y}|}}{\partial x_1} & \cdots & \frac{\partial y_{|\boldsymbol{y}|}}{\partial x_{|\boldsymbol{x}|}} \end{bmatrix},$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors, and let

$$\phi(\boldsymbol{z}^{m-1,i}) = \mathcal{I}_{n_{m-1}} \boldsymbol{z}^{m-1,i},\ L^c < m \leq L,$$

where $\mathcal{I}_p$ is the $p \times p$ identity matrix.

For the fully-connected layers, from (2.21), we have

$$\begin{aligned} \boldsymbol{s}^{m,i} &= W^m \boldsymbol{z}^{m-1,i} + \boldsymbol{b}^m \\ &= (\mathcal{I}_1 \otimes W^m)\, \boldsymbol{z}^{m-1,i} + (\mathbb{1}_1 \otimes \mathcal{I}_{n_m}) \boldsymbol{b}^m \qquad (3.18) \\ &= \left((\boldsymbol{z}^{m-1,i})^T \otimes \mathcal{I}_{n_m}\right) \text{vec}(W^m) + (\mathbb{1}_1 \otimes \mathcal{I}_{n_m}) \boldsymbol{b}^m, \qquad (3.19) \end{aligned}$$

where (3.18) and (3.19) are from (3.14) and (3.15), respectively. For the convolutional layers, from (2.11), we have

$$\begin{aligned} \text{vec}(S^{m,i}) &= \text{vec}(W^m \phi(Z^{m-1,i})) + \text{vec}(\boldsymbol{b}^m \mathbb{1}^T_{a^m b^m}) \\ &= (\mathcal{I}_{a^m b^m} \otimes W^m)\, \text{vec}(\phi(Z^{m-1,i})) + (\mathbb{1}_{a^m b^m} \otimes \mathcal{I}_{d^m}) \boldsymbol{b}^m \qquad (3.20) \\ &= \left(\phi(Z^{m-1,i})^T \otimes \mathcal{I}_{d^m}\right) \text{vec}(W^m) + (\mathbb{1}_{a^m b^m} \otimes \mathcal{I}_{d^m}) \boldsymbol{b}^m, \qquad (3.21) \end{aligned}$$

where (3.20) and (3.21) are from (3.14) and (3.15), respectively.

An advantage of using (3.18) and (3.20) is that they are in the same form, and so are (3.19) and (3.21). Thus we can derive the gradient together. We begin with calculating the gradient for convolutional layers. From (3.21), we derive

$$\begin{aligned} \frac{\partial \xi_i}{\partial \text{vec}(W^m)^T} &= \frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \frac{\partial \text{vec}(S^{m,i})}{\partial \text{vec}(W^m)^T} \\ &= \frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \left(\phi(Z^{m-1,i})^T \otimes \mathcal{I}_{d^m}\right) \end{aligned}$$

doi:10.6342/NTU201802108

$$= \text{vec}\left(\frac{\partial \xi_i}{\partial S^{m,i}} \phi(Z^{m-1,i})^T\right)^T \tag{3.22}$$

and

$$\begin{aligned}
\frac{\partial \xi_i}{\partial (\boldsymbol{b}^m)^T} &= \frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \frac{\partial \text{vec}(S^{m,i})}{\partial (\boldsymbol{b}^m)^T} \\
&= \frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \left(\mathbb{1}_{a^m b^m} \otimes \mathcal{I}_{d^m}\right) \\
&= \text{vec}\left(\frac{\partial \xi_i}{\partial S^{m,i}} \mathbb{1}_{a^m b^m}\right)^T, \qquad (3.23)
\end{aligned}$$

where (3.22) and (3.23) are from (3.17). To calculate (3.22), $\phi(Z^{m-1,i})$ is available in the forward process of calculating the function value.

For $\partial \xi_i / \partial S^{m,i}$ also needed in (3.22) and (3.23), it can be obtained by a backward process. Here we assume that $\partial \xi_i / \partial S^{m,i}$ is available, and calculate $\partial \xi_i / \partial S^{m-1,i}$ for layer $m-1$.

$$\begin{aligned}
\frac{\partial \xi_i}{\partial \text{vec}(Z^{m-1,i})^T} &= \frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \frac{\partial \text{vec}(S^{m,i})}{\partial \text{vec}(\phi(Z^{m-1,i}))^T} \frac{\partial \text{vec}(\phi(Z^{m-1,i}))}{\partial \text{vec}(Z^{m-1,i})^T} \\
&= \frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \left(\mathcal{I}_{a^m b^m} \otimes W^m\right) P_\phi^{m-1} \qquad (3.24) \\
&= \text{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}}\right)^T P_\phi^{m-1}, \qquad (3.25)
\end{aligned}$$

where (3.24) is from (2.8) and (3.20), and (3.25) is from (3.16).

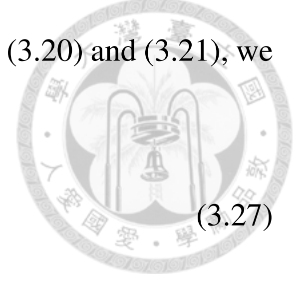
Then, because the RELU activation function is considered for the convolutional layers, we have

$$\frac{\partial \xi_i}{\partial \text{vec}(S^{m-1,i})^T} = \frac{\partial \xi_i}{\partial \text{vec}(Z^{m-1,i})^T} \odot \text{vec}(I[Z^{m-1,i}])^T, \tag{3.26}$$

where $\odot$ is Hadamard product (i.e., element-wise products) and

$$I[Z^{m-1,i}]_{(p,q)} = \begin{cases} 1 & \text{if } z_{(p,q)}^{m-1,i} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

doi:10.6342/NTU201802108



For fully-connected layers, by the same form in (3.18), (3.19), (3.20) and (3.21), we immediately get from (3.22), (3.23), (3.26) and (3.25) that

$$\frac{\partial \xi_i}{\partial \text{vec}(W^m)^T} = \text{vec}\left(\frac{\partial \xi_i}{\partial \boldsymbol{s}^{m,i}}(\boldsymbol{z}^{m-1,i})^T\right)^T, \tag{3.27}$$

$$\frac{\partial \xi_i}{\partial (\boldsymbol{b}^m)^T} = \frac{\partial \xi_i}{\partial (\boldsymbol{s}^{m,i})^T}, \tag{3.28}$$

$$\begin{aligned}\frac{\partial \xi_i}{\partial (\boldsymbol{z}^{m-1,i})^T} &= \left((W^m)^T \frac{\partial \xi_i}{\partial (\boldsymbol{s}^{m,i})}\right)^T \mathcal{I}_{n_{m-1}} \\ &= \frac{\partial \xi_i}{\partial (\boldsymbol{s}^{m,i})^T} W^m,\end{aligned} \tag{3.29}$$

$$\frac{\partial \xi_i}{\partial (\boldsymbol{s}^{m-1,i})^T} = \frac{\partial \xi_i}{\partial (\boldsymbol{z}^{m-1,i})^T} \odot I[\boldsymbol{z}^{m-1,i}]^T. \tag{3.30}$$

Finally, we check the initial values of the backward process. From the square loss in (2.24), we have

$$\begin{aligned}\frac{\partial \xi_i}{\partial \boldsymbol{z}^{L,i}} &= 2(\boldsymbol{z}^{L,i} - \boldsymbol{y}^{L,i}), \\ \frac{\partial \xi_i}{\partial \boldsymbol{s}^{L,i}} &= \frac{\partial \xi_i}{\partial \boldsymbol{z}^{L,i}}.\end{aligned}$$

3.2.1 Padding, Pooling, and the Overall Procedure

For the padding operation, from (2.15), we have

$$\begin{aligned}\frac{\partial \xi_i}{\partial \text{vec}(Z^{m-1,i})^T} &= \frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} \frac{\partial \text{vec}(Z^{m,i})}{\partial \text{vec}(Z^{m-1,i})^T} \\ &= \frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} P_{\text{padding}}^{m-1,i}.\end{aligned} \tag{3.31}$$

Similarly, for the pooling layer, from (2.19), we have

$$\frac{\partial \xi_i}{\partial \text{vec}(Z^{m-1,i})^T} = \frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} P_{\text{pool}}^{m-1,i}. \tag{3.32}$$

Following the explanation in Chapter 2.1.3, to generate $\partial \xi_i / \partial \text{vec}(S^{m-1,i})$ from $\partial \xi_i / \text{vec}(S^{m,i})$, we consider the following cycle.

$$S^{m-1} \rightarrow Z^{m-1} \rightarrow \text{pooling} \rightarrow \text{padding} \rightarrow \phi(Z^{m-1,i}) \rightarrow S^m.$$

doi:10.6342/NTU201802108

Therefore, by combining (3.25), (3.26), (3.31) and (3.32), details of obtaining $\partial \xi_i / \partial \text{vec}(Z^{m-1,i})^T$ is by

$$\frac{\partial \xi_i}{\partial \text{vec}(Z^{m-1,i})} = \text{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}} \right)^T P_\phi^{m-1} P_{\text{padding}}^{m-1} P_{\text{pool}}^{m-1}. \quad (3.33)$$

3.3 Jacobian Evaluation

For (3.6), the Jacobian matrix is needed and it can be partitioned into L blocks.

$$J^i = \begin{bmatrix} J^{1,i} & J^{2,i} & \dots & J^{L,i} \end{bmatrix}, \ m = 1, \dots, L, \ i = 1, \dots, l, \quad (3.34)$$

where

$$J^{m,i} = \begin{bmatrix} \frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix}.$$

The calculation is very similar to that for the gradient. For the convolutional layers, from (3.22) and (3.23), we have

$$\begin{aligned} \begin{bmatrix} \frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix} &= \begin{bmatrix} \frac{\partial z_1^{L,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial z_1^{L,i}}{\partial (\boldsymbol{b}^m)^T} \\ \vdots & \\ \frac{\partial z_{n_L}^{L,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial z_{n_L}^{L,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix} \\ &= \begin{bmatrix} \text{vec}(\frac{\partial z_1^{L,i}}{\partial S^{m,i}} \phi(Z^{m-1,i})^T)^T & \text{vec}(\frac{\partial z_1^{L,i}}{\partial S^{m,i}} \mathbb{1}_{a^m b^m})^T \\ \vdots & \\ \text{vec}(\frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}} \phi(Z^{m-1,i})^T)^T & \text{vec}(\frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}} \mathbb{1}_{a^m b^m})^T \end{bmatrix} \\ &= \begin{bmatrix} \text{vec}\left(\frac{\partial z_1^{L,i}}{\partial S^{m,i}} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right)^T \\ \vdots \\ \text{vec}\left(\frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right)^T \end{bmatrix}. \end{aligned} \quad (3.35)$$

In the backward process, if $\partial \boldsymbol{z}^{L,i} / \partial \text{vec}(S^{m,i})^T$ is available, we calculate $\partial \boldsymbol{z}^{L,i} / \partial \text{vec}(S^{m-1,i})^T$ for convolutional layer $m-1$. From a derivation similar to (3.25) for the gradient, we

doi:10.6342/NTU201802108

have

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(Z^{m-1,i})^T} = \begin{bmatrix} \frac{\partial z_1^{L,i}}{\partial \text{vec}(Z^{m-1,i})^T} \\ \vdots \\ \frac{\partial z_{n_L}^{L,i}}{\partial \text{vec}(Z^{m-1,i})^T} \end{bmatrix} = \begin{bmatrix} \text{vec}\left((W^m)^T \frac{\partial z_1^{L,i}}{\partial S^{m,i}}\right)^T P_\phi^{m-1} \\ \vdots \\ \text{vec}\left((W^m)^T \frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}}\right)^T P_\phi^{m-1} \end{bmatrix}. \tag{3.36}$$

Similar to (3.26),

$$\frac{\partial z_j^{L,i}}{\partial \text{vec}(S^{m-1,i})^T} = \frac{\partial z_j^{L,i}}{\partial \text{vec}(Z^{m-1,i})^T} \odot \text{vec}(I[Z^{m-1,i}])^T,\ j = 1, \ldots, n_L.$$

They can be written together as

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m-1,i})^T} = \frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(Z^{m-1,i})^T} \odot \left(\mathbb{1}_{n_L} \text{vec}(I[Z^{m-1,i}])^T\right). \tag{3.37}$$

For the padding operation and pooling layer, similar to (3.31) and (3.32), we have

$$\frac{\partial \boldsymbol{z}_i}{\partial \text{vec}(Z^{m-1,i})^T} = \frac{\partial \boldsymbol{z}_i}{\partial \text{vec}(Z^{m,i})^T} P_{\text{padding}}^{m-1,i} \tag{3.38}$$

and

$$\frac{\partial \boldsymbol{z}_i}{\partial \text{vec}(Z^{m-1,i})^T} = \frac{\partial \boldsymbol{z}_i}{\partial \text{vec}(Z^{m,i})^T} P_{\text{pool}}^{m-1,i} \tag{3.39}$$

respectively.

For the fully-connected layers, we follow the same derivation of gradient. Thus, we get

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(W^m)^T} = \left[\text{vec}\left(\frac{\partial z_1^{L,i}}{\partial \boldsymbol{s}^{m,i}}(\boldsymbol{z}^{m-1,i})^T\right) \ \ldots \ \text{vec}\left(\frac{\partial z_{n_L}^{L,i}}{\partial \boldsymbol{s}^{m,i}}(\boldsymbol{z}^{m-1,i})^T\right)\right]^T \tag{3.40}$$

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{b}^m)^T} = \frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{s}^{m,i})^T} \tag{3.41}$$

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{z}^{m-1,i})^T} = \frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{s}^{m,i})^T} W^m \tag{3.42}$$

doi:10.6342/NTU201802108

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{s}^{m-1,i})^T} = \frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{z}^{m-1,i})^T} \odot \left(\mathbb{1}_{n_L} I[\boldsymbol{z}^{m-1,i}]^T\right) \tag{3.43}$$

For layer L, because of using (2.24) and the linear activation function, we have

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{s}^{L,i})^T} = \mathcal{I}_{n_L}.$$

3.4 Gauss-Newton Matrix-Vector Products

For solving (3.7), conjugate gradient (CG) methods are often applied and the main operation at each CG iteration is the Gauss-Newton matrix-vector product.

From (3.34), we rearrange (3.6) to

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} B^i \begin{bmatrix} J^{1,i} & \dots & J^{L,i} \end{bmatrix} \tag{3.44}$$

and the Gauss-Newton matrix vector product becomes

$$\begin{aligned} G\boldsymbol{v} &= \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} B^i \begin{bmatrix} J^{1,i} & \dots & J^{L,i} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^L \end{bmatrix} \\ &= \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} \left(B^i \sum_{m=1}^{L} J^{m,i}\boldsymbol{v}^m \right) \end{aligned} \tag{3.45}$$

where

$$\boldsymbol{v} = \begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^L \end{bmatrix}, \ \boldsymbol{v}^m = \begin{bmatrix} \boldsymbol{v}^m_w \\ \boldsymbol{v}^m_b \end{bmatrix}, \ m = 1, \dots, L.$$

In this subsection, we focus on the ith instance and give the implementation details of calculating (3.45) for the whole data in Chapter 4.5.

doi:10.6342/NTU201802108

For the convolutional layers, from (3.35) and (3.45), we first calculate

$$J^{m,i}\boldsymbol{v}^m = \begin{bmatrix} \text{vec}\left(\frac{\partial z_1^{L,i}}{\partial S^{m,i}}\left[\phi(Z^{m-1,i})^T\ \mathbb{1}_{a^m b^m}\right]\right)^T \boldsymbol{v}^m \\ \vdots \\ \text{vec}\left(\frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}}\left[\phi(Z^{m-1,i})^T\ \mathbb{1}_{a^m b^m}\right]\right)^T \boldsymbol{v}^m \end{bmatrix}. \tag{3.46}$$

To simplify (3.46), we use the following property

$$\text{vec}(AB)^T\text{vec}(C) = \text{vec}(A)^T\text{vec}(CB^T)$$

to have for example, the first element in (3.46) is

$$\begin{aligned} &\text{vec}\left(\frac{\partial z_1^{L,i}}{\partial S^{m,i}}\left[\phi(Z^{m-1,i})^T\ \mathbb{1}_{a^m b^m}\right]\right)^T \boldsymbol{v}^m \\ =& \frac{\partial z_1^{L,i}}{\partial \text{vec}(S^{m,i})^T}\text{vec}\left(\text{mat}(\boldsymbol{v}^m)_{d^m\times(h^m h^m d^{m-1}+1)}\begin{bmatrix}\phi(Z^{m-1,i}) \\ \mathbb{1}^T_{a^m b^m}\end{bmatrix}\right). \end{aligned}$$

Therefore,

$$J^{m,i}\boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T}\text{vec}\left(\text{mat}(\boldsymbol{v}^m)_{d^m\times(h^m h^m d^{m-1}+1)}\begin{bmatrix}\phi(Z^{m-1,i}) \\ \mathbb{1}^T_{a^m b^m}\end{bmatrix}\right). \tag{3.47}$$

After deriving (3.47), from (3.45), we sum results of all layers

$$\sum_{m=1}^{L} J^{m,i}\boldsymbol{v}^m$$

and then calculate

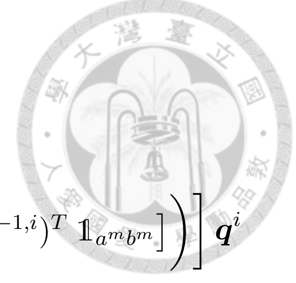
$$\boldsymbol{q}^i = B^i(\sum_{m=1}^{L} J^{m,i}\boldsymbol{v}^m). \tag{3.48}$$

From (2.24) and (3.5),

$$B^i_{ts} = \frac{\partial^2 \xi^i}{\partial z_t^{L,i}\partial z_s^{L,i}} = \frac{\partial^2(\sum_{j=1}^{n_L}(z_j^{L,i}-y_j^i)^2)}{\partial z_t^{L,i}\partial z_s^{L,i}} = \begin{cases} 2 & \text{if } t=s, \\ 0 & \text{otherwise,} \end{cases} \tag{3.49}$$

and we derive $\boldsymbol{q}^i$ by multiplying every element of $\sum_{m=1}^{L} J^{m,i}\boldsymbol{v}^m$ by two.

doi:10.6342/NTU201802108



After deriving (3.48), from (3.35) and (3.45), we calculate

$$
\begin{aligned}
&(J^{m,i})^T \boldsymbol{q}^i \\
=& \left[\text{vec}\left(\tfrac{\partial z_1^{L,i}}{\partial S^{m,i}} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right) \ \cdots \ \text{vec}\left(\tfrac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right) \right] \boldsymbol{q}^i \\
=& \sum_{j=1}^{n_L} q_j^i \text{vec}\left(\frac{\partial z_j^{L,i}}{\partial S^{m,i}} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right) \\
=& \text{vec}\left(\sum_{j=1}^{n_L} q_j^i \left(\frac{\partial z_j^{L,i}}{\partial S^{m,i}} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right) \right) \\
=& \text{vec}\left(\left(\sum_{j=1}^{n_L} q_j^i \frac{\partial z_j^{L,i}}{\partial S^{m,i}} \right) \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right) \\
=& \text{vec}\left(\text{mat}\left(\left(\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T} \right)^T \boldsymbol{q}^i \right)_{d^m \times a^m b^m} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right). \qquad (3.50)
\end{aligned}
$$

Similar to the results of the convolutional layers, for the fully-connected layers, we have

$$
J^{m,i} \boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{s}^{m,i})^T} \text{mat}(\boldsymbol{v}^m)_{n_m \times (n_{m-1}+1)} \begin{bmatrix} \boldsymbol{z}^{m-1,i} \\ \mathbb{1}_1 \end{bmatrix}. \qquad (3.51)
$$

$$
(J^{m,i})^T \boldsymbol{q}^i = \text{vec}\left(\left(\frac{\partial \boldsymbol{z}^{L,i}}{\partial (\boldsymbol{s}^{m,i})^T} \right)^T \boldsymbol{q}^i \left[(\boldsymbol{z}^{m-1,i})^T \ \mathbb{1}_1 \right] \right). \qquad (3.52)
$$

doi:10.6342/NTU201802108

CHAPTER IV

Implementation Details

Our goal in this chapter is to show that after some careful derivations, a Newton method for CNN can be implemented by a simple and short program. Here we give a *MATLAB* implementation though a modification for other languages such as *Python* should be straightforward.

For the discussion in Chapter III, we consider a single data instance, but for practical implementations, all instances must be taken care of. In out implementation, we stored $Z^{m-1,i}$, $\forall i = 1, \ldots, l$ as the following matrix.

$$\begin{bmatrix} Z^{m-1,1} & Z^{m-1,2} & \ldots & Z^{m-1,l} \end{bmatrix} \in R^{d^{m-1} \times a^{m-1} b^{m-1} l}. \tag{4.1}$$

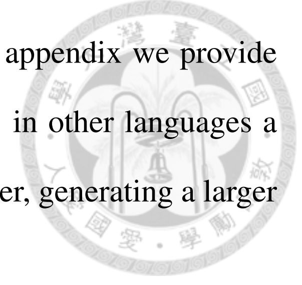
Similarly, we stored $\partial \boldsymbol{z}^{L,i} / \partial \text{vec}(S^{m,i})^T$ as

$$\begin{bmatrix} \frac{\partial z_1^{L,1}}{\partial \text{vec}(S^{m,i})^T} & \ldots & \frac{\partial z_1^{L,l}}{\partial \text{vec}(S^{m,i})^T} & \ldots & \frac{\partial z_{n_L}^{L,1}}{\partial \text{vec}(S^{m,i})^T} & \ldots & \frac{\partial z_{n_L}^{L,l}}{\partial \text{vec}(S^{m,i})^T} \end{bmatrix}^T \in R^{l n_L \times d^m a^m b^m}. \tag{4.2}$$

4.1 Generation of $\phi(Z^{m-1,i})$

MATLAB has a built-in function `im2col` that can generate $\phi(Z^{m-1,i})$ for $s^m = 1$ and $s^m = h^m$. To handle general s^m, we notice that $\phi(Z^{m-1,i})$ is a sub-matrix of the

doi:10.6342/NTU201802108



output matrix of using *MATLAB*'s `im2col` under $s^m = 1$. In appendix we provide an efficient implementation to extract the sub-matrix. However, in other languages a subroutine like *MATLAB*'s `im2col` may not be available. Further, generating a larger matrix under $s^m = 1$ causes extra time and memory.

Therefore, here we show an efficient implementation to get $\phi(Z^{m-1,i})$ without relying on a subroutine like *MATLAB*'s `im2col`. To begin we consider the following linear indices[1] (i.e., counting elements in a column-oriented way) of $Z^{m-1,i}$:

$$\begin{bmatrix} 1 & d^{m-1}+1 & \dots & (b^{m-1}a^{m-1}-1)d^{m-1}+1 \\ 2 & d^{m-1}+2 & \dots & (b^{m-1}a^{m-1}-1)d^{m-1}+2 \\ \vdots & \vdots & \ddots & \vdots \\ d^{m-1} & 2d^{m-1} & \dots & (b^{m-1}a^{m-1})d^{m-1} \end{bmatrix} \in R^{d^{m-1}\times a^{m-1}b^{m-1}}. \tag{4.3}$$

Because every element in

$$\phi(Z^{m-1,i}) \in R^{h^m h^m d^{m-1}\times a^m b^m},$$

is extracted from $Z^{m-1,i}$, the task is to find the mapping between each element in $\phi(Z^{m-1,i})$ and a linear index of $Z^{m-1,i}$. Consider the following example.

$$a^{m-1} = b^{m-1} = 2,\ d^{m-1} = 1,\ s^m = 1,\ h^m = 2.$$

Because $d^{m-1} = 1$, we omit the channel subscript. In addition, we also omit the instance index i. Thus the image and its linear indices are

$$\begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \\ z_{31} & z_{32} \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

[1] Linear indices refer to the sequence of how elements in a matrix are stored. Here we consider a column-oriented setting.

doi:10.6342/NTU201802108

We have

$$\phi(Z^{m-1}) = \begin{bmatrix} z_{11} & z_{21} \\ z_{21} & z_{31} \\ z_{12} & z_{22} \\ z_{22} & z_{32} \end{bmatrix}.$$

Thus we store the following vector to indicate the mapping between linear indices of $Z^{m-1,i}$ and $\phi(Z^{m-1,i})$.

$$[\,1\ \ 2\ \ 4\ \ 5\ \ 2\ \ 3\ \ 5\ \ 6\,]^T. \tag{4.4}$$

It also corresponds to row indices of non-zero elements in P_{ϕ}^{m-1}.

We begin with checking how linear indices of $Z^{m-1,i}$ can be mapped to the first column of $\phi(Z^{m-1})$. For simplicity, we consider only channel j. From (2.9) and (4.3), we have

$$\left[\begin{array}{c|c} j & z_{1,1,j}^{m-1} \\ d^{m-1}+j & z_{2,1,j}^{m-1} \\ \vdots & \vdots \\ (h^m-1)d^{m-1}+j & z_{h^m,1,j}^{m-1} \\ a^{m-1}d^{m-1}+j & z_{1,2,j}^{m-1} \\ \vdots & \vdots \\ ((h^m-1)+a^{m-1})d^{m-1}+j & z_{h^m,2,j}^{m-1} \\ \vdots & \vdots \\ ((h^m-1)+(h^m-1)a^{m-1})d^{m-1}+j & z_{h^m,h^m,j}^{m-1} \end{array}\right], \tag{4.5}$$

where the left column gives the linear indices in $Z^{m-1,i}$, while the right column shows

doi:10.6342/NTU201802108

the corresponding values. We rewrite linear indices in (4.5) as

$$\begin{bmatrix} 0 + 0a^{m-1} \\ \vdots \\ (h^m - 1) + 0a^{m-1} \\ 0 + 1a^{m-1} \\ \vdots \\ (h^m - 1) + 1a^{m-1} \\ \vdots \\ 0 + (h^m - 1)a^{m-1} \\ \vdots \\ (h^m - 1) + (h^m - 1)a^{m-1} \end{bmatrix} d^{m-1} + j. \tag{4.6}$$

Clearly, every linear index in (4.6) can be represented as

$$(p + qa^{m-1})d^{m-1} + j, \tag{4.7}$$

where

$$p,\ q \in \{0, \ldots, h^m - 1\}$$

correspond to the pixel position in the convolutional filter.[2]

Next we consider other columns in $\phi(Z^{m-1,i})$ by still fixing the channel to be j. From (2.9), similar to the right column in (4.5), each column contains the following elements from the jth channel of $Z^{m-1,i}$.

$$\begin{aligned} z^{m-1,i}_{1+p+as^m,1+q+bs^m,j}\ ,\ \ & a = 0, 1, \ldots, a^m - 1, \\ & b = 0, 1, \ldots, b^m - 1, \end{aligned} \tag{4.8}$$

where $(1 + as^m,\ 1 + bs^m)$ denotes the top-left position of a sub-image in the channel j

[2] More precisely, $p + 1$ and $q + 1$ are the pixel position.

doi:10.6342/NTU201802108

of $Z^{m-1,i}$. From (4.3), the linear index of each element in (4.8) is

$$\begin{aligned} &((1+p+as^m-1)+(1+q+bs^m-1)a^{m-1})d^{m-1}+j \\ &=(a+ba^{m-1})s^m d^{m-1}+\underbrace{(p+qa^{m-1})d^{m-1}+j}_{\text{see (4.7)}}. \end{aligned} \tag{4.9}$$

Now we have known for each element of $\phi(Z^{m-1,i})$ what the corresponding linear index in $Z^{m-1,i}$ is. Next we discuss the implementation details, where the code is shown in Listing IV.1. First, we compute elements in (4.6) with $j=1$ by applying *MATLAB*'s `bsxfun` function on the following two arrays.

$$\begin{bmatrix} 1 \\ d^{m-1}+1 \\ \vdots \\ (h^m-1)d^{m-1}+1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & a^{m-1}d^{m-1} & \dots & (h^m-1)a^{m-1}d^{m-1} \end{bmatrix}.$$

The results is the following matrix

$$\begin{bmatrix} 1 & a^{m-1}d^{m-1}+1 & \dots & (h^m-1)a^{m-1}d^{m-1}+1 \\ d^{m-1}+1 & (1+a^{m-1})d^{m-1}+1 & \dots & (1+(h^m-1)a^{m-1})d^{m-1}+1 \\ \vdots & \vdots & \dots & \vdots \\ (h^m-1)d^{m-1}+1 & ((h^m-1)+a^{m-1})d^{m-1}+1 & \dots & ((h^m-1)+(h^m-1)a^{m-1})d^{m-1}+1 \end{bmatrix}, \tag{4.10}$$

whose columns, if concatenated, lead to values in (4.6) with $j=1$; see line 2 of the code. To get (4.7) for all channels $j=1,\dots,d^{m-1}$, we apply `bsxfun` again to plus the vector form of (4.10) and

$$\begin{bmatrix} 0 & 1 & \dots & d^{m-1}-1 \end{bmatrix},$$

and then vectorize the resulting matrix; see line 3.

To obtain other columns in $\phi(Z^{m-1,i})$, next we calculate a^m and b^m by (2.4) in lines 4-5. To get all linear indices in (4.9), we see that the second term corresponds to indices

doi:10.6342/NTU201802108

of the first column and therefore we must calculate the following column offset

$$(a + ba^{m-1})s^m d^{m-1}, \ \forall a = 0, 1, \ldots, a^m - 1,$$
$$b = 0, 1, \ldots, b^m - 1.$$

This is by a `bsxfun` to plus the following two arrays.

$$\begin{bmatrix} 0 \\ \vdots \\ a^m - 1 \end{bmatrix} \times s^m d^{m-1} \quad \text{and} \quad \begin{bmatrix} 0 & \ldots & b^m - 1 \end{bmatrix} \times a^{m-1} s^m d^{m-1};$$

see line 6 in the code. Finally, we use another `bsxfun` to add the column offset and the values in (4.6); see line 7.

The obtained linear indices are independent of Z's values. Thus the above procedure only needs to be run once in the beginning. For any Z, we apply the indices to extract $\phi(Z)$; see line 20-21 in Listing IV.1.

For $\phi(Z^{m-1,i})$ in the pooling layer, the same implementation can be used.

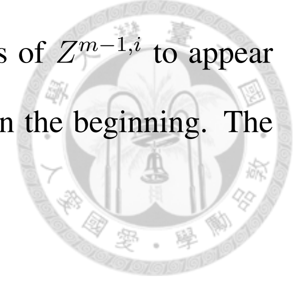
4.2 Construction of $P_{\text{pool}}^{m-1,i}$

Following (2.19), we assume that $Z^{m-1,i}$ and $\phi(Z^{m-1,i})$ are input and output before/after the pooling operation, respectively. In the beginning of the training procedure, we use the proposed procedure in Chapter 4.1 to have $\phi(Z^{m-1,i})$ that partitions each non-overlapping sub-regions; see (2.16) and (2.17). At each Newton iteration, Listing IV.2 is a *MATLAB* implementation for constructing $P_{\text{pool}}^{m-1,i}$, $\forall i$. In line 13, we handle the situation if

$$a^{m-1} \mod h^m \neq 0 \quad \text{or} \quad b^{m-1} \mod h^m \neq 0. \tag{4.11}$$

That is, we must ensure that (2.17) holds with a^m and b^m being integers. If (4.11) occurs, we simply discard the last several rows or columns in $Z^{m-1,i}$ to make a^{m-1} and

doi:10.6342/NTU201802108



b^{m-1} be multiples of h^m. In line 8, we extract the linear indices of $Z^{m-1,i}$ to appear in $\text{vec}(\phi(Z^{m-1,i}))$, which as we mentioned has been generated in the beginning. The resulting vector `P` contains

$$h^m h^m d^{m-1} a^m b^m$$

elements and each element is in the range of

$$1, \ \ldots, \ d^{m-1} a^{m-1} b^{m-1}.$$

In line 9, we get

$$Z^{m-1,i}, \ i = 1, \ldots, l, \tag{4.12}$$

which are stored as a matrix in (4.1). In line 23-24, we use `P` to generate

$$\left[\text{vec}(\phi(Z^{m-1,1})) \ \cdots \ \text{vec}(\phi(Z^{m-1,l}))\right] \in h^m h^m d^{m-1} a^m b^m \times l. \tag{4.13}$$

Next we rewrite the above matrix so that each column contains a sub-region:

$$\begin{bmatrix} z^{m-1,i}_{1,1,1} & z^{m-1,i}_{1,1,2} & \cdots & z^{m-1,l}_{1+(a^m-1)\times s^m, 1+(b^m-1)\times s^m, d^{m-1}} \\ \vdots & \vdots & \ddots & \vdots \\ z^{m-1,i}_{h^m,h^m,1} & z^{m-1,i}_{h^m,h^m,2} & \cdots & z^{m-1,l}_{h^m+(a^m-1)\times s^m, h^m+(b^m-1)\times s^m, d^{m-1}} \end{bmatrix} \in R^{h^m h^m \times d^{m-1} a^m b^m l}. \tag{4.14}$$

We apply a max function to get the largest value of each column and its index in the range of $1, \ldots, h^m h^m$. The resulting row vector has $d^m a^m b^m l$ elements, where $d^m = d^{m-1}$; see line 25. In line 26, we reformulate it to be

$$d^m \times a^m b^m l$$

as the output $Z^{m,i}$, $\forall i$.

Next we find linear indices in the matrix (4.13) that corresponds to the largest elements obtained from (4.14).[3] First we obtain linear indices of (4.1), which are in the

[3] Note that we do not really generate a sparse matrix $P^{m-1,i}_{\text{pool}}$ in (2.19).

doi:10.6342/NTU201802108

range of

$$1, \ldots, d^{m-1}a^{m-1}b^{m-1}l.$$

We store these indices in a matrix the same size as in (4.1); see line 19. Then we generate

$$\phi(\text{linear indices of (4.1)}), \tag{4.15}$$

which has the same size as in (4.13); see line 28-29. Next, if we can find positions in (4.15) that correspond to the largest elements identified earlier, then we have the desired linear indices. We mentioned that in line 25, not only the maximal value in each sub-region is obtained, but also we get the corresponding index in $\{1, \ldots, h^m h^m\}$. Therefore, for these max values, their linear indices in (4.14)[4] are

$$\begin{bmatrix} \text{row indices of} \\ \text{max values in (4.14)} \end{bmatrix} + h^m h^m \begin{bmatrix} 0 \\ \vdots \\ d^{m-1}a^m b^m l - 1 \end{bmatrix}. \tag{4.16}$$

The reason is that $h^m h^m$ is the column offset in (4.14). See line 30 for the implementation of (4.16). Next in line 31 we use (4.16) to extract values in (4.15) and obtain linear indices in (4.1) for the selected max values. Similar to the situation in Chapter 4.1 for $\phi(Z^{m-1,i})$, the index finding procedure in Listing IV.2 is conducted only once in the beginning of the training process.

Note that because max pooling depends on $Z^{m-1,i}$ values, linear indices for $P^{m-1,i}_{\text{pool}}$ must be re-generated in the beginning of each Newton iteration.

4.3 Details of Padding Operation

To implement zero-padding, we first capture the linear indices of the input image in the padded image. For example, if the size of the imput image is 3×3 and the output

[4] Also linear indices in (4.13) because (4.14) is separating each column in (4.13) to several columns.

doi:10.6342/NTU201802108

padded image is 5×5, we have

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

where "1" values indicate positions of the input image. Based on the column-major order, we derive

$$\texttt{pad_idx_one} = \{7,\ 8,\ 9,\ 11,\ 12,\ 13,\ 16,\ 17,\ 18\}.$$

It is obtained in the begining of the training procedure and it will be used in the following situations. First, `pad_idx_one` contains row indices in P^{m-1}_{padding} of (2.15) that correspond to the input image. We can use it to conduct the padding operation in (2.15). Second, from (3.33), in gradient and Jacobian evaluations, we need

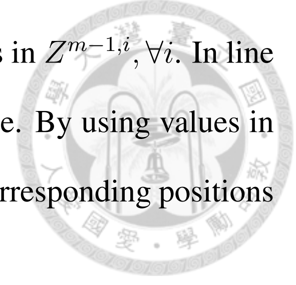
$$\boldsymbol{v}^T P^{m-1}_{\text{padding}}.$$

This can be considered as the inverse of the padding operation: we would like to remove zeros and get back the original image.

MATLAB implementations are shown in Listing IV.4 and IV.5, where Listing IV.4 is for generating the indices and Listing IV.5 is for the padding operation (2.15). In line 12 of Listing IV.4, we use the *MATLAB* bulit-in function `padarray` to pad zeros around the input image. Because $Z^{m-1,i}, \forall i$ are stored as a matrix in (4.1), in line 8 of Listing IV.5, we extend `pad_idx_one` to conver all instances:

$$\begin{bmatrix} \texttt{pad_idx_one} & \begin{matrix} \texttt{pad_idx_one} \\ + a^m b^m \end{matrix} & \begin{matrix} \texttt{pad_idx_one} \\ + 2a^m b^m \end{matrix} & \dots \end{bmatrix}. \tag{4.17}$$

doi:10.6342/NTU201802108



These values indicate the new column indices of original columns in $Z^{m-1,i}$, $\forall i$. In line 9 of Listing IV.5, we create the zeros of the output padding image. By using values in (4.17), in line 10 of Listing IV.5, we put the input $Z^{m-1,i}$ to the corresponding positions of the output padding image.

4.4 Evaluation of $\boldsymbol{v}^T P_\phi^{m-1}$

For (3.25) and (3.36), the following operation is applied.

$$\boldsymbol{v}^T P_\phi^{m-1}, \tag{4.18}$$

where

$$\boldsymbol{v} = \text{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}}\right)$$

for (3.25) and

$$\boldsymbol{v} = \text{vec}\left((W^m)^T \frac{\partial z_j^{L,i}}{\partial S^{m,i}}\right), \ j = 1, \ldots, n_L \tag{4.19}$$

for (3.36).

Consider the same example in Chapter 4.1. We note that

$$(P_\phi^{m-1})^T \boldsymbol{v} = \left[\, v_1 \ \ v_2 + v_5 \ \ v_6 \ \ v_3 \ \ v_4 + v_7 \ \ v_8 \,\right]^T, \tag{4.20}$$

which is a kind of "inverse" operation of $\phi(Z^{m-1})$: we accumulate elements in $\phi(Z^{m-1})$ back to their original positions in Z^{m-1}. In *MATLAB*, given indices in (4.4), a function `accumarray` can directly generate the vector (4.20).

To calculate (3.25) over a batch of instances, we aim to have

$$\begin{bmatrix} (P_\phi^{m-1})^T \boldsymbol{v}_1 \\ \vdots \\ (P_\phi^{m-1})^T \boldsymbol{v}_l \end{bmatrix}^T. \tag{4.21}$$

doi:10.6342/NTU201802108

We can manage to apply *MATLAB*'s `accumarray` on the vector

$$\begin{bmatrix} \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_l \end{bmatrix} \tag{4.22}$$

by giving the following indices as the input.

$$\begin{bmatrix} (4.4) \\ (4.4) + a^{m-1}b^{m-1}d^{m-1}\mathbb{1}_{a^m b^m d^{m-1} h^m h^m} \\ (4.4) + 2a^{m-1}b^{m-1}d^{m-1}\mathbb{1}_{a^m b^m d^{m-1} h^m h^m} \\ \vdots \\ (4.4) + (l-1)a^{m-1}b^{m-1}d^{m-1}\mathbb{1}_{a^m b^m d^{m-1} h^m h^m} \end{bmatrix}, \tag{4.23}$$

where

$$a^{m-1}b^{m-1}d^{m-1} \text{ is the size of } Z^{m-1,i}, \text{ and}$$
$$a^m b^m d^{m-1} h^m h^m \text{ is the size of } \phi(Z^{m-1,i}) \text{ and } \boldsymbol{v}_i.$$

That is, by using the offset $(i-1)a^{m-1}b^{m-1}d^{m-1}$, `accumarray` accumulate $\boldsymbol{v}_i$ to the following positions:

$$(i-1)a^{m-1}b^{m-1}d^{m-1} + 1, \ \ldots, \ ia^{m-1}b^{m-1}d^{m-1}. \tag{4.24}$$

We can easily rearrange the resulting vector by `reshape` and transpose operator to get the matrix (3.25), where each row is corresponding to an instance. We show the program in Listing IV.6, where line 8 generates the indices shown in (4.23) and `V(:)` in line 9 is the vector (4.22).

Following the same idea, to calculate (3.36) over a batch of instances, we do not have to calculate (4.18) n_L times. Notice that (3.36) is equivalent to

$$\left((P_\phi^{m-1})^T \left[\text{vec}\left((W^m)^T \frac{\partial z_1^{L,i}}{\partial S^{m,i}} \right) \ \ldots \ \text{vec}\left((W^m)^T \frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}} \right) \right] \right)^T. \tag{4.25}$$

doi:10.6342/NTU201802108

We can calculate $\boldsymbol{v}$ by vectorizing the result of fast matrix-matrix multiplication.

$$\boldsymbol{v} = \begin{bmatrix} \text{vec}\left((W^m)^T \frac{\partial z_1^{L,i}}{\partial S^{m,i}}\right) \\ \vdots \\ \text{vec}\left((W^m)^T \frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}}\right) \end{bmatrix} = \text{vec}\left((W^m)^T \begin{bmatrix} \frac{\partial z_1^{L,i}}{\partial S^{m,i}} & \dots & \frac{\partial z_{n_L}^{L,i}}{\partial S^{m,i}} \end{bmatrix}\right). \tag{4.26}$$

By passing the appropriate input indices to `accumarray`, we can calculate

$$(P_\phi^{m-1})^T \left[(4.26) \right].$$

So the calculation is fallen back to the same case as evaluating (3.25). That is, after having (4.26), we calculate (4.18) once instead of calculating (4.18) and (4.19) n_L times.

The input indices for `accumarray` are easy to be obtained. Before the transpose operation in (4.25), the linear indices for each column, i.e.

$$(P_\phi^{m-1})^T \text{vec}\left((W^m)^T \frac{\partial z_u^{L,i}}{\partial S^{m,i}}\right), \quad \forall u = \{1, \dots, n_L\}$$

is start from

$$(u-1)la^{m-1}b^{m-1}d^{m-1} + 1.$$

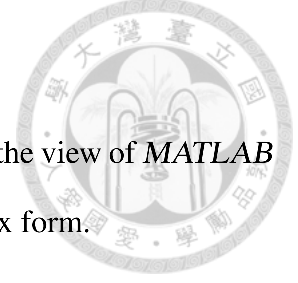
Therefore, we can apply `accumarray` on the vector (4.26) with the following input indices.

$$\begin{bmatrix} (4.23) \\ (4.23) + la^{m-1}b^{m-1}d^{m-1}\mathbb{1}_{a^m b^m d^{m-1} h^m h^m} \\ (4.23) + 2la^{m-1}b^{m-1}d^{m-1}\mathbb{1}_{a^m b^m d^{m-1} h^m h^m} \\ \vdots \\ (4.23) + (n_L - 1)la^{m-1}b^{m-1}d^{m-1}\mathbb{1}_{a^m b^m d^{m-1} h^m h^m} \end{bmatrix}.$$

The program is the same as Listing IV.6, except that we pass the value of the product of `n_L` and `S_k` for the parameter `S_k`.

doi:10.6342/NTU201802108

4.5 Gauss-Newton Matrix-Vector Products



Because (3.45) is the form of summation of every instance, in the view of *MATLAB* implementation, we extend (3.45) to the whole data set in a matrix form.

$$
\begin{aligned}
G\boldsymbol{v} &= \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} \left(B^i \sum_{m=1}^{L} J^{m,i}\boldsymbol{v}^m \right) \\
&= \frac{1}{C}\boldsymbol{v} + \frac{1}{l} \begin{bmatrix} (J^{1,1})^T & \dots & (J^{1,l})^T \\ \vdots & \ddots & \vdots \\ (J^{L,1})^T & \dots & (J^{L,l})^T \end{bmatrix} \begin{bmatrix} B^1 \sum_{m=1}^{L} J^{m,1}\boldsymbol{v}^m \\ \vdots \\ B^l \sum_{m=1}^{L} J^{m,l}\boldsymbol{v}^m \end{bmatrix}.
\end{aligned} \tag{4.27}
$$

To derive (4.27), we first calculate

$$
\begin{bmatrix} \sum_{m=1}^{L} J^{m,1}\boldsymbol{v}^m \\ \vdots \\ \sum_{m=1}^{L} J^{m,l}\boldsymbol{v}^m \end{bmatrix}. \tag{4.28}
$$

From (3.47), for a particular m, we have

$$
\begin{aligned}
\begin{bmatrix} J^{m,1}\boldsymbol{v}^m \\ \vdots \\ J^{m,l}\boldsymbol{v}^m \end{bmatrix} &= \begin{bmatrix} \frac{\partial \boldsymbol{z}^{L,1}}{\partial \text{vec}(S^{m,1})^T} \text{vec}\left(\text{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(Z^{m-1,1}) \\ \mathbb{1}^T_{a^m b^m} \end{bmatrix} \right) \\ \vdots \\ \frac{\partial \boldsymbol{z}^{L,l}}{\partial \text{vec}(S^{m,l})^T} \text{vec}\left(\text{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(Z^{m-1,l}) \\ \mathbb{1}^T_{a^m b^m} \end{bmatrix} \right) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial \boldsymbol{z}^{L,1}}{\partial \text{vec}(S^{m,1})^T} \boldsymbol{p}^{m,1} \\ \vdots \\ \frac{\partial \boldsymbol{z}^{L,l}}{\partial \text{vec}(S^{m,l})^T} \boldsymbol{p}^{m,l} \end{bmatrix},
\end{aligned}
$$

where

$$
\text{mat}(\boldsymbol{v}^m) \in R^{d^m \times (h^m h^m d^{m-1}+1)}
$$

doi:10.6342/NTU201802108

and

$$\begin{bmatrix} \boldsymbol{p}^{m,1} \\ \vdots \\ \boldsymbol{p}^{m,l} \end{bmatrix} = \text{vec}\left(\text{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(Z^{m-1,1}) & \cdots & \phi(Z^{m-1,l}) \\ \mathbb{1}^T_{a^m b^m} & \cdots & \mathbb{1}^T_{a^m b^m} \end{bmatrix} \right). \tag{4.29}$$

A *MATLAB* implementation for (4.28) is shown in Listing IV.7. We store the model in a structure of different layers. Thus in lines 11 and 13 we respectively obtain

$$Z^{m-1,i} \text{ (after padding) and } \frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T},\ i = 1, \ldots, l.$$

In line 14, we calculate (4.29) to derive $\boldsymbol{p}^{m,1}, \ldots, \boldsymbol{p}^{m,l}$. In line 15, we separately calculate

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T} \boldsymbol{p}^{m,i},\ i = 1, \ldots, l.$$

In line 16, we sum of the results of each layer.

After deriving (4.28), from (3.48) and (4.27), we must calculate

$$\begin{bmatrix} \boldsymbol{q}^1 \\ \vdots \\ \boldsymbol{q}^l \end{bmatrix} = \begin{bmatrix} B^1 \sum_{m=1}^{L} J^{m,1} \boldsymbol{v}^m \\ \vdots \\ B^l \sum_{m=1}^{L} J^{m,l} \boldsymbol{v}^m \end{bmatrix}. \tag{4.30}$$

From (3.49), (4.30) can be derived by multiplying every element of (4.28) by two.

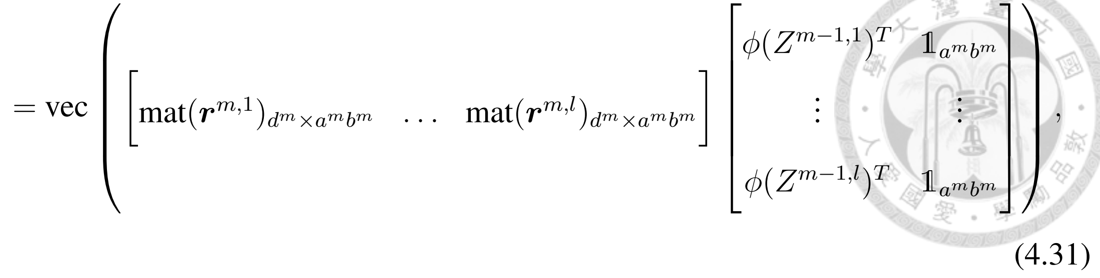
When (4.30) is available, from (4.27), the following matrix-vector product is needed.

$$\begin{bmatrix} (J^{1,1})^T & \ldots & (J^{1,l})^T \\ \vdots & \ddots & \vdots \\ (J^{L,1})^T & \ldots & (J^{L,l})^T \end{bmatrix} \begin{bmatrix} \boldsymbol{q}^1 \\ \vdots \\ \boldsymbol{q}^l \end{bmatrix}$$

For the layer m, from (3.50), the calculation is written as

$$\sum_{i=1}^{l} \text{vec}\left(\text{mat}\left(\left(\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T} \right)^T \boldsymbol{q}^i \right)_{d^m \times a^m b^m} \left[\phi(Z^{m-1,i})^T \ \mathbb{1}_{a^m b^m} \right] \right)$$

doi:10.6342/NTU201802108



$$= \text{vec}\left(\begin{bmatrix} \text{mat}(\boldsymbol{r}^{m,1})_{d^m \times a^m b^m} & \dots & \text{mat}(\boldsymbol{r}^{m,l})_{d^m \times a^m b^m} \end{bmatrix} \begin{bmatrix} \phi(Z^{m-1,1})^T & \mathbb{1}_{a^m b^m} \\ \vdots & \vdots \\ \phi(Z^{m-1,l})^T & \mathbb{1}_{a^m b^m} \end{bmatrix} \right), \tag{4.31}$$

where

$$\begin{bmatrix} \boldsymbol{r}^{m,1} \\ \vdots \\ \boldsymbol{r}^{m,l} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \boldsymbol{z}^{L,1}}{\partial \text{vec}(S^{m,1})^T} \right)^T \boldsymbol{q}^1 \\ \vdots \\ \left(\frac{\partial \boldsymbol{z}^{L,l}}{\partial \text{vec}(S^{m,l})^T} \right)^T \boldsymbol{q}^l \end{bmatrix}. \tag{4.32}$$

A *MATLAB* implementation for (4.31) is shown in Listing IV.8. In line 12, we separately calculate $\boldsymbol{r}^{m,1}, \dots, \boldsymbol{r}^{m,l}$ by (4.32). In line 18, we build

$$\begin{bmatrix} \phi(Z^{m-1,1})^T & \mathbb{1}_{a^m b^m} \\ \vdots & \vdots \\ \phi(Z^{m-1,l})^T & \mathbb{1}_{a^m b^m} \end{bmatrix}.$$

In line 19, we calculate (4.31). In line 20, we derive the result of (4.27) for the mth layer.

4.6 Mini-Batch Function and Gradient Evaluation

Later in Chapter 5.1 we will discuss details of memory usage. One important conclusion is that in many places of the Newton method, the memory consumption is proportional to the number of data. This fact causes difficulties in handling large data sets. Therefore, here we discuss some implementation techniques to address the memory difficulty.

In subsampled Newton methods discussed in Chapter 3.1, a subset S of the training data is used to derive the subsampled Gauss-Newton matrix for approximating the Hessian matrix. While a motivation of this technique is to trade a slightly less accurate

doi:10.6342/NTU201802108

direction for shorter running time per iteration, it is also useful to reduce the memory consumption. For example, at the mth convolutional layer, we only need to store the following matrices

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T},\ \forall i \in S \tag{4.33}$$

for the Gauss-Newton matrix-vector products.

However, function and gradient evaluations must use the whole training data. Fortunately, both operations involve the summation of independent results over all instances. Here we follow Wang et al. (2018a) to split the index set $\{1, \ldots, l\}$ of data to, for example, R equal-sized subsets $S_1, \ldots, S_R$. We then calculate the result corresponding to each subset and accumulate them for the final output. Take the function evaluation as an example. For each subset, we must store only

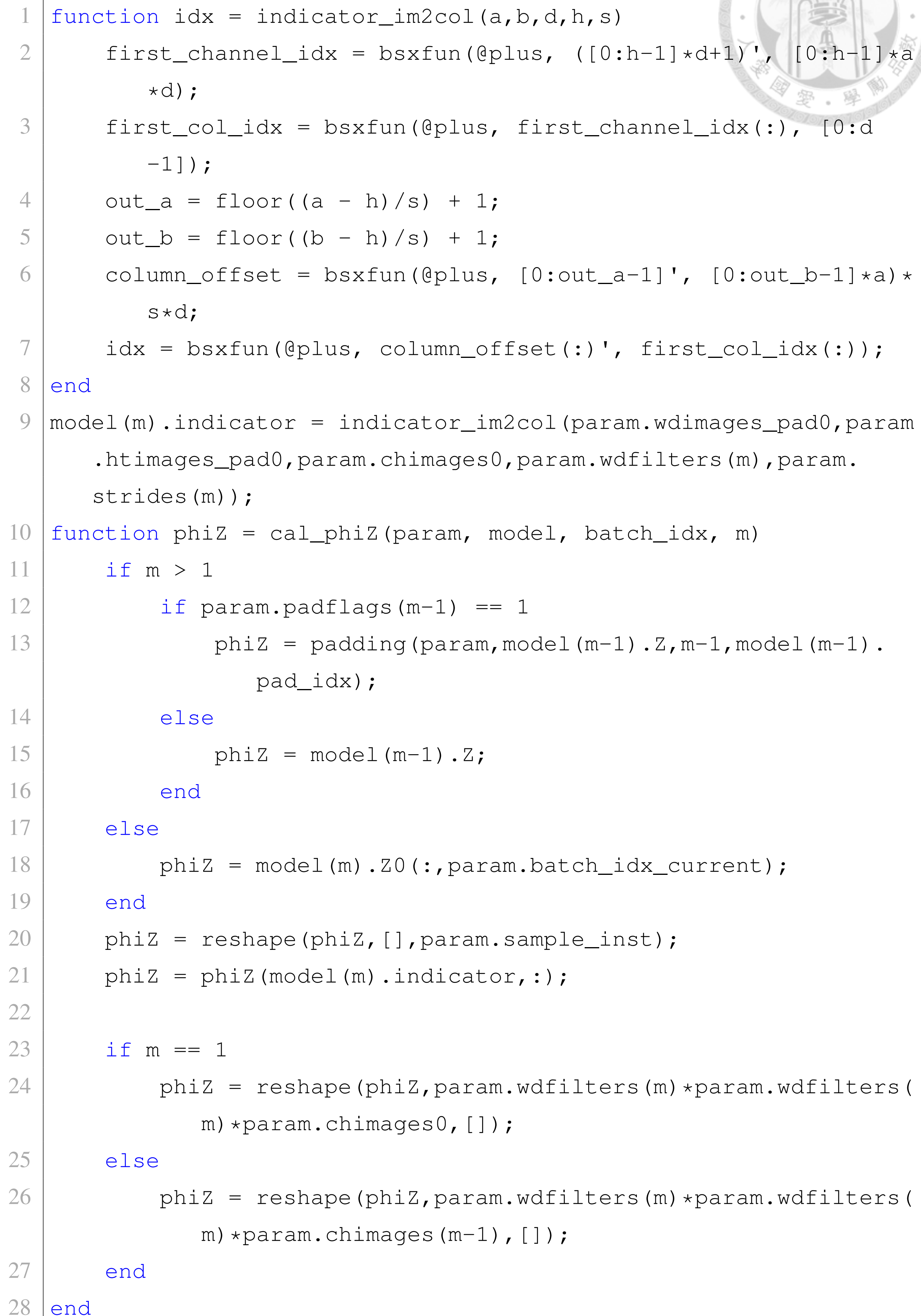
$$Z^{m,i},\ \forall m,\ \forall i \in S_r.$$

Thus, the memory usage can be dramatically reduced.

For the Gauss-Newton matrix-vector product, to calculate (4.33), we need $Z^{m,i}$, $\forall i \in S$. However, under the mini-batch setting, the needed values may not be kept. Our strategy is to let the last subset S_R be the same subset used for the sub-sampled Hessian. Then we can preserve the needed $Z^{m,i}$ for subsequent operations.

doi:10.6342/NTU201802108

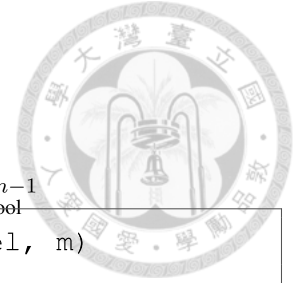
Listing IV.1: *MATLAB* implementation for $\phi(Z^{m-1,i})$



```
function idx = indicator_im2col(a,b,d,h,s)
    first_channel_idx = bsxfun(@plus, ([0:h-1]*d+1)', [0:h-1]*a*d);
    first_col_idx = bsxfun(@plus, first_channel_idx(:), [0:d-1]);
    out_a = floor((a - h)/s) + 1;
    out_b = floor((b - h)/s) + 1;
    column_offset = bsxfun(@plus, [0:out_a-1]', [0:out_b-1]*a)*s*d;
    idx = bsxfun(@plus, column_offset(:)', first_col_idx(:));
end
model(m).indicator = indicator_im2col(param.wdimages_pad0,param.htimages_pad0,param.chimages0,param.wdfilters(m),param.strides(m));
function phiZ = cal_phiZ(param, model, batch_idx, m)
    if m > 1
        if param.padflags(m-1) == 1
            phiZ = padding(param,model(m-1).Z,m-1,model(m-1).pad_idx);
        else
            phiZ = model(m-1).Z;
        end
    else
        phiZ = model(m).Z0(:,param.batch_idx_current);
    end
    phiZ = reshape(phiZ,[],param.sample_inst);
    phiZ = phiZ(model(m).indicator,:);

    if m == 1
        phiZ = reshape(phiZ,param.wdfilters(m)*param.wdfilters(m)*param.chimages0,[]);
    else
        phiZ = reshape(phiZ,param.wdfilters(m)*param.wdfilters(m)*param.chimages(m-1),[]);
    end
end
```


doi:10.6342/NTU201802108



Listing IV.2: *MATLAB* implementation for P_{pool}^{m-1}

```
function [param, model] = maxpooling(param, model, m)

a = param.htimages(m);
b = param.wdimages(m);
d = param.chimages(m);
h = param.wdpool(m);
S_k = param.num_sampled_data;
P = model(m).idx_phiZ_pool;
Z = model(m).Z;

rm_idx = [];
pool_idx = [1:d*a*b*S_k];
if (mod(a,h) > 0 || mod(b,h) > 0)
    newa = a - mod(a,h); newb = b - mod(b,h);
    remained_idx = bsxfun(@plus,[1:newa]',[0:newb-1]*a);
    remained_idx = bsxfun(@plus,remained_idx(:),[0:S_k-1]*a*b);
    Z = Z(:,remained_idx(:));

    pool_idx = reshape(pool_idx,d,[]);
    pool_idx = pool_idx(:,remained_idx(:));
end

Z = reshape(Z,[],S_k);
Z = Z(P,:);
[Z, WS] = max(reshape(Z,h*h,[]));
model(m).Z = reshape(Z,d,[]);

pool_idx = reshape(pool_idx,[],S_k);
pool_idx = pool_idx(P,:);
WS = WS + h*h*([0:floor(a/h)*floor(b/h)*d*S_k-1]);
model(m).pool_idx = pool_idx(WS);
```


doi:10.6342/NTU201802108

Listing IV.3: *MATLAB* implementation for evaluating (3.32)

```
function output = maxpooling_grad(param,m,input,pool_idx)

a = param.wdimages(m);
b = param.htimages(m);
d = param.chimages(m);
S_k = param.num_sampled_data;

output = zeros(d,a*b*S_k);
output(pool_idx) = reshape(input,[],1);
```

Listing IV.4: *MATLAB* implementation for the index of zero-padding

```
function [pad_idx] = padding_idx(param,m)

if m == 0
    a = param.wdimages0;
    b = param.htimages0;
else
    a = param.wdimages_pool(m);
    b = param.htimages_pool(m);
end

p = (param.wdfilters(m+1)-1)/2;
newa = a+2*p; newb = b+2*p;
pad_idx = repmat(p+(1:a)', b,1) + repeat_elements(newa*(p+(0:b
    -1)'), a);
```


doi:10.6342/NTU201802108

Listing IV.5: *MATLAB* implementation for zero-padding

```
1  function output = padding(param,Z,m,pad_idx_one)
2
3  a = param.wdimages_pad(m);
4  b = param.htimages_pad(m);
5  d = param.chimages(m);
6  S_k = param.num_sampled_data;
7
8  idx = reshape(bsxfun(@plus, pad_idx_one, [0:S_k-1]*a*b), [], 1)
       ;
9  output = zeros(d,a*b*S_k);
10 output(:,idx) = Z;
```

Listing IV.6: *MATLAB* implementation to evaluate $\boldsymbol{v}^T P_{\phi}^{m-1}$

```
1  function vTP = vTP(param, model, S_k, m, V)
2  % V: a matrix with #cols = S_k
3
4  a_prev = param.htimages(m-1);
5  b_prev = param.wdimages(m-1);
6  d_prev = param.chimages(m-1);
7
8  idx = reshape(bsxfun(@plus, model(m).idx_phiZ(:), [0:S_k-1]*
       d_prev*a_prev*b_prev), [], 1);
9  vTP = reshape(accumarray(idx, V(:), [d_prev*a_prev*b_prev*S_k
       1]), [], S_k)';
```


doi:10.6342/NTU201802108

Listing IV.7: *MATLAB* implementation for $J\boldsymbol{v}$

```
function Jv = Jv(param, model, v_in, subset_idx)

n = param.n;
nL = param.nL;
L = param.L;
S_k = param.num_sampled_data;
Jv = zeros(nL*S_k, 1);

for m = param.L : -1 : param.LC+1
    n_m = param.neurons(m+1);
    v = reshape(v_in(n(m)+1:n(m+1)), n_m, []);
    p = v * [model(m-1).Z; ones(1, S_k)];
    if m < L
        p = p';
        p = repeat_elements(p, nL);
        p = sum(model(m).dZLdS_T.*p, 2);
    else
        p = p(:);
    end
    Jv = Jv + p;
end

for m = param.LC : -1 : 1
    a = param.wdimages(m);
    b = param.htimages(m);
    d = param.chimages(m);
    v = reshape(v_in(n(m)+1:n(m+1)), d, []);
    phiZ = [cal_phiZ(param, model, subset_idx, m); ones(1, a*b*
       S_k)];
    p = reshape(v * phiZ, [], S_k);
    p = p';
    p = repeat_elements(p, nL);
    p = sum(model(m).dZLdS_T.*p, 2);
    Jv = Jv + p;
end
```


doi:10.6342/NTU201802108

Listing IV.8: *MATLAB* implementation for $J^T \boldsymbol{q}$

```
nL = param.nL;
S_K = param.sample_inst;
idx = param.batch_idx_current;
lambda = param.lambda;
C = param.C;
q = model(1).Jv;
for m = param.LC : -1 : 1
    ZsT = model(m).ZsT;
    Z = model(m-1).Z;
    d = param.chimages(m);
    v = cgparam(m).p;
    r = arrayfun(@(i) ZsT(1+(i-1)*nL:i*nL,:)' * q(1+(i-1)*nL:i*
       nL),[1:S_K],'un',0);
    r = horzcat(r{:});
    r = reshape(r,d,[]);
    if m > 1
        Z_pad = padding(param,Z,m-1,model(m-1).pad_idx);
    end
    phiZ = cal_phiZ(param,model,idx,m,Z_pad);
    r = reshape(r*[phiZ' ones(size(phiZ,2),1)],[],1);
    model(m).Gv = (lambda + 1/C)*v + r/S_K;
end
```


doi:10.6342/NTU201802108

CHAPTER V

Analysis of Newton Methods for CNN

In this chapter, based on the implementation details in Chapter IV, we analyze the memory and computational cost per iteration. We consider that all training instances are used. If the subsampled Hessian in Chapter III is considered, then in the Jacobian calculation and the Gauss-Newton matrix vector products, the number of instances l should be replaced by the subset size $|S|$.

In this discussion we exclude the padding operation and the pooling layer because first they are optional steps and second they are not the bottleneck.

5.1 Memory Requirement

(1) Weight matrix and bias vector: For every layer, we must store

$$W^m \text{ and } \boldsymbol{b}^m,\ m = 1, \dots, L.$$

From (2.10) and (2.20), the memory usage is

$$\mathcal{O}\left(\sum_{m=1}^{L^c} d^m \times (h^m h^m d^{m-1} + 1) + \sum_{m=L^c+1}^{L} n_m \times (n_{m-1} + 1)\right).$$

(2) To construct $\phi(Z^{m-1,i})$, in Chapter 4.1, we store each position's corresponding

doi:10.6342/NTU201802108

linear index in $Z^{m-1,i}$.

$$\mathcal{O}\left(\sum_{m=1}^{L^c} h^m h^m d^{m-1} a^m b^m\right).$$

(3) Function evaluation: From Chapter II, we store

$$Z^{m,i},\ m = 0, \ldots, L,\ i = 1, \ldots, l.$$

Therefore, the memory usage is

$$\mathcal{O}\left(l \times \left(\sum_{m=0}^{L^c} d^m a^m b^m + \sum_{m=L^c+1}^{L} n_m\right)\right).$$

(4) Gradient evaluation: From Chapter 3.2, because

$$\frac{\partial \xi_i}{\partial \text{vec}(S^{m-1,i})^T},\ m = 2, \ldots, L,\ \forall i.$$

is only used in backward process. We just store this matrix for two adjacent layers. Therefore, the memory usage is

$$\mathcal{O}\left(l \times \sum_{\{m,m+1\}} d^m a^m b^m\right),\ 1 \leq m < L^c$$

in convolutional layers or

$$\mathcal{O}\left(l \times \sum_{m \in \{m,m+1\}} n^m\right),\ L^c < m < L$$

in fully-connected layers. The following matrix must be stored.

$$\frac{\partial \xi_i}{\partial \text{vec}(W^m)^T},\ \text{and}\ \frac{\partial \xi_i}{\partial (\boldsymbol{b}^m)^T},\ m = 1, \ldots, L,\ \forall i.$$

Therefore, the memory usage is

$$\mathcal{O}\left(l \times \left(\sum_{m=1}^{L^c} d^m \left(h^m h^m d^{m-1} + 1\right) + \sum_{m=L^c+1}^{L} n_m \left(n_{m-1} + 1\right)\right)\right).$$

doi:10.6342/NTU201802108

(5) Jacobian evaluation and Gauss-Newton matrix-vector products: Besides W^m and $Z^{m-1,i}$, from (3.36), (3.47), (3.50), we explicitly store

$$\frac{\partial \boldsymbol{z}^{L,i}}{\partial \text{vec}(S^{m,i})^T},\ m = 1, \ldots, L,\ \forall i.$$

Thus, the memory usage is

$$\mathcal{O}\left(l \times n_L \times \left(\sum_{m=1}^{L^c}(d^m a^m b^m) + \sum_{m=L^c+1}^{L} n_m\right)\right).$$

5.2 Computational Cost

To avoid clutter, we show the computational cost for mth conolutional/fully-connected layer.

(1) Function evaluation:

- Convolutional layers: From (2.8), (2.11), and (2.12), the computational cost is

$$\mathcal{O}(l \times h^m h^m d^{m-1} d^m a^m b^m).$$

- Fully-connected layers: From (2.21) and (2.22), the computational cost is

$$\mathcal{O}(l \times n_m n_{m-1})$$

(2) Gradient evaluation:

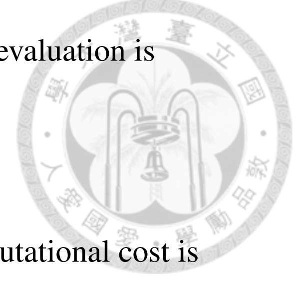
- Convolutional layers: From (3.22) and (3.23), the computational cost is

$$\mathcal{O}(l \times h^m h^m d^{m-1} d^m a^m b^m).$$

  From (3.25) and (3.26), the computational cost is

$$\mathcal{O}(l \times a^{m-1} b^{m-1} d^{m-1} d^m a^m b^m).$$

doi:10.6342/NTU201802108



Therefore, the total computational cost for the gradient evaluation is

$$\mathcal{O}(l \times a^{m-1}b^{m-1}d^{m-1}d^m a^m b^m).$$

- Fully-connected layers: For (3.27) and (3.28), the computational cost is

$$\mathcal{O}(l \times n_m n_{m-1}).$$

For (3.29) and (3.30), the computational cost is similar. Therefore, the total computational cost is

$$\mathcal{O}(l \times n_m n_{m-1}).$$

(3) Jacobian evaluation:

- Convolutional layers: From (3.35), the computational cost is

$$\mathcal{O}\left(l \times n_L \times d^m a^m b^m (h^m h^m d^{m-1} + 1)\right).$$

From (3.36), the computational cost is

$$\mathcal{O}\left(l \times n_L \times (d^m a^m b^m h^m h^m d^{m-1} + h^m h^m a^{m-1} b^{m-1} d^{m-1})\right).$$

The computational cost of (3.37) can be omitted. Therefore, the total computational cost is

$$\mathcal{O}(l \times n_L \times d^m a^m b^m h^m h^m d^{m-1}).$$

- Fully-connected layers: From (3.40) and (3.42), the computational cost is

$$\mathcal{O}(l \times n_L \times n_m n_{m-1}).$$

(4) Gauss-Newton matrix-vector products:

- Convolutional layers: From (3.47) and (3.50), the computational cost is

$$\mathcal{O}\left(l \times (d^m h^m h^m d^{m-1} a^m b^m + n_L d^m a^m b^m)\right).$$

doi:10.6342/NTU201802108

- Fully-connected layers: From (3.51) and (3.52), the computational cost is

$$\mathcal{O}\left(l \times \left(n_m n_{m-1} + n_L n_m\right)\right).$$

doi:10.6342/NTU201802108

CHAPTER VI

Experiments

We choose the following image data sets for experiments. All the data sets are publicly available[1] and the summary is in Table 6.1.

- MNIST: This data set, containing hand-written digits, is a widely used benchmark for data classification (LeCun et al., 1998).
- SVHN: This data set consists of the colored images of house numbers (Netzer et al., 2011).
- CIFAR10: This data set is a famous colored image classification benchmark (Krizhevsky and Hinton, 2009).
- smallNORB: This data set is built for 3D object recognition (LeCun et al., 2004). The original dimension is $96 \times 96 \times 2$ because every object is taken two 96×96 grayscale images from the different angles. These two images are then placed in two channels. For the dimensionality reduction, we downsample each channel of every object with the max pooling ($h = 3, s = 3$) to the dimension 32×32.

All the data sets were pre-processed by the following procedure.

[1] All data sets used can be found at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

doi:10.6342/NTU201802108

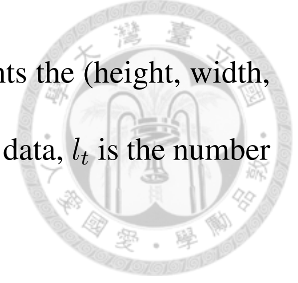


Table 6.1: Summary of the data sets, where $a^0 \times b^0 \times d^0$ represents the (height, width, channel) of the input image, l is the number of training data, l_t is the number of test data, and n_L is the number of classes.

Data set	$a^0 \times b^0 \times d^0$	l	l_t	n_L
MNIST	$28 \times 28 \times 1$	$60,000$	$10,000$	10
SVHN	$32 \times 32 \times 3$	$73,257$	$26,032$	10
CIFAR10	$32 \times 32 \times 3$	$50,000$	$10,000$	10
smallNORB	$32 \times 32 \times 2$	$24,300$	$24,300$	5

(1) Min-max normalization. That is, for every image $Z^{0,i}$, we have

$$Z^{0,i} \leftarrow \frac{Z^{0,i} - \min}{\max - \min},$$

where $\max/\min$ is the maximum/minimum value in $Z^{0,i}$.

(2) Zero-centering. This is commonly applied before training CNN (Krizhevsky et al., 2012; Zeiler and Fergus, 2014). That is, for every image $Z^{0,i}$, we have

$$Z^{0,i} \leftarrow Z^{0,i} - \text{mean}(Z^{0,i}),$$

where mean$(Z^{0,i})$ is the mean value in $Z^{0,i}$.

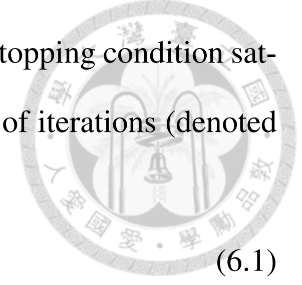
We consider two simple CNN structure shown in Table 6.2. The parameters used in our algorithm are given as follows. For the initialization, we follow He et al. (2015) to randomly set the weight values from the $\mathcal{N}(0,1)$ distribution and multiply by

$$\sqrt{\frac{2}{n_{\text{in}}^m}},$$

where

$$n_{\text{in}}^m = \begin{cases} d^{m-1} \times a^{m-1} \times b^{m-1} & \text{if } m \leq L^c, \\ n_{m-1} & \text{otherwise.} \end{cases}$$

doi:10.6342/NTU201802108



For a CG procedure, we terminate it when the following relative stopping condition satisfies or the number of CG iterations reaches a maximal number of iterations (denoted as CG_{max}).

$$||(G + \lambda\mathcal{I})\boldsymbol{d} + \nabla f(\boldsymbol{\theta})|| \leq \sigma||\nabla f(\boldsymbol{\theta})||, \quad (6.1)$$

where $\sigma = 0.1$ and $\text{CG}_{\text{max}} = 250$. For the implementation of the Levenberg-Marquardt method, we set the initial $\lambda_1 = 1$ and (drop, boost, ρ_{upper}, ρ_{lower}) constants in (3.10) are ($2/3$, $3/2$, 0.75, 0.1). In addition, the sampling rate for the Gauss-Newton matrix is set to 1% and the value of C in (2.26) is set to $0.01l$.

6.1 Comparison Between Newton and Stochastic Gradient Methods

In this chapter, the goal is to compare SG methods with the proposed subsampled Newton method for CNN. For SG methods, we consider mini-batch SG with momentum. We use the python deep learning library, *Keras* (Chollet et al., 2015), to implement it. To have a fair comparison between SG and subsampled Newton methods, the following conditions are fixed.

- Initial points.
- Network structures.
- Objective function.
- Regularization parameter.

The training mini-batch size is 128 for all SG experiments. The initial learning rate is selected from $\{0.003, 0.001, 0.0003, 0.0001\}$ by five-fold cross validation.[2] When

[2] We split the training data by stratified sampling for the cross validation.

doi:10.6342/NTU201802108

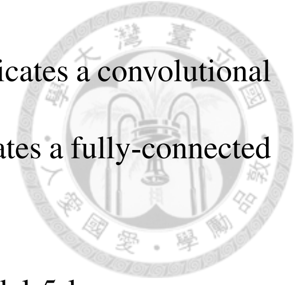


Table 6.2: Structure of convolutional neural networks. "conv" indicates a convolutional layer, "pool" indicates a pooling layer, and "full" indicates a fully-connected layer.

	model-3-layers			model-5-layers		
	filter size	#filters	stride	filter size	#filters	stride
conv 1	$5 \times 5 \times 3$	32	1	$5 \times 5 \times 3$	32	1
pool 1	2×2	-	2	2×2	-	2
conv 2	$3 \times 3 \times 32$	64	1	$3 \times 3 \times 32$	32	1
pool 2	2×2	-	2	-	-	-
conv 3	$3 \times 3 \times 32$	64	1	$3 \times 3 \times 32$	64	1
pool 3	2×2	-	2	2×2	-	2
conv 4	-	-	-	$3 \times 3 \times 64$	64	1
pool 4	-	-	-	-	-	-
conv 5	-	-	-	$3 \times 3 \times 64$	128	1
pool 5	-	-	-	2×2	-	2

conducting the cross validation and training process, the learning rate is adapted to the *Keras* framework's default scheduling with a decaying factor 10^{-6} and the momentum coefficient is 0.9.

From the results shown in Table 6.3, we can see that

doi:10.6342/NTU201802108

Table 6.3: Test accuracy for Newton method and SG. For Newton method, we trained for 250 iterations; for SG, we trained for 1000 epochs.

	model-3-layers		model-5-layers	
	Newton	SG	*Newton*	SG
MNIST	$(99.15, 99.25)\%$	99.15%	99.46%	99.35%
SVHN	$(92.91, 92.99)\%$	93.21%	93.49%	94.60%
CIFAR10	$(77.85, 79.41)\%$	79.27%	76.7%	79.47%
smallNORB	$(98.14, 98.16)\%$	98.09%	97.68%	98.00%

doi:10.6342/NTU201802108

CHAPTER VII

Conclusions

In this study, we establish all the building blocks of Newton methods for CNN. A simple and elegant *MATLAB* implementation is developed for public use. Based on our results, it is possible to develop novel techniques to further enhance Newton methods for CNN.

doi:10.6342/NTU201802108

APPENDICES

doi:10.6342/NTU201802108

APPENDIX A. List of Symbols

Notation	Description
$\boldsymbol{y}^i$	The label vector of the ith training instance.
$Z^{0,i}$	The input image of the ith training instance.
l	The number of training instances.
K	The number of classes.
$\boldsymbol{\theta}$	The model vector (weights and biases) of the neural network.
ξ	The loss function.
ξ_i	The training loss of the ith instance.
f	The objective function.
C	The regularization parameter.
L	The number of layers of the neural network.
L^c	The number of convolutional layers of the neural network.
L^f	The number of fully-connected layers of the neural network.
n_m	The number of neurons in the mth layer ($L^c < m \leq L$).
n	The total number of weights and biases.
a^m	the height of the data at the mth layer ($0 \leq m \leq L^c$).
b^m	the width of the data at the mth layer ($0 \leq m \leq L^c$).
d^m	the depth (or the number of channels) of the data at the mth layer ($0 \leq m \leq L^c$).
h^m	the height (width) of the filters at the mth layer.
W^m	The weight matrix in the mth layer.
$\boldsymbol{b}^m$	The bias vector in the mth layer.

doi:10.6342/NTU201802108

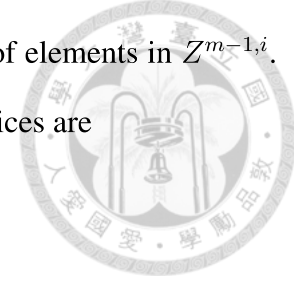
Notation	Description
$S^{m,i}$	The output matrix of the function $(W^m)^T\phi(Z^{m-1,i})+\boldsymbol{b}^m\mathbb{1}^T_{a^mb^m}$ in the mth layer for the ith instance ($1 \leq m \leq L^c$).
$Z^{m,i}$	The output matrix (element-wise application of the activation function on $S^{m,i}$) in the mth layer for the ith instance ($1 \leq m \leq L^c$).
$\boldsymbol{s}^{m,i}$	The output vector of the function $(W^m)^T\boldsymbol{z}^{m-1,i}+\boldsymbol{b}^m$ in the mth layer for the ith instance ($L^c < m \leq L$).
$\boldsymbol{z}^{m,i}$	The output vector (element-wise application of the activation function on $\boldsymbol{s}^{m,i}$) in the mth layer for the ith instance ($L^c < m \leq L$).
σ	The activation function.
J^i	The Jacobian matrix of $\boldsymbol{z}^{L,i}$ with respect to $\boldsymbol{\theta}$.
$\mathcal{I}$	An identity matrix.
α_k	A step size at the kth iteration.
ρ_k	The ratio between the actual function reduction and the predicted reduction at the kth iteration.
λ_k	A parameter in the Levenberg-Marquardt method.
$\mathcal{N}(\mu, \sigma^2)$	A Gaussian distribution with mean μ and variance σ^2.

APPENDIX B. Alternative Method for the Generation of $\phi(Z^{m-1,i})$

For the alternative method here, we use *MATLAB*'s `im2col` with $s^m = 1$ and extract a sub-matrix as $\phi(Z^{m-1,i})$.

We now explain each line of the program. To find P_ϕ^{m-1}, from (2.9) what we need is to extract elements in $Z^{m-1,i}$. Some elements may be extracted multiple times. For the

doi:10.6342/NTU201802108



extraction it is more convenient to chapter on the linear indices of elements in $Z^{m-1,i}$. Following *MATLAB*'s setting, for an $a \times b$ matrix, the linear indices are

$$[\,1,\ \ldots,\ a,\ a+1,\ \ldots,\ ab\,],$$

where elements are mapped to the above indices in a cloumn-wise setting. In line 2, we start with obtaining the linear indices of the first row of $Z^{m-1,i}$, which corresponds to the first channel of the image. In line 3, we use `im2col` to build $\phi(Z^{m-1,i})$ under $s^m = d^{m-1} = 1$, though contents of the input matrix are linear indices of $Z^{m-1,i}$ rather than values. For $\phi(Z^{m-1,i})$ under $s^m = d^{m-1} = 1$, the matrix size is

$$h^m h^m \times \bar{a}^m \bar{b}^m,$$

where from (2.4),

$$\bar{a}^m = a^{m-1} - h^m + 1,\ \bar{b}^m = b^{m-1} - h^m + 1.$$

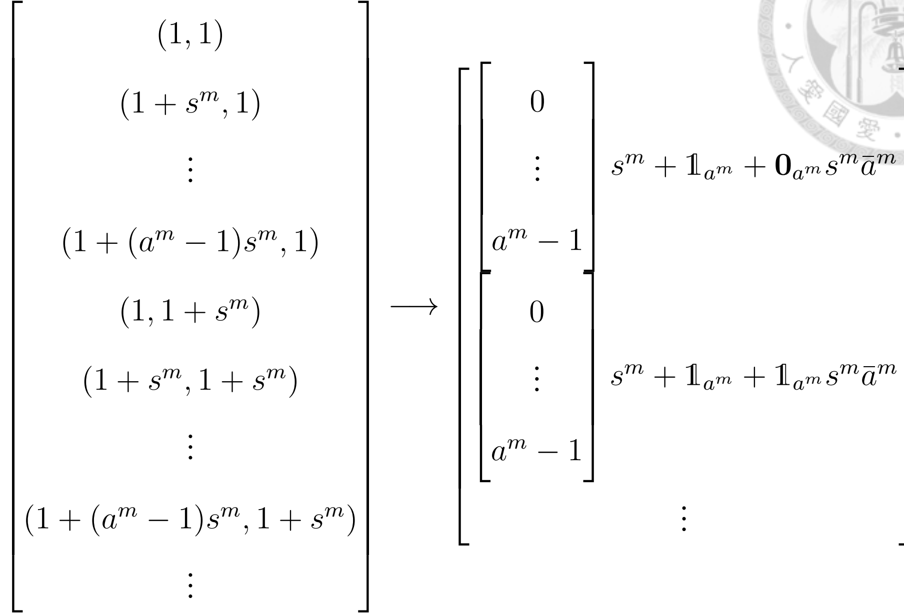
From (2.9), when a general s^m is considered, we must select some columns, whose column indices are the following subset of $\{1, \ldots, \bar{a}^m \bar{b}^m\}$:

$$\mathbb{1}_{b^m} \otimes \left(\begin{bmatrix} 0 \\ \vdots \\ a^m - 1 \end{bmatrix} s^m + \mathbb{1}_{a^m} \right) + \left(\begin{bmatrix} 0 \\ \vdots \\ b^m - 1 \end{bmatrix} s^m \right) \otimes \begin{bmatrix} \bar{a}^m \\ \vdots \\ \bar{a}^m \end{bmatrix}_{a^m \times 1}, \tag{B.1}$$

where a^m and b^m are defined in (2.4). More precisely, (B.1) comes from the following

doi:10.6342/NTU201802108

mapping between the first row of $\phi(Z^{m-1,i})$ in (2.9) and $\{1, \ldots, \bar{a}^m \bar{b}^m\}$:



$$\begin{bmatrix} (1,1) \\ (1+s^m,1) \\ \vdots \\ (1+(a^m-1)s^m,1) \\ (1,1+s^m) \\ (1+s^m,1+s^m) \\ \vdots \\ (1+(a^m-1)s^m,1+s^m) \\ \vdots \end{bmatrix} \longrightarrow \begin{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ a^m-1 \end{bmatrix} s^m + \mathbb{1}_{a^m} + \mathbf{0}_{a^m} s^m \bar{a}^m \\ \begin{bmatrix} 0 \\ \vdots \\ a^m-1 \end{bmatrix} s^m + \mathbb{1}_{a^m} + \mathbb{1}_{a^m} s^m \bar{a}^m \\ \vdots \end{bmatrix}$$

Next we discuss how to extend the linear indices of the first channel to others. From (2.5), each column of $Z^{m-1,i}$ contains values of the same pixel in different channels. Therefore, because we consider a column-major order, indices in $Z^{m-1,i}$ for a given pixel are a continuous segment. Then in (2.9) for $\phi(Z^{m-1,i})$, essentially we have d^{m-1} segments ordered vertically and elements in two consecutive segments are from two consecutive rows in $Z^{m-1,i}$. Therefore, the following index matrix can be used to extract all needed elements in $Z^{m-1,i}$ for $\phi(Z^{m-1,i})$.

$$\begin{bmatrix} \text{linear indices of } Z^{m-1,i} \text{ for} \\ \text{1st channel of } \phi(Z^{m-1,i}) \end{bmatrix}_{h^m h^m \times a^m b^m} \otimes \mathbb{1}_{d^{m-1}} + \left(\begin{bmatrix} 0 \\ \vdots \\ d^{m-1}-1 \end{bmatrix} \otimes \mathbb{1}_{h^m h^m} \right) \otimes \mathbb{1}^T_{a^m b^m}. \tag{B.2}$$

The implementation is in line 10 and we use a property of *MATLAB* to add a matrix and a vector. Thus the $\otimes$ operation in the second term in (B.2) is not needed.

doi:10.6342/NTU201802108

Listing B.1: *Matlab* implementation for P_{ϕ}^{m-1}

```
1  function indicator = indicator_im2col(a,b,d,h,s)
2      input_idx = reshape(([1:a*b]-1)*d+1,a,b);
3      output_idx = im2col(input_idx,[h,h],'sliding');
4      a_bar = a - h + 1;
5      b_bar = b - h + 1;
6      a_idx = 1:s:a_bar;
7      b_idx = 1:s:b_bar;
8      select_idx = repelem(a_idx,1,length(a_idx)) + a_bar*repmat(
          b_idx-1,1,length(b_idx));
9      output_idx = output_idx(:,select_idx);
10     output_idx = repmat(output_idx,d,1) + repelem([0:d-1]',h*h
          ,1);
11 end
```


doi:10.6342/NTU201802108

BIBLIOGRAPHY

A. Botev, H. Ritter, and D. Barber. Practical gauss-newton optimisation for deep learning. In *Procceedings of International Conference on Machine Learning (ICML)*, pages 557–565, 2017.

R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal. On the use of stochastic Hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

F. Chollet et al. Keras. `https://keras.io`, 2015.

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1): 1–17, 1990.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2015.

X. He, D. Mudigere, M. Smelyanskiy, and M. Takáč. Large scale distributed Hessian-free optimization for deep neural network, 2016. arXiv preprint arXiv:1606.00511.

doi:10.6342/NTU201802108

R. Kiros. Training neural networks with stochastic Hessian-free optimization, 2013. arXiv preprint arXiv:1301.3641.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.

Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 265–272, 2011.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. MNIST database available at `http://yann.lecun.com/exdb/mnist/`.

Y. LeCun, F. J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 97–104, 2004.

K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

doi:10.6342/NTU201802108

C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region Newton method for large-scale logistic regression. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, 2007. Software available at http://www.csie.ntu.edu.tw/~cjlin/liblinear.

D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

J. Martens. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010.

Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

N. N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, 2002.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. arXiv preprint arXiv:1409.1556.

A. Vedaldi and K. Lenc. MatConvNet: Convolutional neural networks for matlab. In *Proceedings of the 23rd ACM International Conference on Multimedia*, pages 689–692, 2015.

O. Vinyals and D. Povey. Krylov subspace descent for deep learning. In *Proceedings of Artificial Intelligence and Statistics*, pages 1261–1268, 2012.

doi:10.6342/NTU201802108

C.-C. Wang, C.-H. Huang, and C.-J. Lin. Subsampled Hessian Newton methods for supervised learning. *Neural Computation*, 27:1766–1795, 2015. URL http://www.csie.ntu.edu.tw/~cjlin/papers/sub_hessian/sample_hessian.pdf.

C.-C. Wang, K.-L. Tan, C.-T. Chen, Y.-H. Lin, S. S. Keerthi, D. Mahajan, S. Sundararajan, and C.-J. Lin. Distributed Newton methods for deep learning. *Neural Computation*, 30:1673–1724, 2018a. URL http://www.csie.ntu.edu.tw/~cjlin/papers/dnn/dsh.pdf.

C.-C. Wang, K. L. Tan, and C. J. Lin. Newton methods for convolutional neural networks. Technical report, National Taiwan University, 2018b.

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Proceedings of European Conference on Computer Vision*, pages 818–833, 2014.

doi:10.6342/NTU201802108