Newton Methods for Convolutional Neural Networks

CHIEN-CHIH WANG, Rakuten Institute of Technology
KENT LOONG TAN, National Taiwan University
CHIH-JEN LIN, National Taiwan University

Deep learning involves a difficult non-convex optimization problem, which is often solved by stochastic gradient (SG) methods. While SG is usually effective, it may not be robust in some situations. Recently, Newton methods have been investigated as an alternative optimization technique, but most existing studies consider only fully-connected feedforward neural networks. These studies do not investigate some more commonly used networks such as Convolutional Neural Networks (CNN). One reason is that Newton methods for CNN involve complicated operations, and so far no works have conducted a thorough investigation. In this work, we give details of all building blocks including the evaluation of function, gradient, Jacobian, and Gauss-Newton matrix-vector products. These basic components are very important not only for practical implementation but also for developing variants of Newton methods for CNN. We show that an efficient *MATLAB* implementation can be done in just several hundred lines of code. Preliminary experiments indicate that Newton methods are less sensitive to parameters than the stochastic gradient approach.

CCS Concepts: • **Computing methodologies** → **Supervised learning by classification**; **Neural networks**; • **Software and its engineering** → **Software libraries and repositories**.

Additional Key Words and Phrases: Convolution Neural Networks, Newton methods, Large-scale classification, Subsampled Hessian

1 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., [11, 25]). 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 work, 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., [1, 7, 9, 20, 28–30]). 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 have shown details of all the building blocks including the evaluation of function, gradient, Jacobian, and Hessian-vector products. In particular, because interpreter-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 work, 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

Authors' addresses: Chien-Chih Wang, Rakuten Institute of Technology, d98922007@csie.ntu.edu.tw; Kent Loong Tan, Department of Computer Science and Information Engineering, National Taiwan University, r04944005@csie.ntu.edu.tw; Chih-Jen Lin, cjlin@csie.ntu.edu.tw, Department of Computer Science and Information Engineering, National Taiwan University, cjlin@csie.ntu.edu.tw.

of Newton methods for fully-connected networks can be even tried. Our work will enable many further developments of Newton methods for CNN and maybe even other types of networks.

In experiments, we compare SG and Newton methods for CNN in several aspects. Results show that for SG under suitable parameters, it converges faster than Newton. However, SG is more sensitive to parameters and requires a careful parameter-selection procedure. Therefore, if we take the cross-validation procedure into consideration, the overall running time of Newton methods may be competitive.

This paper is organized as follows. In Section 2, we introduce CNN. In Section 3, Newton methods for CNN are investigated and the detailed mathematical formulations of all operations are derived. In Section 4, we discuss related works of Newton methods for training neural networks. The analysis of memory usage and computational complexity is in Section 5. Preliminary experiments to demonstrate the viability of Newton methods for CNN are in Section 6. Section 7 concludes this work.

A simple and efficient *MATLAB* implementation in just a few hundred lines of code is available at

https://www.github.com/cjlin1/simpleNN

Programs used for experiments in this paper and supplementary materials (including a list of symbols and implementation details) can be found at

https://www.csie.ntu.edu.tw/~cjlin/simpleNN

2 OPTIMIZATION PROBLEM OF CONVOLUTIONAL NEURAL NETWORKS

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

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

A CNN [13] utilizes a stack of convolutional layers followed by fully-connected layers to predict the target vector. Let L^c be the number of convolutional layers, and L^f be the number of fully-connected layers. The number of layers is

$$L = L^c + L^f.$$

Images

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

are input to the first layer, while the last (the Lth) layer outputs a predicted label vector

$$\hat{\boldsymbol{y}}^i,\ i = 1, \ldots, l.$$

A hallmark of CNN is that both input and output of convolutional layers are explicitly assumed to be images.

2.1 Convolutional Layer

In a convolutional layer, besides the main convolutional operations, two optional steps are padding and pooling, each of which can also be considered as a layer with input/output images. To easily describe all these operations in a unified setting, for the ith instance, we assume the input image of the current layer is

$$Z^{\text{in},i}$$

containing d^{in} channels of $a^{\text{in}} \times b^{\text{in}}$ images:

$$\begin{bmatrix} z^i_{1,1,1} & & z^i_{1,b^{\text{in}},1} \\ & \ddots & \\ z^i_{a^{\text{in}},1,1} & & z^i_{a^{\text{in}},b^{\text{in}},1} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} z^i_{1,1,d^{\text{in}}} & & z^i_{1,b^{\text{in}},d^{\text{in}}} \\ & \ddots & \\ z^i_{a^{\text{in}},1,d^{\text{in}}} & & z^i_{a^{\text{in}},b^{\text{in}},d^{\text{in}}} \end{bmatrix}. \tag{1}$$

The goal is to generate an output image

$$Z^{\text{out},i}$$

of d^{out} channels of $a^{\text{out}} \times b^{\text{out}}$ images.

Now we describe details of convolutional operations. To generate the output, we consider d^{out} filters, each of which is a 3-D weight matrix of size

$$h \times h \times d^{\text{in}}.$$

Specifically, the jth filter includes the following matrices

$$\begin{bmatrix} w^j_{1,1,1} & & w^j_{1,h,1} \\ & \ddots & \\ w^j_{h,1,1} & & w^j_{h,h,1} \end{bmatrix}, \quad \cdots \quad , \begin{bmatrix} w^j_{1,1,d^{\text{in}}} & & w^j_{1,h,d^{\text{in}}} \\ & \ddots & \\ w^j_{h,1,d^{\text{in}}} & & w^j_{h,h,d^{\text{in}}} \end{bmatrix}$$

and a bias term b_j.

The main idea of CNN is to extract local information by convolutional operations, each of which is the inner product between a small sub-image and a filter. For the jth filter, we scan the entire input image to obtain small regions of size (h, h) 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 sub-image of channel d is

$$\begin{bmatrix} z^i_{1,1,d} & \cdots & z^i_{1,h,d} \\ & \ddots & \\ z^i_{h,1,d} & \cdots & z^i_{h,h,d} \end{bmatrix}.$$

We then calculate the following value.

$$\sum_{d=1}^{d^{\text{in}}} \left\langle \begin{bmatrix} z^i_{1,1,d} & \cdots & z^i_{1,h,d} \\ & \ddots & \\ z^i_{h,1,d} & \cdots & z^i_{h,h,d} \end{bmatrix}, \begin{bmatrix} w^j_{1,1,d} & \cdots & w^j_{1,h,d} \\ & \ddots & \\ w^j_{h,1,d} & \cdots & w^j_{h,h,d} \end{bmatrix} \right\rangle + b_j, \tag{2}$$

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 for sliding the filter. That is, we move s pixels vertically or horizontally to get sub-images. For the $(2, 1)$ position of the output image, we move down s pixels vertically to obtain the following sub-image:

$$\begin{bmatrix} z^i_{1+s,1,d} & \cdots & z^i_{1+s,h,d} \\ & \ddots & \\ z^i_{h+s,1,d} & \cdots & z^i_{h+s,h,d} \end{bmatrix}.$$

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

$$\sum_{d=1}^{d^{\text{in}}} \left\langle \begin{bmatrix} z^i_{1+s,1,d} & \cdots & z^i_{1+s,h,d} \\ & \ddots & \\ z^i_{h+s,1,d} & \cdots & z^i_{h+s,h,d} \end{bmatrix}, \begin{bmatrix} w^j_{1,1,d} & \cdots & w^j_{1,h,d} \\ & \ddots & \\ w^j_{h,1,d} & \cdots & w^j_{h,h,d} \end{bmatrix} \right\rangle + b_j. \tag{3}$$

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

$$a^{\text{out}} = \lfloor \frac{a^{\text{in}} - h}{s} \rfloor + 1, \quad b^{\text{out}} = \lfloor \frac{b^{\text{in}} - h}{s} \rfloor + 1. \tag{4}$$

For efficient implementations, we can conduct all operations including (2) and (3) by matrix operations. To begin, we concatenate the matrices of the different channels in (1) to

$$Z^{\text{in},i} = \begin{bmatrix} z^i_{1,1,1} & \cdots & z^i_{a^{\text{in}},1,1} & z^i_{1,2,1} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ z^i_{1,1,d^{\text{in}}} & \cdots & z^i_{a^{\text{in}},1,d^{\text{in}}} & z^i_{1,2,d^{\text{in}}} & \cdots & z^i_{a^{\text{in}},b^{\text{in}},d^{\text{in}}} \end{bmatrix}, \ i = 1, \ldots, l. \tag{5}$$

We note that (2) is the inner product between the following two vectors

$$\begin{bmatrix} w^j_{1,1,1} & \cdots & w^j_{h,1,1} & w^j_{1,2,1} & \cdots & w^j_{h,h,1} & \cdots & w^j_{h,h,d^{\text{in}}} & b_j \end{bmatrix}^T$$

and

$$\begin{bmatrix} z^i_{1,1,1} & \cdots & z^i_{h,1,1} & z^i_{1,2,1} & \cdots & z^i_{h,h,1} & \cdots & z^i_{h,h,d^{\text{in}}} & 1 \end{bmatrix}^T.$$

Based on [27], we define the following two operators

$$\text{vec}(M) = \begin{bmatrix} M_{:,1} \\ \vdots \\ M_{:,b} \end{bmatrix} \in R^{ab \times 1}, \text{ where } M \in R^{a \times b}, \tag{6}$$

$$\text{mat}(\boldsymbol{v})_{a \times b} = \begin{bmatrix} v_1 & & v_{(b-1)a+1} \\ \vdots & \cdots & \vdots \\ v_a & & v_{ba} \end{bmatrix} \in R^{a \times b}, \text{ where } \boldsymbol{v} \in R^{ab \times 1}. \tag{7}$$

There exists a $0/1$ matrix

$$P_\phi \in R^{hhd^{\text{in}}a^{\text{out}}b^{\text{out}} \times d^{\text{in}}a^{\text{in}}b^{\text{in}}}$$

so that a linear operator

$$\phi : R^{d^{\text{in}} \times a^{\text{in}}b^{\text{in}}} \longrightarrow R^{hhd^{\text{in}} \times a^{\text{out}}b^{\text{out}}}$$

defined as

$$\phi(Z^{\text{in},i}) \equiv \text{mat}\left(P_\phi \text{vec}(Z^{\text{in},i})\right)_{hhd^{\text{in}} \times a^{\text{out}}b^{\text{out}}}, \ \forall i, \tag{8}$$

collects all sub-images in $Z^{\text{in},i}$. Specifically, $\phi(Z^{\text{in},i})$ is

$$\begin{bmatrix} z^i_{1,1,1} & \cdots & z^i_{1+(a^{\text{out}}-1)\times s,1,1} & z^i_{1,1+s,1} & \cdots & z^i_{1+(a^{\text{out}}-1)\times s,1+(b^{\text{out}}-1)\times s,1} \\ z^i_{2,1,1} & \cdots & z^i_{2+(a^{\text{out}}-1)\times s,1,1} & z^i_{2,1+s,1} & \cdots & z^i_{2+(a^{\text{out}}-1)\times s,1+(b^{\text{out}}-1)\times s,1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ z^i_{h,h,1} & \cdots & z^i_{h+(a^{\text{out}}-1)\times s,h,1} & z^i_{h,h+s,1} & \cdots & z^i_{h+(a^{\text{out}}-1)\times s,h+(b^{\text{out}}-1)\times s,1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ z^i_{1,1,d^{\text{in}}} & \cdots & z^i_{1+(a^{\text{out}}-1)\times s,1,d^{\text{in}}} & z^i_{1,1+s,d^{\text{in}}} & \cdots & z^i_{1+(a^{\text{out}}-1)\times s,1+(b^{\text{out}}-1)\times s,d^{\text{in}}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ z^i_{h,h,d^{\text{in}}} & \cdots & z^i_{h+(a^{\text{out}}-1)\times s,h,d^{\text{in}}} & z^i_{h,h+s,d^{\text{in}}} & \cdots & z^i_{h+(a^{\text{out}}-1)\times s,h+(b^{\text{out}}-1)\times s,d^{\text{in}}} \end{bmatrix} \in R^{hhd^{\text{in}}\times a^{\text{out}}b^{\text{out}}}. \tag{9}$$

By considering

$$W = \begin{bmatrix} w^1_{1,1,1} & w^1_{2,1,1} & \cdots & w^1_{h,h,d^{\text{in}}} \\ \vdots & \vdots & \ddots & \vdots \\ w^{d^{\text{out}}}_{1,1,1} & w^{d^{\text{out}}}_{2,1,1} & \cdots & w^{d^{\text{out}}}_{h,h,d^{\text{in}}} \end{bmatrix} \in R^{d^{\text{out}}\times hhd^{\text{in}}} \text{ and } \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_{d^{\text{out}}} \end{bmatrix} \in R^{d^{\text{out}}\times 1}, \tag{10}$$

all convolutional operations can be combined as

$$S^{\text{out},i} = W\phi(Z^{\text{in},i}) + \boldsymbol{b}\mathbb{1}^T_{a^{\text{out}}b^{\text{out}}} \in R^{d^{\text{out}}\times a^{\text{out}}b^{\text{out}}}, \tag{11}$$

where

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

Next, an activation function scales each element of $S^{\text{out},i}$ to obtain the output matrix $Z^{\text{out},i}$.

$$Z^{\text{out},i} = \sigma(S^{\text{out},i}) \in R^{d^{\text{out}}\times a^{\text{out}}b^{\text{out}}}. \tag{12}$$

For CNN, commonly the following RELU activation function

$$\sigma(x) = \max(x, 0) \tag{13}$$

is used.[1]

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

$$d^{\text{in}} \times a^{\text{in}}b^{\text{in}}$$

in (1) to

$$(hhd^{\text{in}}) \times a^{\text{out}}b^{\text{out}}$$

in (9). From (4), roughly

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

folds increase of the memory occurs. However, we gain efficiency by using fast matrix-matrix multiplications in optimized BLAS [4].

[1] To use Newton methods, $\sigma(x)$ should be twice differentiable, but the RELU function is not. For simplicity, we follow [11] to assume $\sigma'(x) = 1$ if $x > 0$ and 0 otherwise. It is possible to use a differentiable approximation of the RELU function, though we leave this issue for future investigation.

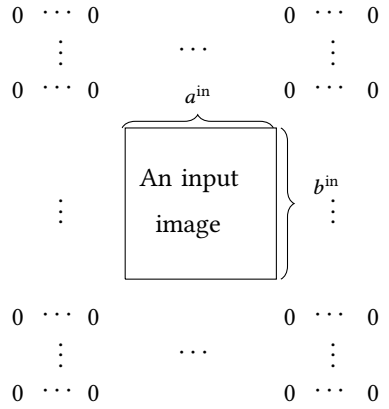

Fig. 1. An illustration of the padding operation to have zeros around the border.

$$\left[\begin{array}{cc|cc} 2 & 3 & 6 & 8 \\ 5 & 4 & 9 & 7 \\ \hline 1 & 2 & 6 & 0 \\ 4 & 3 & 2 & 1 \end{array}\right] \to \begin{bmatrix} 5 & 9 \\ 4 & 6 \end{bmatrix}$$

(a) Image A

$$\left[\begin{array}{cc|cc} 3 & 2 & 3 & 6 \\ 4 & 5 & 4 & 9 \\ \hline 2 & 1 & 2 & 6 \\ 3 & 4 & 3 & 2 \end{array}\right] \to \begin{bmatrix} 5 & 9 \\ 4 & 6 \end{bmatrix}$$

(b) Image B

Fig. 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.

2.1.1 Zero-padding. To better control the size of the output image, before the convolutional operation we may enlarge the input image to have zero values around the border. This technique is called zero-padding in CNN training. See an illustration in Figure 1.

To specify the mathematical operation we can treat the padding operation as a layer of mapping an input $Z^{\text{in},i}$ to an output $Z^{\text{out},i}$. Let

$$d^{\text{out}} = d^{\text{in}}.$$

There exists a $0/1$ matrix

$$P_{\text{pad}} \in R^{d^{\text{out}}a^{\text{out}}b^{\text{out}} \times d^{\text{in}}a^{\text{in}}b^{\text{in}}}$$

so that the padding operation can be represented by

$$Z^{\text{out},i} \equiv \text{mat}(P_{\text{pad}}\text{vec}(Z^{\text{in},i}))_{d^{\text{out}} \times a^{\text{out}}b^{\text{out}}}. \tag{14}$$

2.1.2 Pooling Operations. For CNN, to reduce the computational cost, a dimension reduction is often applied by a pooling step after convolutional operations. Usually we consider an operation that can (approximately) extract rotational or translational invariance features. Among the various types of pooling methods such as average pooling, max pooling, and stochastic pooling, we consider max pooling as an illustration because it is the most used setting for CNN. We show an example of max pooling by considering two 4×4 images, A and B, in Figure 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 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 derive the mathematical representation. Similar to Section 2.1.1, we consider the operation as a separate layer for the easy description though in our implementation pooling is just an operation at the end of the convolutional layer. Assume $Z^{\text{in},i}$ is an input image. We partition every channel of $Z^{\text{in},i}$ into non-overlapping sub-regions by $h \times h$ filters with the stride $s = h$.[2] This partition step is a special case of how we generate sub-images in convolutional operations. Therefore, by the same definition as (8) we can generate the matrix

$$\phi(Z^{\text{in},i}) = \text{mat}(P_\phi \text{vec}(Z^{\text{in},i}))_{hh \times d^{\text{out}} a^{\text{out}} b^{\text{out}}}, \tag{15}$$

where

$$a^{\text{out}} = \lfloor \frac{a^{\text{in}}}{h} \rfloor,\ b^{\text{out}} = \lfloor \frac{b^{\text{in}}}{h} \rfloor,\ d^{\text{out}} = d^{\text{in}}. \tag{16}$$

If for example max pooling is considered, to select the largest element of each sub-region, there exists a matrix

$$M^i \in R^{d^{\text{out}} a^{\text{out}} b^{\text{out}} \times hhd^{\text{out}} a^{\text{out}} b^{\text{out}}}$$

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

$$Z^{\text{out},i} = \text{mat}\left(M^i \text{vec}(\phi(Z^{\text{in},i}))\right)_{d^{\text{out}} \times a^{\text{out}} b^{\text{out}}}. \tag{17}$$

A comparison with (11) shows that M^i is in a similar role to the weight matrix W.

By combining (15) and (17), we have

$$Z^{\text{out},i} = \text{mat}\left(P^i_{\text{pool}} \text{vec}(Z^{\text{in},i})\right)_{d^{\text{out}} \times a^{\text{out}} b^{\text{out}}}, \tag{18}$$

where

$$P^i_{\text{pool}} = M^i P_\phi \in R^{d^{\text{out}} a^{\text{out}} b^{\text{out}} \times d^{\text{in}} a^{\text{in}} b^{\text{in}}}. \tag{19}$$

Note that this derivation is not limited to max pooling. It is valid for any pooling operation that can be represented in a form of (17).

2.1.3 Summary of a Convolutional Layer. For the practical implementation, we find it is more suitable to consider padding and pooling as part of the convolutional layers. Here we discuss details of considering all operations together. The whole convolutional layer involves the following procedure:

$$\begin{aligned} Z^{m,i} &\rightarrow \text{padding by (14)} \rightarrow \text{convolutional operations by (11), (12)} \\ &\rightarrow \text{pooling by (18)} \rightarrow Z^{m+1,i}, \end{aligned} \tag{20}$$

where $Z^{m,i}$ and $Z^{m+1,i}$ are input and output of the mth layer, respectively.

We use the following symbols to denote image sizes at different stages of the convolutional layer.

$$\begin{aligned} &a^m,\ b^m:\ \text{size in the beginning} \\ &a^m_{\text{pad}},\ b^m_{\text{pad}}:\ \text{size after padding} \\ &a^m_{\text{conv}},\ b^m_{\text{conv}}:\ \text{size after convolution.} \end{aligned}$$

Table 1 indicates how these values are $a^{\text{in}}, b^{\text{in}}, d^{\text{in}}$ and $a^{\text{out}}, b^{\text{out}}, d^{\text{out}}$ at different stages. We further denote the filter size, mapping matrices and weight matrices at the mth layer as

$$h^m,\ P^m_{\text{pad}},\ P^m_\phi,\ P^{m,i}_{\text{pool}},\ W^m,\ \boldsymbol{b}^m.$$

Then from (11), (12), (14), (18), and Table 1, all operations can be summarized as

$$S^{m,i} = W^m \text{mat}(P^m_\phi P^m_{\text{pad}} \text{vec}(Z^{m,i}))_{h^m h^m d^m \times a^m_{\text{conv}} b^m_{\text{conv}}} + \boldsymbol{b}^m \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}},$$

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

Table 1. Detailed information of operations at a convolutional layer.

Operation	$a^{\text{in}},\ b^{\text{in}},\ d^{\text{in}}$	$a^{\text{out}},\ b^{\text{out}},\ d^{\text{out}}$	Input	Output
Padding: (14)	$a^m,\ b^m,\ d^m$	$a^m_{\text{pad}},\ b^m_{\text{pad}},\ d^m$	$Z^{m,i}$	$\text{pad}(Z^{m,i})$
Convolution: (11)	$a^m_{\text{pad}},\ b^m_{\text{pad}},\ d^m$	$a^m_{\text{conv}},\ b^m_{\text{conv}},\ d^{m+1}$	$\text{pad}(Z^{m,i})$	$S^{m,i}$
Convolution: (12)	$a^m_{\text{conv}},\ b^m_{\text{conv}},\ d^{m+1}$	$a^m_{\text{conv}},\ b^m_{\text{conv}},\ d^{m+1}$	$S^{m,i}$	$\sigma(S^{m,i})$
Pooling: (18)	$a^m_{\text{conv}},\ b^m_{\text{conv}},\ d^{m+1}$	$a^{m+1},\ b^{m+1},\ d^{m+1}$	$\sigma(S^{m,i})$	$Z^{m+1,i}$

$$= W^m \phi(\text{pad}(Z^{m,i}))_{h^m h^m d^m \times a^m_{\text{conv}} b^m_{\text{conv}}} + \boldsymbol{b}^m \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}}, \tag{21}$$

and

$$Z^{m+1,i} = \text{mat}(P^{m,i}_{\text{pool}} \text{vec}(\sigma(S^{m,i})))_{d^{m+1} \times a^{m+1} b^{m+1}}. \tag{22}$$

2.2 Fully-Connected Layer

After passing through the convolutional layers, we concatenate columns in the matrix in (22) to form the input vector of the first fully-connected layer.

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

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

$$W^m \in R^{n_{m+1} \times n_m} \text{ and } \boldsymbol{b}^m \in R^{n_{m+1} \times 1}, \tag{23}$$

where n_m and n_{m+1} are the numbers of neurons in layers m and $m+1$, respectively.[3] If $\boldsymbol{z}^{m,i} \in R^{n_m}$ is the input vector, the following operations are applied to generate the output vector $\boldsymbol{z}^{m+1,i} \in R^{n_{m+1}}$.

$$\boldsymbol{s}^{m,i} = W^m \boldsymbol{z}^{m,i} + \boldsymbol{b}^m, \tag{24}$$

$$\boldsymbol{z}^{m+1,i} = \sigma(\boldsymbol{s}^{m,i}). \tag{25}$$

2.3 The Overall Optimization Problem

We can collect all model parameters such as filters of convolutional layers in (10) and weights/biases in (23) 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 section,

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

The output $\boldsymbol{z}^{L+1,i}$ of the last layer is a function of $\boldsymbol{\theta}$. We can apply a loss function $\xi(\boldsymbol{z}^{L+1}; \boldsymbol{y}, Z^1)$ to check how close $\boldsymbol{z}^{L+1,i}$ is to the label vector $\boldsymbol{y}^i$. For example, if the squared loss is considered, then

$$\xi(\boldsymbol{z}^{L+1}; \boldsymbol{y}, Z^1) = ||\boldsymbol{z}^{L+1} - \boldsymbol{y}||^2. \tag{26}$$

The optimization problem to train a CNN is

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}),\ \text{where } f(\boldsymbol{\theta}) = \frac{1}{2C}\boldsymbol{\theta}^T\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i}). \tag{27}$$

The first term with the parameter $C > 0$ avoids overfitting by regularization, while the second term is the average training loss.

[3] $n_{L^c+1} = d^{L^c+1} a^{L^c+1} b^{L^c+1}$ and $n_{L+1} = K$ is the number of classes.

3 HESSIAN-FREE NEWTON METHODS FOR TRAINING CNN

To solve an unconstrained minimization problem such as (27), a Newton method iteratively finds a search direction $\boldsymbol{d}$ by solving the following second-order approximation.

$$\min_{\boldsymbol{d}} \nabla f(\boldsymbol{\theta})^T \boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T \nabla^2 f(\boldsymbol{\theta})\boldsymbol{d}, \tag{28}$$

where $\nabla f(\boldsymbol{\theta})$ and $\nabla^2 f(\boldsymbol{\theta})$ are the gradient vector and the Hessian matrix, respectively. In this section we present details of applying a Newton method to solve the CNN problem (27).

3.1 Procedure of the Newton Method

For problem (27), the gradient is

$$\nabla f(\boldsymbol{\theta}) = \frac{1}{C}\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} (J^i)^T \nabla_{\boldsymbol{z}^{L+1,i}} \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i}), \tag{29}$$

where

$$J^i = \begin{bmatrix} \frac{\partial z_1^{L+1,i}}{\partial \theta_1} & \cdots & \frac{\partial z_1^{L+1,i}}{\partial \theta_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \theta_1} & \cdots & \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \theta_n} \end{bmatrix}_{n_{L+1}\times n}, \ i = 1, \ldots, l, \tag{30}$$

is the Jacobian of $\boldsymbol{z}^{L+1,i}$. The Hessian matrix of $f(\boldsymbol{\theta})$ is

$$\begin{aligned} \nabla^2 f(\boldsymbol{\theta}) = & \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l} (J^i)^T B^i J^i \\ & + \frac{1}{l}\sum_{i=1}^{l}\sum_{j=1}^{n_{L+1}} \frac{\partial \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i})}{\partial z_j^{L+1,i}} \begin{bmatrix} \frac{\partial^2 z_j^{L+1,i}}{\partial \theta_1 \partial \theta_1} & \cdots & \frac{\partial^2 z_j^{L+1,i}}{\partial \theta_1 \partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 z_j^{L+1,i}}{\partial \theta_n \partial \theta_1} & \cdots & \frac{\partial^2 z_j^{L+1,i}}{\partial \theta_n \partial \theta_n} \end{bmatrix}, \end{aligned} \tag{31}$$

where $\mathcal{I}$ is the identity matrix and B^i is the Hessian of $\xi(\cdot)$ with respect to $\boldsymbol{z}^{L+1,i}$:

$$B^i_{ts} = \frac{\partial^2 \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i})}{\partial z_t^{L+1,i} \partial z_s^{L+1,i}}, \ t = 1, \ldots, n_{L+1}, \ s = 1, \ldots, n_{L+1}. \tag{32}$$

From now on for simplicity we let

$$\xi_i \equiv \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i}).$$

In general (31) is not positive semi-definite, so $f(\boldsymbol{\theta})$ is non-convex for deep learning. The sub-problem (28) is difficult to solve and the resulting direction may not lead to the decrease of the function value. Following past works [20, 29], we consider the following Gauss-Newton approximation [24]

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l} (J^i)^T B^i J^i \approx \nabla^2 f(\boldsymbol{\theta}). \tag{33}$$

If $\xi(\boldsymbol{z}^{L+1}; \boldsymbol{y}, Z^1)$ is convex in $\boldsymbol{z}^{L+1}$, then B^i is positive semi-definite. Then G is positive definite and (28) becomes the same as solving the following linear system.

$$G\boldsymbol{d} = -\nabla f(\boldsymbol{\theta}). \tag{34}$$

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 an exponentially decreased sequence like $\{1, \frac{1}{2}, \frac{1}{4}, \ldots\}$ satisfying the following sufficient decrease condition.

$$f(\boldsymbol{\theta} + \alpha \boldsymbol{d}) \leq f(\boldsymbol{\theta}) + \eta \alpha \nabla f(\boldsymbol{\theta})^T \boldsymbol{d}. \tag{35}$$

In (35), $\eta \in (0, 1)$ is a pre-defined constant. The procedure to find α is called a backtracking line search.

Past works (e.g., [20, 30]) have shown that sometimes (34) 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 [21, 30], the Levenberg-Marquardt method [17, 19] is used to solve the following linear system rather than (34).

$$(G + \lambda I)\boldsymbol{d} = -\nabla f(\boldsymbol{\theta}), \tag{36}$$

where λ is a parameter decided by how good the function reduction is. Specifically, if $\boldsymbol{\theta} + \boldsymbol{d}$ is the next iterate after line search, 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

$$\lambda_{\text{next}} = \begin{cases} \lambda \times \text{drop} & \rho > \rho_{\text{upper}}, \\ \lambda & \rho_{\text{lower}} \leq \rho \leq \rho_{\text{upper}}, \\ \lambda \times \text{boost} & \text{otherwise}, \end{cases} \tag{37}$$

where (drop,boost) with drop < 1 and boost > 1 are given constants. From (37) 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. Therefore, depending on the function reduction, λ decides if a more aggressive setting (i.e., direction closer to Newton) or a more conservative setting (i.e., direction closer to negative gradient) is considered.

Next, we discuss how to solve the linear system (36). When the number of variables n is large, the memory cost of the matrix G is $O(n^2)$, which is prohibitive.[4] To address this memory difficulty, for some optimization problems including neural networks, it has been shown that without explicitly storing G we can still calculate the product between G and any vector $\boldsymbol{v}$ [12, 20, 30]. For example, from (33),

$$(G + \lambda I)\boldsymbol{v} = (\frac{1}{C} + \lambda)\boldsymbol{v} + \frac{1}{l} \sum_{i=1}^{l} \left((J^i)^T \left(B^i (J^i \boldsymbol{v}) \right) \right). \tag{38}$$

If the product between J^i and a vector can be easily calculated, we can apply the conjugate gradient (CG) method [8] to solve (34) by a sequence of matrix-vector products. Because G is not explicitly formed, 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 [18].

Because (38) involves a summation over all instances, the memory as well as computational cost may still be very high. Subsampled Hessian Newton methods have been proposed [2, 20, 29] to reduce the cost by taking the property that the second term in (33) is the average training loss. If the

[4] Consider the five layers network structure in Section 6 and double precision. For *CIFAR10*, the memory cost of the matrix G will take about 208GB.

ALGORITHM 1: A subsampled Hessian Newton method for CNN.

Given initial $\boldsymbol{\theta}$. Calculate $f(\boldsymbol{\theta})$;
while $\nabla f(\boldsymbol{\theta}) \neq \mathbf{0}$ **do**
  Choose a set $S \subset \{1, \ldots, l\}$;
  Compute $\nabla f(\boldsymbol{\theta})$ and the needed information for Gauss Newton matrix-vector products;
  Approximately solve the linear system in (36) by CG to obtain a direction $\boldsymbol{d}$;
  $\alpha = 1$;
  while true **do**
    Compute $f(\boldsymbol{\theta} + \alpha \boldsymbol{d})$;
    if (35) is satisfied **then**
      break;
    end
    $\alpha \leftarrow \alpha/2$;
  end
  Update λ based on (37);
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{d}$;
end

large number of data points are assumed to be from the same distribution, (33) can be reasonably approximated by selecting a subset $S \subset \{1, \ldots, l\}$ and having

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

Then (38) becomes

$$(G^S + \lambda I)\boldsymbol{v} = (\frac{1}{C} + \lambda)\boldsymbol{v} + \frac{1}{|S|} \sum_{i \in S} \left((J^i)^T \left(B^i (J^i \boldsymbol{v}) \right) \right) \approx (G + \lambda I)\boldsymbol{v}. \tag{39}$$

A summary of the subsampled Newton method is in Algorithm 1.

3.2 Gradient Evaluation

In order to solve (34), $\nabla f(\boldsymbol{\theta})$ is needed. It can be obtained by (29) if the Jacobian matrices $J^i,\ i = 1, \ldots, l$ are available. From (38), it seems that $J^i, \forall i$ are also needed for the matrix-vector product in CG. However, as mentioned in Section 3.1, in practice a sub-sampled Hessian method is used, so from (39) only a subset of $J^i, \forall i$ are needed.[5] Therefore we present a backward process to calculate the gradient without using Jacobian.

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

$$\frac{\partial f}{\partial W^m} = \frac{1}{C} W^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial W^m}, \tag{40}$$

$$\frac{\partial f}{\partial \boldsymbol{b}^m} = \frac{1}{C} \boldsymbol{b}^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial \boldsymbol{b}^m}. \tag{41}$$

Because (40) is in a matrix form, following past developments such as [27], it is easier to transform them to a vector form for the derivation. To begin, we list the following properties of the vec$(\cdot)$

[5] Further, we do not need to explicitly store J^i for matrix-vector products; see Sections 3.3 and 3.4.

function, in which $\otimes$ is the Kronecker product.

$$\begin{aligned}\mathrm{vec}(AB) &= (\mathcal{I} \otimes A)\mathrm{vec}(B), &(42)\\ &= (B^T \otimes \mathcal{I})\mathrm{vec}(A), &(43)\\ \mathrm{vec}(AB)^T &= \mathrm{vec}(B)^T(\mathcal{I} \otimes A^T), &(44)\\ &= \mathrm{vec}(A)^T(B \otimes \mathcal{I}). &(45)\end{aligned}$$

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.

For the convolutional layers, from (11) and Table 1, we have

$$\begin{aligned}\mathrm{vec}(S^{m,i}) &= \mathrm{vec}(W^m\phi(\mathrm{pad}(Z^{m,i}))) + \mathrm{vec}(\boldsymbol{b}^m \mathbb{1}^T_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}})\\ &= \left(\mathcal{I}_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}} \otimes W^m\right)\mathrm{vec}(\phi(\mathrm{pad}(Z^{m,i}))) + (\mathbb{1}_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}} \otimes \mathcal{I}_{d^{m+1}})\boldsymbol{b}^m &(46)\\ &= \left(\phi(\mathrm{pad}(Z^{m,i}))^T \otimes \mathcal{I}_{d^{m+1}}\right)\mathrm{vec}(W^m) + (\mathbb{1}_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}} \otimes \mathcal{I}_{d^{m+1}})\boldsymbol{b}^m, &(47)\end{aligned}$$

where (46) and (47) are from (42) and (43), respectively.

For the fully-connected layers, from (24), we have

$$\begin{aligned}\boldsymbol{s}^{m,i} &= W^m\boldsymbol{z}^{m,i} + \boldsymbol{b}^m\\ &= (\mathcal{I}_1 \otimes W^m)\,\boldsymbol{z}^{m,i} + (\mathbb{1}_1 \otimes \mathcal{I}_{n_{m+1}})\boldsymbol{b}^m &(48)\\ &= \left((\boldsymbol{z}^{m,i})^T \otimes \mathcal{I}_{n_{m+1}}\right)\mathrm{vec}(W^m) + (\mathbb{1}_1 \otimes \mathcal{I}_{n_{m+1}})\boldsymbol{b}^m, &(49)\end{aligned}$$

where (48) and (49) are from (42) and (43), respectively.

An advantage of using (46) and (48) is that they are in the same form. Further, if for fully-connected layers we define

$$\phi(\mathrm{pad}(\boldsymbol{z}^{m,i})) = \mathcal{I}_{n_m}\boldsymbol{z}^{m,i},\ L^c < m \le L+1,$$

then (47) and (49) are in the same form. Thus we can derive the gradient of convolutional and fully-connected layers together. We begin with calculating the gradient for convolutional layers. From (47), we derive

$$\begin{aligned}\frac{\partial \xi_i}{\partial \mathrm{vec}(W^m)^T} &= \frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T}\frac{\partial \mathrm{vec}(S^{m,i})}{\partial \mathrm{vec}(W^m)^T}\\ =\frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T}&\left(\phi(\mathrm{pad}(Z^{m,i}))^T \otimes \mathcal{I}_{d^{m+1}}\right) = \mathrm{vec}\left(\frac{\partial \xi_i}{\partial S^{m,i}}\phi(\mathrm{pad}(Z^{m,i}))^T\right)^T &(50)\end{aligned}$$

and

$$\begin{aligned}\frac{\partial \xi_i}{\partial (\boldsymbol{b}^m)^T} &= \frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T}\frac{\partial \mathrm{vec}(S^{m,i})}{\partial (\boldsymbol{b}^m)^T}\\ &= \frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T}\left(\mathbb{1}_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}} \otimes \mathcal{I}_{d^{m+1}}\right) = \mathrm{vec}\left(\frac{\partial \xi_i}{\partial S^{m,i}}\mathbb{1}_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}}\right)^T, &(51)\end{aligned}$$

where (50) and (51) are from (45). To calculate (50), $\phi(\text{pad}(Z^{m,i}))$ has been available from the forward process of calculating the function value. In (50) and (51), $\partial\xi_i/\partial S^{m,i}$ is also needed and can be obtained by the following backward process.

By assuming that $\partial\xi_i/\partial Z^{m+1,i}$ is available, we show details of calculating $\partial\xi_i/\partial S^{m,i}$ and $\partial\xi_i/\partial Z^{m,i}$ for layer m. From (20), the workflow is as follows.

$$Z^{m,i} \leftarrow \text{padding} \leftarrow \text{convolution} \leftarrow \sigma(S^{m,i}) \leftarrow \text{pooling} \leftarrow Z^{m+1,i}. \tag{52}$$

We have

$$\frac{\partial\xi_i}{\partial\text{vec}(S^{m,i})^T} = \frac{\partial\xi_i}{\partial\text{vec}(Z^{m+1,i})^T}\frac{\partial\text{vec}(Z^{m+1,i})}{\partial\text{vec}(\sigma(S^{m,i}))^T}\frac{\partial\text{vec}(\sigma(S^{m,i}))}{\partial\text{vec}(S^{m,i})^T} \tag{53}$$

$$= \left(\frac{\partial\xi_i}{\partial\text{vec}(Z^{m+1,i})^T}P^{m,i}_{\text{pool}}\right)\frac{\partial\text{vec}(\sigma(S^{m,i}))}{\partial\text{vec}(S^{m,i})^T} \tag{54}$$

where (54) is from (22).

For the special case if $\sigma(\cdot)$ can be reduced to a scalar function with

$$\sigma(S^{m,i})_{(p,q)} = \sigma(s^{m,i}_{p,q}), \tag{55}$$

then

$$\frac{\partial\text{vec}(\sigma(S^{m,i}))}{\partial\text{vec}(S^{m,i})^T}$$

is a diagonal matrix.[6] We can rewrite (54) into

$$\left(\frac{\partial\xi_i}{\partial\text{vec}(Z^{m+1,i})^T}\,P^{m,i}_{\text{pool}}\right)\odot\ \text{vec}(\sigma'(S^{m,i}))^T, \tag{56}$$

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

$$\sigma'(S^{m,i})_{(p,q)} = \sigma'(s^{m,i}_{p,q}).$$

Next, we must calculate $\partial\xi_i/\partial Z^{m,i}$ and pass it to the previous layer.

$$\frac{\partial\xi_i}{\partial\text{vec}(Z^{m,i})^T} = \frac{\partial\xi_i}{\partial\text{vec}(S^{m,i})^T}\frac{\partial\text{vec}(S^{m,i})}{\partial\text{vec}(\phi(\text{pad}(Z^{m,i})))^T}\frac{\partial\text{vec}(\phi(\text{pad}(Z^{m,i})))}{\partial\text{vec}(\text{pad}(Z^{m,i}))^T}\frac{\partial\text{vec}(\text{pad}(Z^{m,i}))}{\partial\text{vec}(Z^{m,i})^T}$$

$$= \frac{\partial\xi_i}{\partial\text{vec}(S^{m,i})^T}\left(\mathcal{I}_{a^m_{\text{conv}}b^m_{\text{conv}}}\otimes W^m\right)P^m_\phi P^m_{\text{pad}} \tag{57}$$

$$= \text{vec}\left((W^m)^T\frac{\partial\xi_i}{\partial S^{m,i}}\right)^T P^m_\phi P^m_{\text{pad}}, \tag{58}$$

where (57) is from (8), (14) and (46), and (58) is from (44).

For fully-connected layers, by the same form in (48), (49), (46) and (47), we immediately get the following results from (50), (51), (54) and (58).

$$\frac{\partial\xi_i}{\partial\text{vec}(W^m)^T} = \text{vec}\left(\frac{\partial\xi_i}{\partial\boldsymbol{s}^{m,i}}(\boldsymbol{z}^{m,i})^T\right)^T, \tag{59}$$

$$\frac{\partial\xi_i}{\partial(\boldsymbol{b}^m)^T} = \frac{\partial\xi_i}{\partial(\boldsymbol{s}^{m,i})^T}, \tag{60}$$

$$\frac{\partial\xi_i}{\partial(\boldsymbol{z}^{m,i})^T} = \left((W^m)^T\frac{\partial\xi_i}{\partial(\boldsymbol{s}^{m,i})}\right)^T\mathcal{I}_{n_m} = \left((W^m)^T\frac{\partial\xi_i}{\partial(\boldsymbol{s}^{m,i})}\right)^T, \tag{61}$$

[6] For example, the RELU function in (13) satisfies such a property.

where

$$\frac{\partial \xi_i}{\partial (\boldsymbol{s}^{m,i})^T} = \frac{\partial \xi_i}{\partial (\boldsymbol{z}^{m+1,i})^T} \frac{\partial \sigma(\boldsymbol{s}^{m,i})}{\partial (\boldsymbol{s}^{m,i})^T}. \tag{62}$$

If $\sigma(\cdot)$ can be reduced to a scalar function as in (55), then from (56), (62) can be simplified to

$$\frac{\partial \xi_i}{\partial (\boldsymbol{s}^{m,i})^T} = \frac{\partial \xi_i}{\partial (\boldsymbol{z}^{m+1,i})^T} \odot \sigma'(\boldsymbol{s}^{m,i})^T.$$

3.3 Jacobian Evaluation

For the matrix-vector product (33), the Jacobian matrix is needed. We note that it can be partitioned into L blocks according to layers.

$$J^i = \begin{bmatrix} J^{1,i} & J^{2,i} & \dots & J^{L,i} \end{bmatrix}, \ m = 1, \dots, L, \ i = 1, \dots, l, \tag{63}$$

where

$$J^{m,i} = \begin{bmatrix} \dfrac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} & \dfrac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix}.$$

The calculation is very similar to that for the gradient. For the convolutional layers, from (50) and (51), we have

$$\begin{aligned}
\begin{bmatrix} \dfrac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} & \dfrac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix} &= \begin{bmatrix} \dfrac{\partial z_1^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} & \dfrac{\partial z_1^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \\ \vdots & \\ \dfrac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} & \dfrac{\partial z_{n_{L+1}}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix} \\
&= \begin{bmatrix} \mathrm{vec}(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T)^T & \mathrm{vec}(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m})^T \\ \vdots & \\ \mathrm{vec}(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T)^T & \mathrm{vec}(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m})^T \end{bmatrix} \\
&= \begin{bmatrix} \mathrm{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \begin{bmatrix} \phi(\mathrm{pad}(Z^{m,i}))^T & \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m} \end{bmatrix}\right)^T \\ \vdots \\ \mathrm{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \begin{bmatrix} \phi(\mathrm{pad}(Z^{m,i}))^T & \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m} \end{bmatrix}\right)^T \end{bmatrix}.
\end{aligned} \tag{64}$$

Clearly, each row in (64) involves the product of two matrices. Following [30] to take this property for fully-connected networks, we explain that explicitly forming $J^{m,i}$ is not needed. Instead, if

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \cdots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}, \text{ and } \begin{bmatrix} \phi(\mathrm{pad}(Z^{m,i}))^T & \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m} \end{bmatrix}, \forall i \tag{65}$$

are available, then we are able to conduct the matrix-vector product in (38); see more details in Section 3.4. For the two types of matrices in (65), the latter has been obtained in the forward process of calculating the function value. For the former we develop the following backward process to calculate $\partial \boldsymbol{z}^{L+1,i} / \partial \mathrm{vec}(S^{m,i})^T, \ \forall i$.

Assume that $\partial \boldsymbol{z}^{L+1,i} / \partial \mathrm{vec}(Z^{m+1,i})^T$ are available. From (54), we have

$$\frac{\partial z_j^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T} = \left(\frac{\partial z_j^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} P_{\mathrm{pool}}^{m,i} \right) \frac{\partial \mathrm{vec}(\sigma(S^{m,i}))}{\partial \mathrm{vec}(S^{m,i})^T}, \ j = 1, \dots, n_{L+1}.$$

These vectors can be written together as

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T} = \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} P^{m,i}_{\mathrm{pool}}\right) \frac{\partial \mathrm{vec}(\sigma(S^{m,i}))}{\partial \mathrm{vec}(S^{m,i})^T}. \tag{66}$$

If $\sigma(\cdot)$ can be reduced to a scalar function as in (55), from (56), we have

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T} = \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} P^{m,i}_{\mathrm{pool}}\right) \odot \left(\mathbb{1}_{n_{L+1}} \mathrm{vec}(\sigma'(S^{m,i}))^T\right). \tag{67}$$

We then generate $\partial \boldsymbol{z}^{L+1,i}/\partial \mathrm{vec}(Z^{m,i})^T$ and pass it to the previous layer. From (58), we derive

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(Z^{m,i})^T} = \begin{bmatrix} \frac{\partial z_1^{L+1,i}}{\partial \mathrm{vec}(Z^{m,i})^T} \\ \vdots \\ \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \mathrm{vec}(Z^{m,i})^T} \end{bmatrix} = \begin{bmatrix} \mathrm{vec}\left((W^m)^T \frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\right)^T P^m_\phi P^m_{\mathrm{pad}} \\ \vdots \\ \mathrm{vec}\left((W^m)^T \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\right)^T P^m_\phi P^m_{\mathrm{pad}} \end{bmatrix}. \tag{68}$$

For the fully-connected layers, we follow the same derivation of gradient to have

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} = \begin{bmatrix} \mathrm{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial \boldsymbol{s}^{m,i}} (\boldsymbol{z}^{m,i})^T\right) & \dots & \mathrm{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \boldsymbol{s}^{m,i}} (\boldsymbol{z}^{m,i})^T\right) \end{bmatrix}^T, \tag{69}$$

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T}, \tag{70}$$

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{z}^{m+1,i})^T} \frac{\partial \sigma(\boldsymbol{s}^{m,i})}{\partial (\boldsymbol{s}^{m,i})^T}, \tag{71}$$

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{z}^{m,i})^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} W^m. \tag{72}$$

Note that if $\sigma(\cdot)$ can be reduced to a scalar function as in (55), (71) can be rewritten as

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{z}^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \sigma'(\boldsymbol{z}^{m+1,i})^T\right).$$

3.4 Gauss-Newton Matrix-Vector Products

As mentioned in Section 3.1, conjugate gradient (CG) methods are used to solve the linear system in (34). The main operation at each CG step is the Gauss-Newton matrix-vector product in (38) or (39) depending on whether the subsampled setting is applied. Here we derive details for calculating the product.

From (63), we rearrange (33) to

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} B^i \begin{bmatrix} J^{1,i} & \dots & J^{L,i} \end{bmatrix} \tag{73}$$

and the Gauss-Newton matrix vector product becomes

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

$$= \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l}\begin{bmatrix}(J^{1,i})^T\\ \vdots \\ (J^{L,i})^T\end{bmatrix}\left(B^i\sum_{m=1}^{L}J^{m,i}\boldsymbol{v}^m\right),\ \text{where } \boldsymbol{v}=\begin{bmatrix}\boldsymbol{v}^1\\ \vdots\\ \boldsymbol{v}^L\end{bmatrix}, \tag{74}$$

and each $\boldsymbol{v}^m, m = 1,\dots,L$ has the same length as the number of variables (including bias) at the mth layer.

For the convolutional layers, from (64) and (74), we have

$$J^{m,i}\boldsymbol{v}^m = \begin{bmatrix}\text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T\ \mathbb{1}_{a^m_{\text{conv}}b^m_{\text{conv}}}\right]\right)^T\boldsymbol{v}^m\\ \vdots\\ \text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T\ \mathbb{1}_{a^m_{\text{conv}}b^m_{\text{conv}}}\right]\right)^T\boldsymbol{v}^m\end{bmatrix}. \tag{75}$$

To simplify (75), we use the following property

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

to have that for example, the first element in (75) is

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

Therefore,

$$J^{m,i}\boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial\text{vec}(S^{m,i})^T}\text{vec}\left(\text{mat}(\boldsymbol{v}^m)_{d^{m+1}\times(h^mh^md^m+1)}\begin{bmatrix}\phi(\text{pad}(Z^{m,i}))\\ \mathbb{1}^T_{a^m_{\text{conv}}b^m_{\text{conv}}}\end{bmatrix}\right). \tag{76}$$

By (76) and the discussion around (65), we can calculate $J^{m,i}\boldsymbol{v}^m$ without explicitly forming $J^{m,i}$.

Next, from (74), we sum results of all layers

$$\sum_{m=1}^{L}J^{m,i}\boldsymbol{v}^m \tag{77}$$

and then calculate

$$\boldsymbol{q}^i = B^i\left(\sum_{m=1}^{L}J^{m,i}\boldsymbol{v}^m\right). \tag{78}$$

From the definition in (32), B^i is easy to calculate if the loss function is not complicated. For example, if the squared loss in (26) is used, from (32),

$$B^i_{ts} = \frac{\partial^2\xi^i}{\partial z_t^{L+1,i}\partial z_s^{L+1,i}} = \frac{\partial^2(\sum_{j=1}^{n_{L+1}}(z_j^{L+1,i}-y_j^i)^2)}{\partial z_t^{L+1,i}\partial z_s^{L+1,i}} = \begin{cases}2 & \text{if } t=s,\\ 0 & \text{otherwise.}\end{cases} \tag{79}$$

After deriving (78), from (64) and (74), we calculate

$$\begin{aligned}&(J^{m,i})^T\boldsymbol{q}^i\\ =\ &\left[\text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T\ \mathbb{1}_{a^m_{\text{conv}}b^m_{\text{conv}}}\right]\right)\ \cdots\ \text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T\ \mathbb{1}_{a^m_{\text{conv}}b^m_{\text{conv}}}\right]\right)\right]\boldsymbol{q}^i\end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=1}^{n_{L+1}} q_j^i \mathrm{vec}\left(\frac{\partial z_j^{L+1,i}}{\partial S^{m,i}}\left[\phi(\mathrm{pad}(Z^{m,i}))^T\ \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}\right]\right)\\
&= \mathrm{vec}\left(\sum_{j=1}^{n_{L+1}} q_j^i \left(\frac{\partial z_j^{L+1,i}}{\partial S^{m,i}}\left[\phi(\mathrm{pad}(Z^{m,i}))^T\ \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}\right]\right)\right)\\
&= \mathrm{vec}\left(\left(\sum_{j=1}^{n_{L+1}} q_j^i \frac{\partial z_j^{L+1,i}}{\partial S^{m,i}}\right)\left[\phi(\mathrm{pad}(Z^{m,i}))^T\ \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}\right]\right)\\
&= \mathrm{vec}\left(\mathrm{mat}\left(\left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T}\right)^T \boldsymbol{q}^i\right)_{d^{m+1}\times a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}\left[\phi(\mathrm{pad}(Z^{m,i}))^T\ \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}\right]\right). \qquad (80)
\end{aligned}
$$

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

$$
J^{m,i}\boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T}\mathrm{mat}(\boldsymbol{v}^m)_{n_{m+1}\times(n_m+1)}\begin{bmatrix}\boldsymbol{z}^{m,i}\\ \mathbb{1}_1\end{bmatrix}, \qquad (81)
$$

$$
(J^{m,i})^T\boldsymbol{q}^i = \mathrm{vec}\left(\left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T}\right)^T \boldsymbol{q}^i\left[(\boldsymbol{z}^{m,i})^T\ \mathbb{1}_1\right]\right). \qquad (82)
$$

3.5 Mini-Batch Function and Gradient Evaluation

Later in Section 5.1 to discuss the memory usage, one important conclusion is that the memory consumption is proportional to the number of data in several places of the Newton method. This fact causes difficulties in handling large data sets, so here we discuss some effective settings to reduce the memory usage.

In the subsampled Newton method discussed in Section 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 direction for shorter running time per iteration, it also effectively reduces the memory consumption. For example, at the mth convolutional layer, we only need to store the following matrices

$$
\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T},\ \forall i \in S \qquad (83)
$$

for the Gauss-Newton matrix-vector products.

However, in function and gradient evaluations we still need the whole training data. Fortunately, both operations involve the summation of independent results over all instances, so we follow [30] to have a mini-batch setting. By splitting the index set $\{1,\ldots,l\}$ of data to, for example, R equal-sized subsets $S_1,\ldots,S_R$, we sequentially calculate the result corresponding to each subset and accumulate them for the final output. For example, to have $Z^{m,i}$ needed in the backward process for calculating the gradient, we must store them after the forward process for function evaluation. By using a subset, only $Z^{m,i}$ with i in this subset are stored, so the memory usage can be dramatically reduced.

For the Gauss-Newton matrix-vector product, to calculate (83) under the subsampled scheme, we have a set S and use $Z^{m,i},\ \forall i \in S$. However, under the mini-batch setting, the needed values may not be kept in the process of function and gradient evaluations. A simple solution 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 Gauss-Newton matrix-vector products.

3.6 Some Notes on Practical Implementations

We discuss some implementation tricks if

- max pooling, and
- RELU activation function

are considered. In (56), from (13), $\sigma'(S^{m,i})$ is now

$$\sigma'(S^{m,i})_{(p,q)} = I[Z^{m+1,i}]_{(p,q)} = \begin{cases} 1 & \text{if } z^{m+1,i}_{(p,q)} > 0, \\ 0 & \text{otherwise,} \end{cases}$$

where I is the indicator function.

Recall that to calculate (50),

$$Z^{m,i},\ \forall m$$

must be stored after the forward process. However, we also need $S^{m,i}$ in (56). To avoid storing both $Z^{m,i}$ and $S^{m,i}$, we can replace (56) with the following calculation.

$$\frac{\partial \xi_i}{\partial \mathrm{vec}(S^{m,i})^T} = \left(\frac{\partial \xi_i}{\partial \mathrm{vec}(Z^{m+1,i})^T} \odot \mathrm{vec}(I[Z^{m+1,i}])^T \right) P^{m,i}_{\mathrm{pool}}. \tag{84}$$

The reason is that, for (56),

$$\frac{\partial \xi_i}{\partial \mathrm{vec}(Z^{m+1,i})^T} \times P^{m,i}_{\mathrm{pool}} \tag{85}$$

generates a large zero vector and puts values of $\partial \xi_i / \partial \mathrm{vec}(Z^{m+1,i})^T$ into positions selected earlier in the max pooling operation. Then, element-wise multiplications of (85) and $\mathrm{vec}(I[Z^{m+1,i}])^T$ are conducted. Because positions not selected in the max pooling procedure are zeros after (85) and they are still zeros after the Hadamard product between (85) and $\mathrm{vec}(I[Z^{m+1,i}])^T$, (56) and (84) give the same results.

Similar to (84), for the Jacobian evaluations, we replace (67) with

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T} = \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \mathrm{vec}(I[Z^{m+1,i}])^T \right) \right) P^{m,i}_{\mathrm{pool}}.$$

3.7 Implementation Details

See Section II of supplementary materials.

4 RELATED WORKS OF NEWTON METHODS FOR TRAINING NEURAL NETWORKS

Some past works of applying Newton methods on neural networks are summarized in Table 2. Among those works, LeCun et al. [15] investigated several second-order optimization methods and discuss some tricks for training fully-connected neural networks. Martens [20] successfully applied a Newton method with the Hessian-free approach for training autoencoders. Martens and Sutskever [21] discussed some techniques about practical Hessian-free approaches, including the R operator, several damping mechanisms, preconditioning, analysis of mini-batch gradient and Hessian information, etc., on fully-connected and recurrent networks. Based on [21], Kiros [9] considered subsampled gradient by selecting a subset at each Newton iteration. Then a further subset is chosen to construct subsampled Hessian for the Newton method to train autoencoders and fully-connected neural networks. Wang et al. [29] improved upon subsampled Newton methods by combining the previous direction and the current Newton direction as the search direction. Wang et al. [30] extended the same idea to large-scale fully-connected neural networks in a distributed environment. Botev et al. [1] considered a block diagonal approximation of the Gauss-Newton matrix in a Newton method for training autoencoders.

Table 2. Previous studies of Newton methods on different types of neural networks, sorted in chronological order.

	Types of Neural Networks
LeCun et al. [15]	Fully-connected
Martens [20]	Autoencoder
Martens and Sutskever [21]	Fully-connected, Recurrent
Kiros [9]	Fully-connected, Autoencoder
Wang et al. [29, 30]	Fully-connected
Botev et al. [1]	Autoencoder

There are other works such as Quasi-Newton method [3, 12], Krylov subspace descent [28] and Kronecker-Factorization approximate curvature [5]. Some of these works investigate the use of second-order optimization methods for training CNN, but their settings are different from the Newton method considered here.

5 COST ANALYSIS OF NEWTON METHODS FOR CNN

In this section, we analyze the memory and computational cost per iteration. We consider that all training instances are used. If the subsampled Hessian in Section 3 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 $|S|$. Furthermore, if mini-batch function and gradient evaluation in Section 3.5 is applied, the number of instance l in the function and gradient evaluation can also be replaced by the size of each batch.

In this discussion we exclude the padding and the pooling operations because first they are optional steps and second they are not the bottleneck. Depending on the type of the activation function, the cost may vary. Here we assume that the RELU activation function is used, so from Section 3.6, $\sigma'(S^{m,i})$ does not have to be stored. In addition, for simplicity, the bias term is not considered.

5.1 Memory Requirement

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

$$W^m,\ m = 1, \ldots, L.$$

From (10) and (23), the memory usage is

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

(2) Gradient vector: For (40), the following matrix must be stored.

$$\frac{\partial f}{\partial \mathrm{vec}(W^m)^T},\ m = 1, \ldots, L.$$

Therefore, the memory usage is

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

(3) P_ϕ^m: We store row indices of non-zero positions of the 0/1 matrix P_ϕ^m for constructing $\phi(Z^{m,i})$. The memory usage is

$$\sum_{m=1}^{L^c} \left(h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m\right).$$

(4) Function evaluation: From Section 2, we store

$$Z^{m,i},\ m = 1, \ldots, L+1,\ \forall i.$$

Therefore, the memory usage is

$$l \times \left(\sum_{m=1}^{L^c} d^m a^m b^m + \sum_{m=L^c+1}^{L+1} n_m \right). \tag{86}$$

The reason why $Z^{m,i}, \forall m, i$ must be stored is because they are used later in the backward process for calculating the gradient; see (50).

(5) Gradient evaluation: To obtain the gradient in each layer m, we need the matrix

$$\frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T}, \forall i$$

for calculating

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

in the backward process. Note that there is no need to keep the matrices of all layers. All we have to store is the matrices for two adjacent layers. Thus, the memory usage is

$$l \times \max_{m=1,\ldots,L^c} \left(d^m a_{\text{conv}}^m b_{\text{conv}}^m + d^{m+1} a_{\text{conv}}^{m+1} b_{\text{conv}}^{m+1}\right)$$

for the convolutional layers and

$$l \times \max_{m=L^c+1,\ldots,L} \left(n_m + n_{m+1}\right).$$

for the fully-connected layers. This is much smaller than (86).

(6) Jacobian evaluation and Gauss-Newton matrix-vector products: At each CG procedure, several Gauss-Newton matrix-vector products are conducted, so we should maintain certain information. Besides W^m and $Z^{m,i}$, from (76), (80), (81) and (82), we must store

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

Thus, the memory usage is[7]

$$l \times n_{L+1} \times \left(\sum_{m=1}^{L^c} d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m + \sum_{m=L^c+1}^{L} n_{m+1} \right). \tag{87}$$

(7) In (21), $\phi(\text{pad}(Z^{m,i}))$ is needed. We discuss it in a separate item because $\phi(\text{pad}(Z^{m,i}))$ is also used in (50), (76), and (80) for gradient evaluation and Gauss-Newton matrix-vector products. Because P_ϕ^m and $Z^{m,i}$ are stored, $\phi(\text{pad}(Z^{m,i}))$ can be calculated when needed. However, the matrix must be temporarily stored. Thus the peak memory usage is

$$l \times \max_{m=1,\ldots,L^c} \left(h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m\right).$$

[7]Note that the dimension of $s^{m,i}$ in fully-connected layers is n_{m+1}.

This cost is smaller than that in (86) for storing $Z^{m,i}$. On the other hand, using P_ϕ^m and $Z^{m,i}$ to calculate $\phi(\mathrm{pad}(Z^{m,i}))$ is computationally expensive. If enough memory is available, we may store all $\phi(\mathrm{pad}(Z^{m,i}))$ after calculating them in the function evaluation. The memory usage is

$$l \times \sum_{m=1}^{L^c} \left(h^m h^m d^m a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m \right), \tag{88}$$

which becomes higher than (86) for $Z^{m,i}$.

From the above discussion, (86) or (88) dominates the memory usage in the function and gradient evaluation depending on whether $\phi(\mathrm{pad}(Z^{m,i}))$ is stored or not. On the other hand, (87) is the main cost for the Jacobian evaluation and Gauss-Newton matrix-vector products. To reduce the memory consumption, as mentioned, the sub-sampled Hessian technique in Section 3 reduces the usage in (87), while for (86) or (88) we use the mini-batch function and gradient evaluation technique described in Section 3.5.

5.2 Computational Cost

We show the computational cost for the mth convolutional/fully-connected layer.

(1) Function evaluation:

- Convolutional layers: From (8), (11), (12) and (21), the computational cost is

$$O(l \times h^m h^m d^m d^{m+1} a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m),$$

  where the bottleneck is on calculating

$$W^m \phi(\mathrm{pad}(Z^{m,i})).$$

- Fully-connected layers: From (24) and (25), the computational cost is

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

(2) Gradient evaluation:

- Convolutional layers: For (50), the computational cost is on a matrix-matrix product:

$$O(l \times h^m h^m d^m d^{m+1} a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m).$$

  For (56), because it is replaced by (84), the computational cost is on Hadamard products.

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

  For (58), the computational cost is

$$O(l \times h^m h^m d^m d^{m+1} a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m),$$

  which mainly comes from calculating

$$(W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}}, \forall i.$$

  Therefore, the total computational cost for the gradient evaluation is

$$O(l \times h^m h^m d^m d^{m+1} a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m).$$

- Fully-connected layers: From (59) and (61), the computational cost is

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

  For (62), the cost is smaller. Therefore, the total computational cost is

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

(3) Jacobian evaluation:

- Convolutional layers: The main computational cost is from calculating

$$(W^m)^T \frac{\partial z_j^{L+1,i}}{\partial S^{m,i}}, j = 1, \ldots, n_{L+1}, \forall i$$

in (68):

$$O\left(l \times n_{L+1} \times h^m h^m d^m d^{m+1} a^m_{\text{conv}} b^m_{\text{conv}}\right),$$

while others are less significant.
- Fully-connected layers: From (72), the computational cost is

$$O(l \times n_{L+1} \times n_{m+1} n_m).$$

(4) CG: The computational cost is the number of CG iterations (#CG) times the cost of a Gauss-Newton matrix-vector product.
- Convolutional layers: The main computational cost is from (76) and (80):

$$O\left(\#\text{CG} \times l \times d^{m+1} h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}\right),$$

while the cost of (32) is insignificant.
- Fully-connected layers: Similarly, the main computational cost is from (81) and (82):

$$O\left(\#\text{CG} \times l \times n_{m+1} n_m\right).$$

(5) line search: The computational cost is on multiple function evaluations.
- Convolutional layers:

$$O\left(\#\text{line search} \times l \times d^{m+1} h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}\right).$$

- Fully-connected layers:

$$O(\#\text{line search} \times l \times n_{m+1} n_m).$$

We summarize the cost in a convolutional layer. Clearly, the cost is proportional to the number of instances, l. After omitting the term $O\left(h^m h^m d^m d^{m+1} a^m_{\text{conv}} b^m_{\text{conv}}\right)$ in all operations, the cost of different components can be compared in the following way.

$$\underbrace{l}_{\text{function/gradient}} \qquad \underbrace{l \times n_{L+1}}_{\text{Jacobian}} \qquad \underbrace{\#\text{CG} \times l}_{\text{CG}} \qquad \underbrace{\#\text{line search} \times l}_{\text{line search}}.$$

In general, the number of line search steps is small, so the CG procedure is often the bottleneck. However, if the sub-sampled Hessian Newton method is applied, l is replaced by the size of the subset, $|S|$, for the cost in the Jacobian evaluation and CG. Then the bottleneck may be shifted to function/gradient evaluations. Note that the mini-batch setting in Section 3.5 for function and gradient evaluation reduces only memory consumption but not running time.

The discussion for the fully-connected layers is omitted because the result is similar to the convolutional layers.

6 EXPERIMENTS

The goal is to compare SG methods with the proposed subsampled Newton method for CNN. We consider a mini-batch SG with momentum [23] shown in Algorithm 2. Though other variants of SG methods such as AdaGrad and Adam have been proposed, it has been shown [e.g., 26, 31] that the mini-batch SG with momentum is a strong baseline.

Table 3. Summary of the data sets, where $a^1 \times b^1 \times d^1$ 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+1} is the number of classes.

Data set	$a^1 \times b^1 \times d^1$	l	l_t	n_{L+1}
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

6.1 Data Sets and Experimental Settings

We choose the following image data sets for experiments. All the data sets are publicly available[8] and the summary is in Table 3.

- MNIST: This data set, containing hand-written digits, is a widely used benchmark for data classification [14].
- SVHN: This data set consists of the colored images of house numbers [22].
- CIFAR10: This data set, containing colored images, is a commonly used classification benchmark [10].
- smallNORB: This data set is built for 3D object recognition [16]. 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. To reduce the training time, 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) Min-max normalization. That is, for each pixel of every image $Z^{1,i}$, we have

$$Z^{1,i}_{a,b,d} \leftarrow \frac{Z^{1,i}_{a,b,d} - \min}{\max - \min},$$

where max/min is the maximum/minimum value of all pixels in $Z^{1,i}$.

(2) Zero-centering. This is commonly applied before training CNN [11, 32]. That is, for every pixel in image $Z^{1,i}$, we have

$$Z^{1,i}_{a,b,d} \leftarrow Z^{1,i}_{a,b,d} - \text{mean}(Z^{1,:}_{a,b,d}),$$

where $\text{mean}(Z^{1,:}_{a,b,d})$ is the per-pixel mean value across all the training images.

We consider two simple CNN structures shown in Table 4. For the initialization, we follow [6] to set the weight values by multiplying random values from the $\mathcal{N}(0, 1)$ distribution and

$$\sqrt{\frac{2}{n^m_{\text{in}}}}, \text{ where } n^m_{\text{in}} = \begin{cases} d^m \times h^m \times h^m & \text{if } m \le L^c, \\ n_m & \text{otherwise.} \end{cases}$$

The bias vector in each layer is set to $\mathbf{0}$. In addition, to avoid the shrinkage of the output image in each convolutional layer, we do zero-padding to ensure

$$a^m_{\text{conv}} = a^m. \tag{89}$$

To determine the padding size for fulfilling (89), by substituting a^m_{pad} and a^m_{conv} into a^{in} and a^{out} in (4), we have

$$a^m_{\text{conv}} = \lfloor \frac{a^m_{\text{pad}} - h}{s} \rfloor + 1.$$

[8] See https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

ALGORITHM 2: Mini-batch stochastic gradient methods with momentum.

Given a regularization parameter $C > 0$, a learning rate $\eta \leftarrow \eta_0$, a momentum coefficient α, a decay factor γ, and an updating vector $\boldsymbol{v} \leftarrow \mathbf{0}$.

for $t = 1, \ldots,$ **do**

Choose a mini batch $S \subset \{1, \ldots, l\}$;

$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \left(\frac{\boldsymbol{\theta}}{C} + \frac{1}{|S|} \nabla_{\boldsymbol{\theta}} \sum_{i:i \in S} \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i}) \right)$;

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$;

$\eta \leftarrow \frac{\eta_0}{1+t\gamma}$;

end

If the padding size p indicates the number of zeros added on each side of the image, we have

$$a^m_{\text{pad}} = a^m + 2p.$$

With (89),

$$a^m = \lfloor \frac{2p + a^m - h}{s} \rfloor + 1.$$

Because $s = 1$ in Table 4, we can let the padding size be

$$p = \frac{h-1}{2}$$

so that (89) holds.

For convolutional layers, max pooling is used. Following [21], we consider the squared loss function shown in (26). For the activation function, the linear activation function at the last layer

$$\sigma(x) = x$$

is considered, while for all other layers (convolutional and fully-connected layers), the RELU activation function is used.

We use *MATLAB* to implement both Newton and stochastic gradient methods.[9] Because main operations are matrix-based and *MATLAB* is optimized for such operations, our implementation should be sufficiently efficient. We run the subsampled Newton method on a machine with 8 cores of Intel Core i7-6900K CPUs and 128GB memory. For SG, we use the same machine for timing comparisons, but use a GPU (Nvidia GeForce GTX 1080 Ti) otherwise to save the running time.

6.2 Test Accuracy and Convergence Speed

We begin with discussing parameters used. The value of C in (27) is set to $0.01l$.

For the Newton method, the CG procedure terminates if the following relative stopping condition holds or the number of CG iterations reaches a maximal number of iterations (denoted as $\text{CG}_{\max}$).

$$||(G + \lambda I)\boldsymbol{d} + \nabla f(\boldsymbol{\theta})|| \le \sigma ||\nabla f(\boldsymbol{\theta})||, \tag{90}$$

where $\sigma = 0.1$ and $\text{CG}_{\max} = 250$. For the implementation of the Levenberg-Marquardt method, we set the initial $\lambda = 1$ and (drop, boost, ρ_{upper}, ρ_{lower}) constants in (37) are (2/3, 3/2, 0.75, 0.25). In addition, the sampling rate for the Gauss-Newton matrix is set to 5%. We terminate the Newton method after 100 iterations.

[9] See https://github.com/cjlin1/simpleNN.

Table 4. Structure of convolutional neural networks. "conv" indicates a convolutional layer, "pool" indicates a pooling layer, and "full" indicates a fully-connected layer.

	3-layer CNN			5-layer CNN		
	filter size $(h^m \times h^m \times d^m)$	#filters (d^{m+1})	stride	filter size $(h^m \times h^m \times d^m)$	#filters (d^{m+1})	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	$3 \times 3 \times 64$	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
full 1	-	-	-	-	-	-

Table 5. Test accuracy by Newton and SG methods. We use five random seeds and report the mean test accuracy. The value within the parenthesis is the initial learning rate for SG, selected from a cross validation procedure on the training set.

	3-layer CNN		5-layer CNN	
	Newton	SG	Newton	SG
MNIST	99.27%	99.17% (0.003)	99.43%	99.42% (0.001)
SVHN	92.75%	93.06% (0.003)	94.28%	93.75% (0.003)
CIFAR10	78.32%	79.58% (0.003)	80.19%	79.63% (0.0003)
smallNORB	95.05%	95.06% (0.001)	95.30%	94.59% (0.001)

For stochastic gradient methods, we consider Algorithm 2 and select the initial learning rate η_0 from $\{0.003, 0.001, 0.0003, 0.0001\}$ by five-fold cross validation.[10] For other parameters, we set

$$|S| = 128, \alpha = 0.9, \gamma = 10^{-6}$$

and terminate the training process after $1,000$ epochs.

To have a fair comparison between SG and subsampled Newton methods, the following settings are the same for both approaches.

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

The first comparison, shown in Table 5, is on the test accuracy. We see that the test accuracy of the subsampled Newton method (with the 5% sampling rate) is comparable to that of SG. The performance of the SG method by using more layers is inferior to that by fewer layers. It seems overfitting occurs, so a tuning on SG's termination criterion may be needed.

[10] We use a stratified split of data in the cross validation procedure.

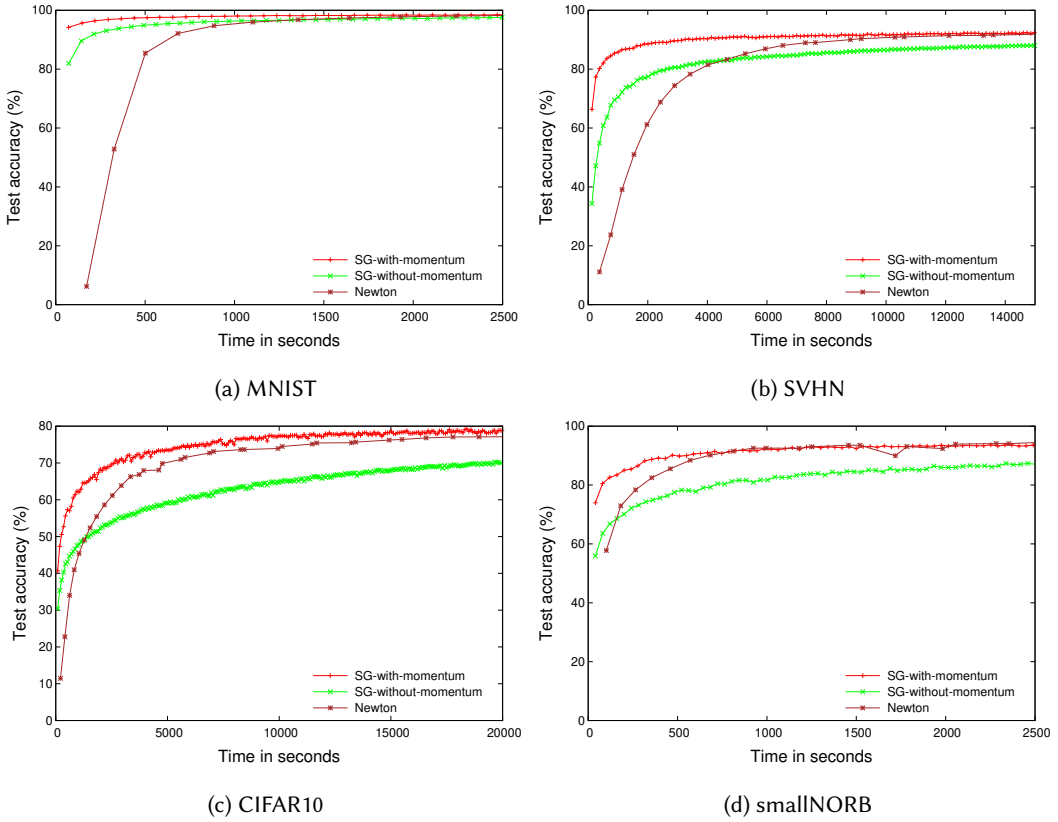

Fig. 3. A comparsion on the convergence speed. We present running time (in seconds) versus the test accuracy.

The next experiment is on the convergence speed. We consider the 3-layer CNN structure in Table 4 and compare the following three settings, where the first two are those used earlier to check the test accuracy.

- Subsampled Newton.
- SG with momentum.
- SG without momentum: This is the simple stochastic gradient method without using the momentum and the learning-rate decay.

In Figure 3, we present the result of

running time versus test accuracy.

We can observe that stochastic gradient methods, if under suitable settings (e.g., using the momentum) is faster than Newton to achieve the final test accuracy. This result is expected because a higher-order method like the Newton method is more expensive per iteration and those early iterations do not give good accuracy yet.

The above analysis seems to indicate that the subsampled Newton method is not efficient in comparison with SG. However, we note that before the training procedure to generate Figure 3, a cross validation procedure may be needed to select suitable parameters. For our experiments so far, no validation procedure is conducted for Newton, but we apply it for SG to select the initial

Table 6. A comparison between Newton and SG on the sensitivity to parameters. Each test accuracy is the average of five results by using the same initial solutions as in Table 5. For SG, in some situations not all the five initial weights lead to the convergence.

	Sampling rate (Newton)			Initial learning rate (SG)						
C	10%	5%	1%	0.1	0.03	0.01	0.003	0.001	0.0003	0.0001
$0.01l$	99.28%	99.27%	99.18%	9.82%	9.96%	10.31%	99.17%	99.22%	99.05%	98.82%
$0.1l$	99.22%	99.27%	99.06%	9.82%	9.96%	10.31%	99.12%	99.18%	98.91%	98.70%
$1l$	99.05%	99.15%	99.02%	9.82%	9.96%	10.31%	98.90%	99.03%	98.87%	98.69%

(a) MNIST.

	Sampling rate (Newton)			Initial learning rate (SG)						
C	10%	5%	1%	0.1	0.03	0.01	0.003	0.001	0.0003	0.0001
$0.01l$	92.79%	92.75%	92.21%	19.59%	19.59%	92.40%	93.06%	92.77%	92.35%	90.68%
$0.1l$	92.11%	92.29%	91.89%	19.59%	19.59%	90.80%	91.17%	91.63%	91.74%	90.32%
$1l$	91.20%	91.99%	91.81%	19.59%	19.59%	87.87%	89.81%	91.27%	91.57%	90.23%

(b) SVHN.

	Sampling rate (Newton)			Initial learning rate (SG)						
C	10%	5%	1%	0.1	0.03	0.01	0.003	0.001	0.0003	0.0001
$0.01l$	78.27%	78.32%	75.46%	10.00%	10.00%	63.10%	79.58%	79.20%	76.94%	71.55%
$0.1l$	74.19%	74.68%	73.24%	10.00%	10.00%	47.38%	71.29%	73.41%	73.99%	70.24%
$1l$	72.75%	73.54%	72.61%	10.00%	10.00%	55.45%	67.17%	71.28%	73.03%	69.88%

(c) CIFAR10.

	Sampling rate (Newton)			Initial learning rate (SG)						
C	10%	5%	1%	0.1	0.03	0.01	0.003	0.001	0.0003	0.0001
$0.01l$	94.94%	95.05%	94.66%	20.00%	64.73%	95.03%	95.08%	95.06%	94.87%	94.38%
$0.1l$	95.90%	95.41%	94.23%	20.00%	77.45%	95.78%	96.02%	95.64%	95.06%	94.23%
$1l$	94.89%	94.83%	93.97%	20.00%	47.35%	94.78%	94.88%	94.79%	94.66%	94.07%

(d) smallNORB.

learning rate. If we take the cross-validation procedure into consideration, the overall cost of SG is higher. Therefore, if we can confirm that SG is more sensitive to parameters, then with the better robustness, the Newton method can be practically viable. To this end, in Section 6.3 we investigate the robustness of the two methods.

6.3 Sensitivity of Newton and SG to Their Parameters

We still consider the 3-layer CNN in Table 4. The following parameters are checked.

- The regularization parameter C. This parameter appears in the objective function, so it must be selected regardless of the optimization method used.
- Size of the set S in the subsampled Newton method. We check different sampling ratios to select S from the whole training set.
- The initial learning rate for the stochastic gradient method.

Besides these parameters, all other settings are the same as those for generating Table 5. From results shown in Table 6 we can make the following observations.

- It is essential to find a suitable range of the initial learning rate for SG. If it is too large, the SG iterations diverge and give poor test accuracy. On the other hand, if the learning rate is too small, the convergence is very slow. That is, after 1,000 epochs, the test accuracy is still slowly increasing.
  Earlier comparisons on Newton and SG for fully-connected networks give similar observations (e.g., Figure 5 of [30]).
- For the subsampled Newton method, under the same C value, the test accuracy values are generally similar. However, in some cases, if only 1% data are used, the test accuracy is slightly worse. The reason is that without sufficient data, more iterations are needed to reach the final test accuracy.
- Test accuracy under different C values does not change dramatically though a selection procedure should be conducted to ensure that the chosen model gives the best validation accuracy.

We can conclude that the subsampled Newton method is less sensitive to parameters than the stochastic gradient method. First, because each Newton iteration is more expensive, a parameter change does not significantly affect the number of needed iterations. For example, for the CIFAR10 set, the accuracy reduction of changing the subset S from 5% to 1% of data is less than that of changing the initial learning rate of SG from 0.0003 to 0.0001. Second, the subsampled Newton method converges as long as enough iterations have been run. In contrast, a large initial learning rate can cause SG to diverge and return a useless model.

7 CONCLUSIONS

In this study, we establish all the building blocks of Newton methods for CNN. A simple and effective *MATLAB* implementation is developed for public use. Experiments show that Newton methods are less sensitive to parameters than stochastic gradient methods. Based on our results, it is possible to further enhance Newton methods for CNN.

ACKNOWLEDGMENTS

This work was supported by MOST of Taiwan via the grant 105-2218-E-002-033. The authors thank reviewers for constructive comments.

REFERENCES

[1] Aleksandar Botev, Hippolyt Ritter, and David Barber. 2017. Practical Gauss-Newton Optimisation for Deep Learning. In *Proceedings of the 34th International Conference on Machine Learning*. 557–565.
[2] Richard H. Byrd, Gillian M. Chin, Will Neveitt, and Jorge Nocedal. 2011. On the use of stochastic Hessian information in optimization methods for machine learning. *SIAM Journal on Optimization* 21, 3 (2011), 977–995.
[3] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew W Senior, Paul A Tucker, et al. 2012. Large Scale Distributed Deep Networks.. In *Advances in Neural Information Processing Systems (NIPS) 25*. 1223–1231.
[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software* 16, 1 (1990), 1–17.
[5] Roger Grosse and James Martens. 2016. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*. 573–582.
[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*.
[7] Xi He, Dheevatsa Mudigere, Mikhail Smelyanskiy, and Martin Takáč. 2017. Large Scale Distributed Hessian-Free Optimization for Deep Neural Network.
[8] Magnus Rudolph Hestenes and Eduard Stiefel. 1952. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Standards* 49, 1 (1952), 409–436.
[9] Ryan Kiros. 2013. Training neural networks with stochastic Hessian-free optimization. arXiv preprint arXiv:1301.3641.
[10] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images.* Technical Report. University of Toronto.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). 1097–1105.

[12] Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. 2011. On Optimization Methods for Deep Learning. In *Proceedings of the 28th International Conference on Machine Learning*. 265–272.

[13] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1, 4 (1989), 541–551.

[14] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (November 1998), 2278–2324. MNIST database available at http://yann.lecun.com/exdb/mnist/.

[15] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 1998. Efficient Backprop. In *Neural Networks: Tricks of the Trade*. Springer Verlag, 9–50.

[16] Yann LeCun, Fu Jie Huang, and Léon Bottou. 2004. 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*. 97–104.

[17] Kenneth Levenberg. 1944. A method for the solution of certain non-linear problems in least squares. *Quart. Appl. Math.* 2, 2 (1944), 164–168.

[18] Chih-Jen Lin, Ruby C. Weng, and S. Sathiya Keerthi. 2008. Trust region Newton method for large-scale logistic regression. *Journal of Machine Learning Research* 9 (2008), 627–650. http://www.csie.ntu.edu.tw/~cjlin/papers/logistic.pdf

[19] Donald W. Marquardt. 1963. An algorithm for least-squares estimation of nonlinear parameters. *J. Soc. Indust. Appl. Math.* 11, 2 (1963), 431–441.

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

[21] James Martens and Ilya Sutskever. 2012. Training deep and recurrent networks with Hessian-free optimization. In *Neural Networks: Tricks of the Trade*. Springer, 479–535.

[22] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

[23] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.

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

[25] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

[26] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*. 1139–1147.

[27] Andrea Vedaldi and Karel Lenc. 2015. MatConvNet: Convolutional neural networks for matlab. In *Proceedings of the 23rd ACM International Conference on Multimedia*. 689–692.

[28] Oriol Vinyals and Daniel Povey. 2012. Krylov subspace descent for deep learning. In *Proceedings of Artificial Intelligence and Statistics*. 1261–1268.

[29] Chien-Chih Wang, Chun-Heng Huang, and Chih-Jen Lin. 2015. Subsampled Hessian Newton Methods for Supervised Learning. *Neural Computation* 27, 8 (2015), 1766–1795. http://www.csie.ntu.edu.tw/~cjlin/papers/sub_hessian/sample_hessian.pdf

[30] Chien-Chih Wang, Kent-Loong Tan, Chun-Ting Chen, Yu-Hsiang Lin, S. Sathiya Keerthi, Dhruv Mahajan, Sellamanickam Sundararajan, and Chih-Jen Lin. 2018. Distributed Newton Methods for Deep Learning. *Neural Computation* 30, 6 (2018), 1673–1724. http://www.csie.ntu.edu.tw/~cjlin/papers/dnn/dsh.pdf

[31] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. 2017. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*. 4148–4158.

[32] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Proceedings of European Conference on Computer Vision*. 818–833.