AFFINE INVARIANCE IN MULTILAYER PERCEPTRON TRAINING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF UNIVERSITY OF TEXAS AT ARLINGTON
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Son Nguyen
August 2019

Abstract

Training methods for both shallow and deep neural nets are dominated by first order algorithms related to back propagation and conjugate gradient. However, these methods lack affine invariance so performance is damaged by nonzero input means, dependent inputs, dependent hidden units and the use of only one learning factor. This dissertation reviews affine invariance and shows how MLP training can be made partially affine invariant when Newton's method is used to train small numbers of MLP parameters. Several novel methods are proposed for scalable partially affine invariant MLP training. The potential application of the algorithm to deep learning is discussed. Ten-fold testing errors for several datasets show that the proposed algorithm outperforms back propagation and conjugate gradient, and that it scales far better than Levenberg-Marquardt.

Acknowledgments

My first thanks goes to my advisor, Dr. Michael Manry, I have been extremely lucky to work with him for the past six years. His constant motivating and guidance has shaped me from a student who knew nothing about neural networks to a capable researcher in deep learning.

I sincerely thank Dr. Victoria Chen, Dr. Ionnis Schizas, Dr. Daniel Levine and Dr. Ramtin Madani for their interest in my research and for taking time to serve on my comprehensive and dissertation committee.

I am thankful to my collaborators who I have had the pleasure to work with. This includes Kanishka Tyagi, Rohit Rawat, Yilong Hao, Soumitro Auddy and Parastoo Kheirkhah.

A special thanks to my group of Ph.D. students in Arlington, which includes Cuong Nguyen, Luan Nguyen, Loan Bui, Duong Le, Dung Tran, Nguyen Cao, Anh Nguyen and Thuong Nguyen. Your guys are really my family here, who always encourage and support me, wake me up every time I was frustrated.

I would like to thank the Vietname Education Fellowship (VEF) organization for giving me the great opportunity to study Ph.D. in the U.S., the opportunity that greatly changed my life to a much better direction. I would like to thank Hung Vo for introducing this fellowship to me and for being my mentor during the application process. Back home in Vietnam, I would like to thank my parents and my sister for unconditional support all of my decisions, for their endless love and Skype calls during all of my years studying.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Machine learning and neural networks

On summer 1956, a group of scientists gathered at Dartmouth to form a new branch of science which is Artificial Intelligent [55]. They had the ambition to make machines that have awareness and that can perform more complicated tasks. They did make some progress since carefully programmed computers can do simple arithmetic, play chess and perform many human-like tasks. While humans evolve and write code for machines, machines learned very little.

Later, in 1959, scientists realized that instead of teaching computers everything, it might be better to teach them to learn by themselves. Arthur Samuel is one of these scientists. His checker-playing program [75] was among the world's first successful self-learning programs that led to the term "Machine Learning." Machine learning is defined as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty[57]. There are many different machine learning methods including neural networks, support vector machines [11] and K-Nearest neighbor classifiers [18].

Neural networks were first proposed by Rosenblatt in the form of the perceptron [70] which is a linear network. After the development of the nonlinear multilayer perceptron (MLP) [53] and the backpropagation training algorithm [82] for it, the field of neural network developed greatly. Neural networks work well in many function approximation and classification applications such as pattern recognition [7, 7, 13, 22, 38, 56, 58, 63, 66, 74, 80], remote sensing [4, 33, 50], image processing [1, 8, 23], power load forecasting [42, 45, 72] and nonlinear estimation [46, 64, 65, 81]. The multilayer perceptron (MLP) is the most widely used type of neural network [77]. Neural networks are well known for their universal approximation property [31], which means that they can approximate any continuous function, given enough hidden units. Study in [67] shows that if a MLP's parameters are chosen to minimize a squared-error cost function, the outputs estimate the conditional probabilities of the Bayesian posteriori.

However, the *no free lunch* theorem[84] states that no machine learning algorithm can beat random guessing over all possible functions that need to be learned. Fortunately, these results hold only when one averages over all possible generating distributions [26], while the goal of machine learning is trying to learn some specific functions of some particular distributions that we care about [5]. The straightforward way to do machine learning is that, we first collect data as much as possible from the domain that our desired algorithm was supposed to learn, the data can be both label and unlabeled. Then we train models and algorithms trying to learn the distributions of **collected** data. That explains why neural networks still perform so well, regardless of the *no free lunch* theorem, and preparing data is such a crucial task in training any machine learning models.

There are many different methods for machine learning. In the next chapter, we review a few of those.

1.2 Methods and reference

Boosting is a powerful technique for combining multiple 'base' [9] classifiers to produce a committee whose performance can be significantly better than that of any of the base classifier. The most widely used boosting approach is AdaBoost [24] which stands for "adaptive boosting". Adaboost collects all "base" classifiers' decisions, then uses training to adjust the weights given to each of them. The idea is that, better "base" classifiers should have stronger weights while worse "base" classifiers should have smaller weights. Adaboost can have a much better performance than the best of the "base" classifiers.

The support vector machine (SVM) [11] was a popular and dominant classification method before the arrival of deep learning. SVMs solve the problem of maximizing the distance or margin separating two classes in input space. The problem turns out to be convex, and any local solution is also a global solution [9]. Using kernel methods such as RBF can boost the performance of the SVM [32].

Deep learning is a set of machine learning models that allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts [26]. The graph of such model is deep with many layers, leading to the name "deep learning". The layers are neural networks that can be trained individually or simultaneously. The current deep learning renaissance began when Hinton [30] demonstrated that a neural network could outperform the RBF kernel SVM on the MNIST benchmark [26]. Deep learning has many successful applications, in image recognition [37], speech recognition [54], natural language processing [17].

Even though there have been many successes, neural nets still have many problems. First, training is still heavily based on first order backpropagation which is slow, easily falls into local minima, and is not affine invariant. Second, current neural nets and deep learning models have an excessive number of parameters that need to be manually adjusted. Third, deep learning works well, but there is still no convincing theory for it.

1.3 Dissertation organization

This dissertation provides solutions to some of the remaining neural network problems are described. The second chapter describes our notation for the MLP, then reviews some well known first-order training methods. In the third chapter, a brief review of second-order training methods is given. The fourth chapter defines affine invariance and partial affine invariance (PAI). Two algorithms with PAI are demonstrated. In the fifth chapter, relevant problems are given and approaches for solving them are listed. Preliminary work is described in chapter six. Conclusions are given in chapter seven.

Chapter 2

Structure and notation

This chapter describes architecture and our notation of two popular neural network models, the the regular multilayer perceptron (MLPs) and the convolutional neural networks (CNNs).

2.1 MLP structure and notation

Figure (2.1) illustrates the structure of a single hidden layer MLP having an input layer, a hidden layer and an output layer. We denote the number of hidden units by N_h and the number of outputs by M. In order to handle hidden and output unit thresholds, the N-dimensional input column vector $\mathbf{x}_\mathrm{p}$ is augmented by an extra element $x_\mathrm{p}(N+1) = 1$. Here, the input vectors are $(N+1)$-dimensional, and the desired output column vectors $\mathbf{t}_\mathrm{p}$ are M-dimensional. The training data $\{\mathbf{x}_\mathrm{p}, \mathbf{t}_\mathrm{p}\}$ has N_v pairs where $p \in \{1, 2, \ldots, N_v\}$.

For the pth training pattern, the N_h dimensional net function vector in the hidden layer is given by:

$$\mathbf{n}_\mathrm{p} = \mathbf{W}_\mathrm{i} \cdot \mathbf{x}_\mathrm{p} \tag{2.1}$$

where $\mathbf{W}_\mathrm{i}$ is a $N_h \times (N+1)$ input weight matrix and the corresponding $N_h + 1$ dimensional hidden unit activation vector $\mathbf{o}_\mathrm{p}$ has elements $o_\mathrm{p}(k) = f(n_\mathrm{p}(k))$ where $f(n)$ is a sigmoid function defined as

$$f(n) = \frac{1}{1 + e^{-n}} \tag{2.2}$$

Only hidden units have sigmoid activation function. The hidden unit activation vector

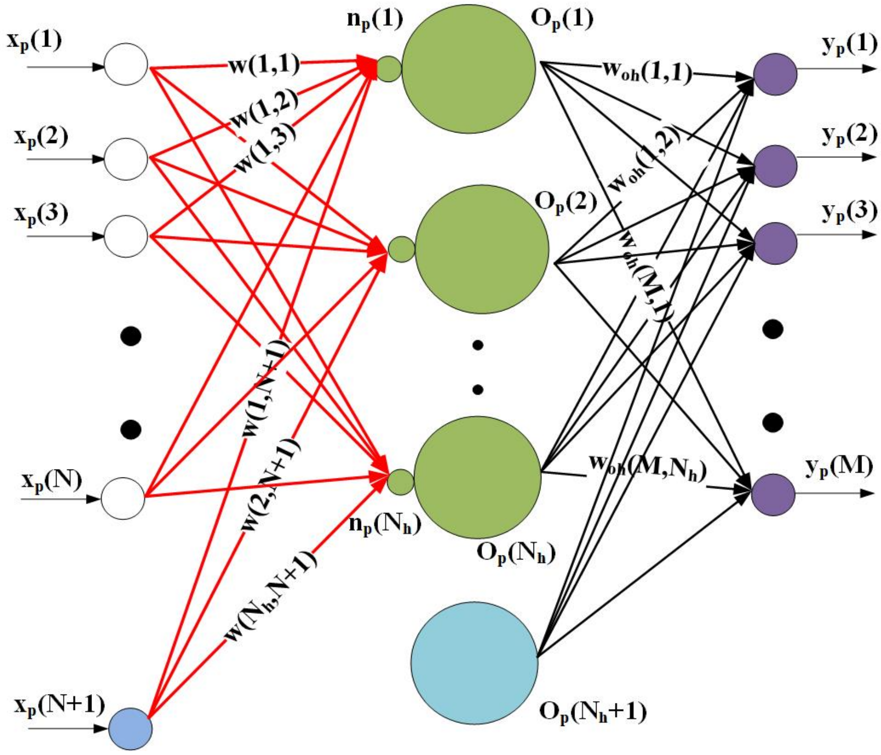

Figure 2.1: Illustration of a Multilayer Perceptron

$\mathbf{o}_{\mathrm{p}}$ also has a threshold $o_{\mathrm{p}}(N_h+1)=1$. The M dimensional output vector for the pth training pattern is

$$\mathbf{y}_{\mathrm{p}} = \mathbf{W}_{\mathrm{o}} \cdot \mathbf{o}_{\mathrm{p}} \tag{2.3}$$

where $\mathbf{W}_{\mathrm{o}} \in R^{M\times(N_h+1)}$ contains weights from hidden units to the outputs. Some quantities that we define for convenience are the number of network weights $N_w = N_h(N+1)+M(N_h+1)$ and the number of basis functions $N_u = N_h+1$.

Neural network trainings methods adjust the pre-selected weights to reduce a cost function. The pre-selected weights are often randomly initialized. In this proposal, we use the Mean Square Error (MSE) as the cost function or objective function, which we will often abbreviate as E. The MSE over a training set, called the training error, is given by

$$E = \frac{1}{N_v}\sum_{\mathrm{p}=1}^{N_v}\sum_{\mathrm{i}=1}^{M}[t_{\mathrm{p}}(\mathrm{i})-y_{\mathrm{p}}(\mathrm{i})]^2 \tag{2.4}$$

where $y_{\mathrm{p}}(\mathrm{i})$ is the ith element of $\mathbf{y}_{\mathrm{p}}$ given in (3). Clearly, $\mathbf{y}_{\mathrm{p}}$ is a function of weight matrices $\mathbf{W}_{\mathrm{i}}$, $\mathbf{W}_{\mathrm{o}}$ or $\mathbf{w}$ where

$$\mathbf{w} = vec(\mathbf{W}_{\mathrm{i}}, \mathbf{W}_{\mathrm{o}}) \tag{2.5}$$

and where the *vec()* operation arranges one or more matrices into a column vector.

2.2 Convolutional neural networks

Convolution neural networks (CNNs) are a specialized kind of neural network for processing input data that has a known grid-like topology [26], as in images. It is considered to be the one of the first deep models that work well. It is also the first deep architecture used in a commercial product[40]. In fact, convolution neural networks do not need to be deep to work well in some tasks. Figure (2.2) illustrates the structure of a simple convolution neural network, with one 32 filters-convolutional layer, one max-pooling layer, and one fully connected layer.

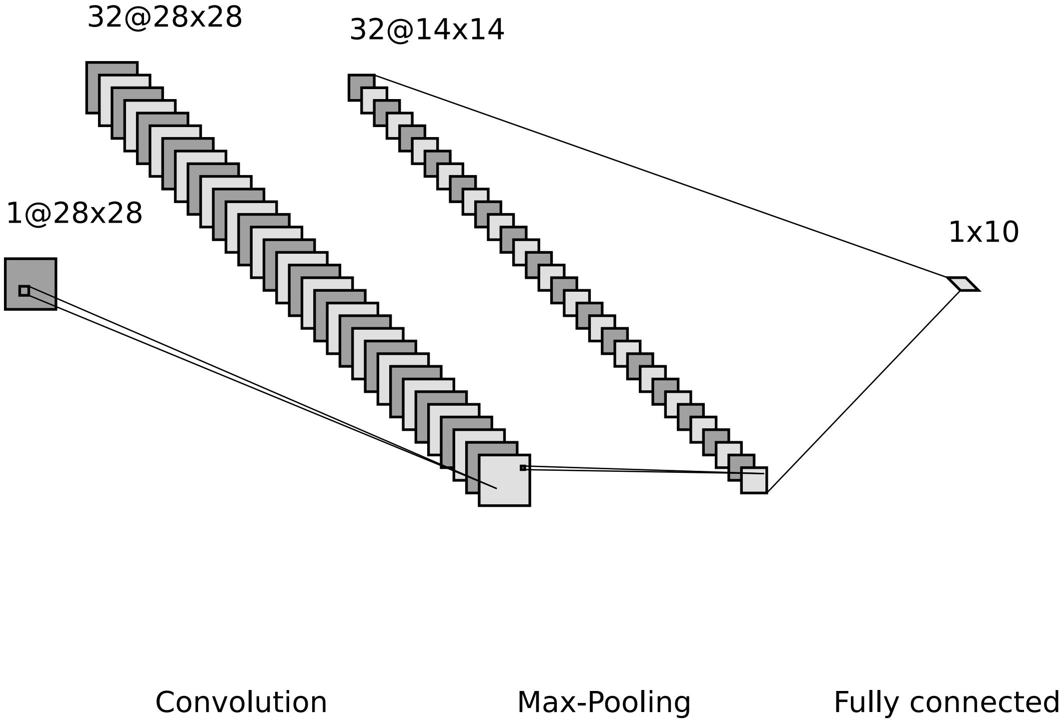

Figure 2.2: CNNs with 1 convolution layer, 1 max-pooling and one fully connected layer

Firstly, each 28×28 binary image is convolved with 32 5×5 filters to create 32 28×28 images. The max-pooling layer reduces size of each image to 14×14.

$$\mathbf{O}_p(k) = pool\bigg(\mathbf{X}_p * \mathbf{W}_i(k)\bigg) \tag{2.6}$$

where $\mathbf{X}_p$ is the p^{th} image of the training data, $\mathbf{W}_i(k)$ is the weights of the k^{th} filter, and (*) is the convolution operation. The $pool()$ is the sub-sampling operation[26]. There are different types of pooling such as $max - pooling$, $average - pooling$..., in this dissertation, we use 2×2 $max - pooling$ which acts as a 2×2 sub-sampling filter and output the max value in the 4 elements. Then, each image is flatten out to be a vector, before plugging in the activation function and then the fully connected layer.

$$\mathbf{y}_p = \mathbf{W_o} \cdot f(vec(\mathbf{O}_p)) \tag{2.7}$$

where the $vec()$ operation flattens all pixels of $\mathbf{O}_p$ into a vector, and $f()$ is the activation function applied element wise to the input matrix. We use $ReLU()$ activation in this dissertation, but all described training methods will work with any activation functions such as $sigmoid()$ or $tanh()$. The cost function is still the mean square error (MSE) as in equation (2.4). We apply the idea of output-reset as described in [79] to make the MSE work well for classification tasks.

Chapter 3

Neural network training methods

Training methods for MLPs can be classified as first or second order training methods. First order training methods are scalable and are heavily used in deep learning, while second order method are not popular in the machine learning community due to their lack of scalability. Figure 3.1 presents an algorithm tree with some typical first and second order methods.

In this section, we review MLP structure and notation. Widely used, scalable first order neural network training methods are described.

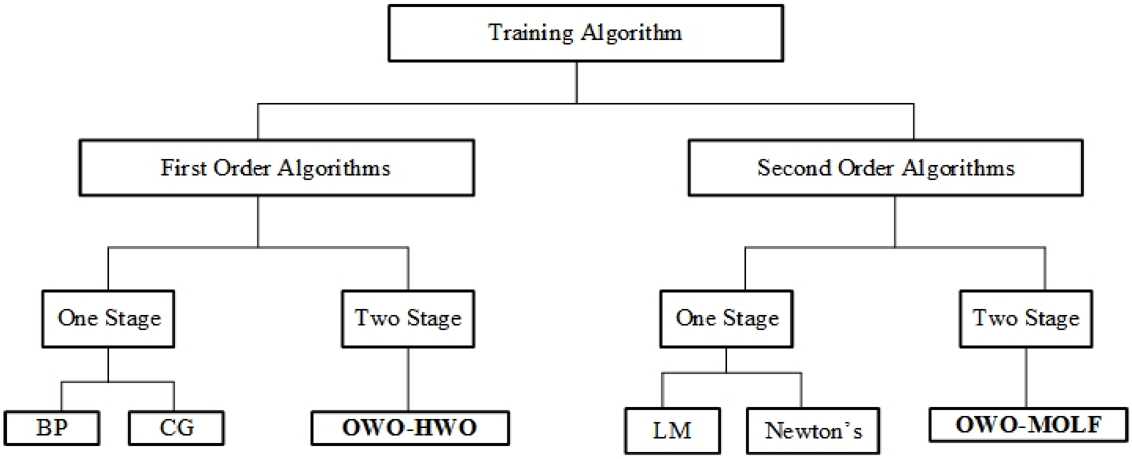

Figure 3.1: An algorithms tree with some typical training methods

3.1 First order training of shallow neural networks

First order training methods are algorithms that use only first order derivative of the cost function with respect to the weights. So basically, they only use gradient information to perform training with make them scalable and were used heavily, especially in deep learning.

3.1.1 Steepest descent

The steepest descent algorithm [9, 12] is the first on the left side of the training algorithm tree in Fig 3.1. It allows the information from the cost function to flow backward through the network in order to compute the gradient [26], which is the first negative derivative of the cost function we want to minimize with respect to the weights as

$$\mathbf{g} = -\frac{\partial E}{\partial \mathbf{w}} \tag{3.1}$$

The recursive method for calculating the gradient is called backpropagation [73]. The gradient is used to update the weights as.

$$\mathbf{w} \leftarrow \mathbf{w} + z \cdot \mathbf{g} \tag{3.2}$$

where z is the step size or learning rate, and is usually a small positive constant. Steepest descent does decrease the cost function but really slowly. A line search can be used to find the learning rate that gives the biggest decrease in the cost function but it is not very effective. There are several different ways to manipulate the weight update based upon gradient, each resulting in a different training method. The learning rate can also be determined using the Gaussian-Newton method [41] as

$$z = -\frac{\frac{\partial E}{\partial z}}{\frac{\partial^2 E}{\partial z^2}} \tag{3.3}$$

The steepest descent algorithm is summarized as follows

Algorithm 1 Steepest descent algorithm

```
Initialize w, $N_{it}$ , it$\leftarrow$ 0
while it $<$ $N_{it}$ do
    Calculate g
    Compute z from equation (3.3)
    Update w as $\mathbf{w} \leftarrow \mathbf{w} + z \cdot \mathbf{g}$
    it $\leftarrow$ it + 1
end while
```

3.1.2 Conjugate gradient algorithm

Conjugate gradient [25], the next algorithm on the algorithm tree, addresses the slow convergence problem of steepest descent by removing the influence of previous iterations' gradients from the current ones, creating a new search direction which is conjugate to the previous ones. Instead of using the gradient to update the weights, conjugate gradient uses the vector

$$\mathbf{p} \leftarrow -\mathbf{g} + \beta \cdot \mathbf{p} \tag{3.4}$$

where β is calculated as the ratio of the gradient's energies from two consecutive iterations. Then the weights are updated as

$$\mathbf{w} \leftarrow \mathbf{w} + \lambda \cdot \mathbf{p} \tag{3.5}$$

where λ is the learning rate, which calculated to maximize the objective function decrease.

Conjugate gradient (CG) is guaranteed to converge to a global minimum in N_w iterations if the cost function is quadratic [12], where N_w is the number of unknowns. Conjugate gradient performs better than steepest descent even when the cost function is non-quadratic. Since there is no Hessian involved, CG is scalable and widely used in training MLPs for big datasets. The conjugate gradient can be summarized as follows

3.1.3 Whitening using Hidden Weight Optimization

Hidden weight optimization (HWO) [86], which is equivalent to applying whitening [9] to the input data, is an improvement over regular back propagation. HWO still

Algorithm 2 Conjugate gradient algorithm

```
Initialize $\mathbf{w}$, $N_{it}$ , it $\leftarrow 0$
while it $< N_{it}$ do
    Calculate $\mathbf{p}$ from $\mathbf{g}$
    Compute z from equation (3.3)
    Update $\mathbf{w}$ as $\mathbf{w} \leftarrow \mathbf{w} + \mathrm{z} \cdot \mathbf{p}$
    it $\leftarrow$ it $+ 1$
end while
```

calculates the negative input gradient from input weight matrix $\mathbf{W_i}$ as

$$\mathbf{G_i} = -\frac{\partial E}{\partial \mathbf{W_i}} \tag{3.6}$$

HWO then finds improved input gradient matrix as $\mathbf{G}_{\mathbf{i}\_\mathrm{hwo}}$ by solving the following linear equations.

$$\mathbf{R}_{\mathrm{i}} \cdot \mathbf{G}_{\mathbf{i}\_\mathrm{hwo}} = \mathbf{G_i} \tag{3.7}$$

where $\mathbf{R}_{\mathrm{i}}$ is the input autocorrelation matrix defined as

$$\mathbf{R}_{\mathrm{i}} = \frac{1}{N_v} \sum_{\mathrm{p}=1}^{N_v} \mathbf{x}_{\mathrm{p}} \cdot \mathbf{x}_{\mathrm{p}}^T \tag{3.8}$$

We then use $\mathbf{G}_{\mathbf{i}\_\mathrm{hwo}}$ to update the input weight matrix as

$$\mathbf{W_i} \leftarrow \mathbf{W_i} + z \cdot \mathbf{G}_{\mathbf{i}\_\mathrm{hwo}} \tag{3.9}$$

where z is the learning rate in regular back propagation algorithm of part B. Then we update output weight using output weight optimization (OWO) [51] which is a second order method and will be explained in the next section. The advantage of HWO is that, while whitening is a pre-processing method which is apply only for input raw data, HWO can apply to training any layer in the neural network.

Hidden weight optimization (HWO) [86] only use HWO for input weights which has similar effect as whitening input data. We can do the same to improve the weight change matrix of output weights. Similar to changing input weight gradient, the output

Algorithm 3 Hidden weights optimization algorithm

```
Initialize $\mathbf{W_i}$, $\mathbf{W_o}$, $N_{it}$ , it$\leftarrow$ 0
while it $< N_{it}$ do
    Calculate gradient matrices $\mathbf{G_i}$
    Update gradient matrices to HWO by using equation (3.7)
    Compute the learning rate $z$ using Newton's method as equation (3.3)
    Update input weight matrix $\mathbf{W_i} \leftarrow \mathbf{W_i} + z \cdot \mathbf{G_{i\_hwo}}$
    Update output weight matrix using OWO
    it $\leftarrow$ it + 1
end while
```

weight gradient can be updated as

$$\mathbf{G_{o\_hwo}} = \mathbf{G_o} \cdot \mathbf{R}_{\mathrm{o}}^{-1} \tag{3.10}$$

where $\mathbf{G_o}$ is the output weight gradient matrix and $\mathbf{R}_{\mathrm{o}}$ is the output autocorrelation matrix defined as

$$\mathbf{R}_{\mathrm{o}} = \frac{1}{N_v} \sum_{p=1}^{N_v} \mathbf{O}_{\mathrm{p}} \cdot \mathbf{O}_{\mathrm{p}}^T \tag{3.11}$$

then we can use $\mathbf{G_{o\_hwo}}$ to update the output weight matrix.

3.2 Second order training methods

Second order training methods are algorithms that used both first and second order derivative to perform learning which make them have a kind of affine invariance. In this section we investigate affine invariance properties and then review some well-known second order training methods.

3.2.1 Affine invariance

Weight initialization has been widely recognized as one of the most effective critical steps in the training of neural networks [19, 20, 52, 60, 85]. In general, if we have a different initial set of weights, the training will return different results. To address this problem, we investigate the affine invariance property of MLP training via equivalent network theory.

Let $E(\mathbf{w})$ denote the the MLP error in term of $\mathbf{w}$

Definition 1 *Two networks are strictly equivalent if* $E(\mathbf{w}) = E(\mathbf{T}\mathbf{w}')$ *where* $\mathbf{w} = \mathbf{T}\mathbf{w}'$ *for a square nonsingular matrix* $\mathbf{T}$.

In other words, two networks are equivalent if the weights $\mathbf{w}$ of one network are replaced by $\mathbf{T}\mathbf{w}'$ in the other network. As a result, there are infinitely many equivalent networks, each with a different set of weights $\mathbf{w}'$. Affine invariance in neural networks can be defined as follows [12].

Definition 2 *If two strictly equivalent networks are formed whose objective functions satisfy* $E(\mathbf{w}) = E(\mathbf{T}\mathbf{w}')$ *with* $\mathbf{w} = \mathbf{T}\mathbf{w}'$, *and an iteration of an optimization method yields* $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{d}$ *and* $\mathbf{w}' \leftarrow \mathbf{w}' + \mathbf{d}'$, *the training method is affine invariant if* $\mathbf{d} = \mathbf{T}\mathbf{d}'$ *for every* $N_w \times N_w$ *nonsingular matrix* $\mathbf{T}$.

In other words, two equivalent networks will be still equivalent after one iteration of an affine invariant training method. So all initially equivalent networks will have the same training performance if the training method is affine invariant. As a result, affine invariant training is more independent of weight initialization than non-affine invariant methods. Unfortunately, first order MLP training methods lack affine invariance.

Lemma 3.2.1 *Steepest descent with constant learning rate is not affine invariant.*

Proof

Consider two strictly equivalent networks which satisfy $E(\mathbf{w}) = E(\mathbf{T}\mathbf{w}')$ where $\mathbf{w} = \mathbf{T}\mathbf{w}'$, so

$$\frac{\partial \mathbf{w}}{\partial \mathbf{w}'} = \mathbf{T} \tag{3.12}$$

Applying the chain rule to the derivative of the second network

$$\begin{aligned} \mathbf{g}' &= -\frac{\partial E}{\partial \mathbf{w}'} \\ &= -\left(\frac{\partial \mathbf{w}}{\partial \mathbf{w}'}\right)^T \cdot \frac{\partial E}{\partial \mathbf{w}} \\ &= \mathbf{T}^T \mathbf{g} \end{aligned} \tag{3.13}$$

As in defintion 2, the training method is affine invariance if the weight change vectors

$\lambda\mathbf{g}$ and $\lambda\mathbf{g}'$ satisfy

$$\begin{aligned} E(\mathbf{w}+\lambda\mathbf{g}) &= E(\mathbf{T}(\mathbf{w}'+\lambda\mathbf{g}')) \\ &= E(\mathbf{w}+\lambda\mathbf{T}\mathbf{g}') \\ &= E(\mathbf{w}+\lambda\mathbf{T}\cdot\mathbf{T}^T\mathbf{g}) \end{aligned} \tag{3.14}$$

which only happens if $\mathbf{T}$ is an orthogonal matrix. Clearly, steepest descent lacks affine invariance. Conjugate gradient also has the same limitation.

Lemma 3.2.2 *Conjugate gradient is not affine invariant.*

Proof

The first iteration of CG is actually steepest descent ,so CG also lacks affine invariance

Definition 3. *If a training algorithm satisfies the conditions of affine invariance in Definition 2 except that* $\boldsymbol{T}$ *has less than* N_w^2 *free parameters, the training algorithm is partially affine invariant (PAI).*

Different PAI algorithms have different number of free parameters in its $\mathbf{T}$ matrix. In order to evaluate affine invariant properties of PAI algorithms, we define PAI order as follows

Definition 4. *If a training algorithm satisfies the conditions of partially affine invariance in Definition 3, then the PAI order of this method is the total number of free parameters in the* $\mathbf{T}$ *matrix divided by total number of elements in the* $\mathbf{T}$ *matrix.*

We have been using Newton's method to develop different training algorithms that have different orders of affine invariance. The following table gives a sense of how much affine invariance these methods have.

Table 3.1: Partial affine invariance orders for various training methods

Algorithm names	PAI order
Back Propagation BP2	$4/N_w^2$
Multiple Optimal Learning Factors [48]	$(N_h+1)^2/N_w^2$
Optimal Input Gains [3]	$(N+1)^2/N_w^2$
Partially affine invariance BP [61]	$((N+1)^2+(N_h+1)^2)/N_w^2$
OWO-Newton	$[(N_h\cdot(N+1))^2+(M\cdot(N_h+1))^2]/N_w^2$

In Fig. 3.2, we show steepest descent's error versus iteration number curves for two initially equivalent MLPs. In Fig. 3.3, we perform the same experiment for CG. The training curves start at the same point but diverge due to a lack of affine invariance.

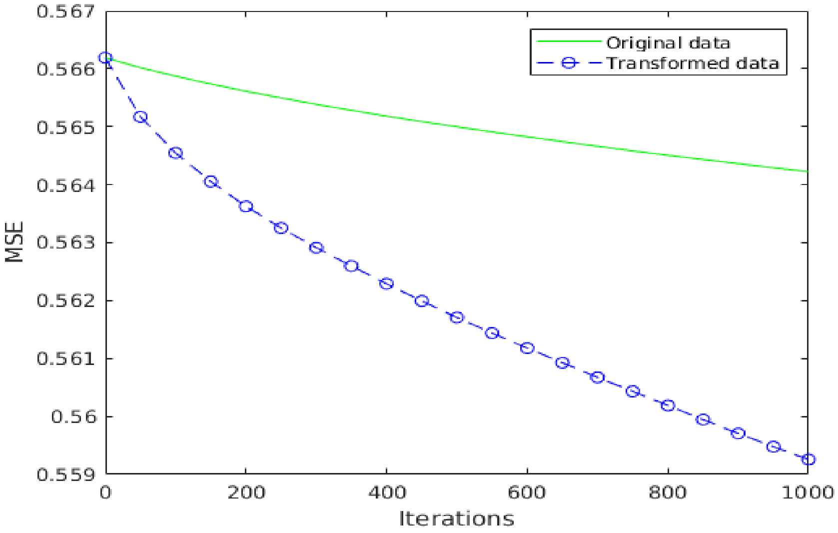

Figure 3.2: Steepest descent diverges in equivalent networks

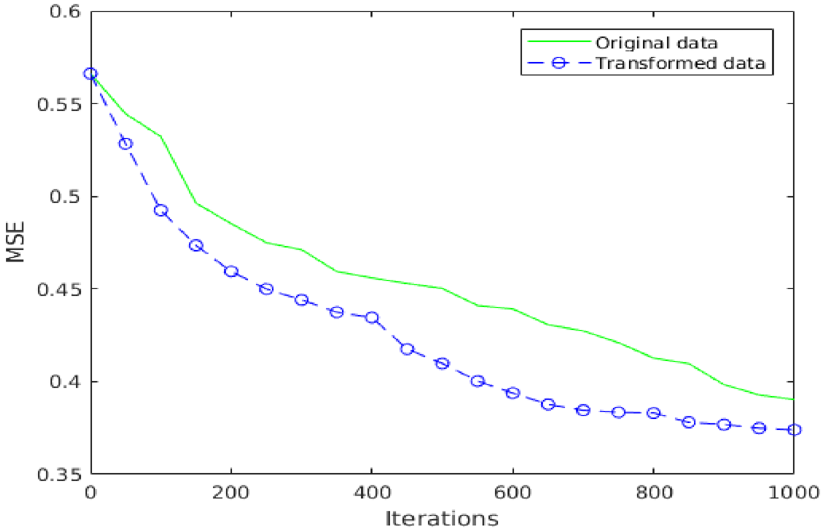

Figure 3.3: Conjugate gradient diverges in equivalent networks

3.2.2 Newton's Algorithm

Newton's algorithm is the basis of a number of second order optimization algorithms including Levenberg-Marquardt [44] and BFGS [62]. In this subsection, we review Newton's method.

Mclaurin's second order expansion of the cost function $E(\mathbf{d})$ is

$$E(\mathbf{d}) = E(0) - \mathbf{d}^T \frac{\partial E}{\partial \mathbf{d}} + \frac{1}{2}\mathbf{d}^T \frac{\partial^2 E}{\partial \mathbf{d}^2}\mathbf{d} \tag{3.15}$$

where $\mathbf{d}$ is the weight change column vector, which has the same dimension as $\mathbf{w}$. In order to minimize $E(\mathbf{d})$, we take derivative of the expansion with respect to $\mathbf{d}$ and set it to 0, yielding

$$-\frac{\partial E}{\partial \mathbf{d}} + \frac{\partial^2 E}{\partial \mathbf{d}^2}\mathbf{d} = 0 \tag{3.16}$$

or

$$-\mathbf{g} + \mathbf{H} \cdot \mathbf{d} = 0 \tag{3.17}$$

where $\mathbf{H}$ is the Hessian matrix and $\mathbf{g}$ is the gradient vector which have elements

$$h(m,n) = \frac{\partial^2 E}{\partial w(m) \partial w(n)} \tag{3.18}$$

$$g(n) = -\frac{\partial E}{\partial w(n)} \tag{3.19}$$

and where $w(n)$ is an element of the weight vector $\mathbf{w}$ in equation (2.5) for $1 \leq n \leq N_w$. The Newton direction vector $\mathbf{d}$ is calculated by solving the set of linear equations

$$\mathbf{Hd} = \mathbf{g} \tag{3.20}$$

Update $\mathbf{w}$ with the direction vector $\mathbf{d}$ as

$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{d} \tag{3.21}$$

Newton's algorithm can be summarized in algorithm 4. It is well known that Newton's algorithm has quadratic convergence and is affine invariant. Which means that it satisfied definition 2.

Algorithm 4 Newton's algorithm

```
Initialize $\mathbf{w}$, $N_{it}$ , it$\leftarrow 0$
while it $< N_{it}$ do
    Calculate $\mathbf{g}$ and $\mathbf{H}$ from equation (3.18) and equation (3.19)
    Compute $\mathbf{d}$ from equation (3.20)
    Update $\mathbf{w}$ as $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{d}$
    it $\leftarrow$ it $+ 1$
end while
```

Lemma 3.2.3 *Newton's method is affine invariant.*

Proof

The following proof is similar to the proof in [12].

Consider two strictly equivalent networks satisfying $E(\mathbf{w}) = E(\mathbf{T}\mathbf{w}')$ where $\mathbf{w} = \mathbf{T}\mathbf{w}'$. We have $\mathbf{g} = (\mathbf{T}^{-1})^T \cdot \mathbf{g}'$ as in equation (3.13). Similar, we have

$$\mathbf{H} = (\mathbf{T}^{-1})^T \mathbf{H}' \mathbf{T}^{-1} \tag{3.22}$$

where $\mathbf{H}'$ is the Hessian of the second network. The weight change becomes

$$\begin{aligned} \mathbf{d} &= \mathbf{H}^{-1}\mathbf{g} = \mathbf{T}\mathbf{H}'^{-1}\mathbf{T}^T \cdot (\mathbf{T}^{-1})^T \mathbf{g}' = \mathbf{T}\mathbf{H}'^{-1} \cdot \mathbf{g}' \\ &= \mathbf{T} \cdot \mathbf{d}' \end{aligned} \tag{3.23}$$

with any non-singular $\mathbf{T}$ matrix, so Newton's method is affine invariant.

3.2.3 Output Weight Optimization

Assume that the input weight matrix $\mathbf{W}_\mathrm{i}$ has been determined in some fashion, usually by random initialization. One method used to find the output weights is the output weight optimization (OWO) algorithm [6, 51, 76] which minimizes the MSE from equation (2.4) with respect to the output weight matrix $\mathbf{W}_\mathrm{o}$. Taking the derivative of E with respect to $\mathbf{W}_\mathrm{o}$ we have

$$\frac{\partial E}{\partial \mathbf{W}_\mathrm{o}} = -\frac{2}{N_v}[\mathbf{T}_\mathrm{o} - \mathbf{O}\mathbf{W}_o^T]^T \mathbf{O} \tag{3.24}$$

where $\mathbf{T}_{\mathrm{o}} \in R^{N_v \times M}$ is target output matrix

$$\mathbf{T}_o = \begin{bmatrix} \mathbf{t}_1^T \\ \mathbf{t}_2^T \\ \cdots \\ \mathbf{t}_{N_v}^T \end{bmatrix} \tag{3.25}$$

and $\mathbf{O} \in R^{N_v \times (N_h+1)}$ is augmented hidden activation with a constant 1

$$\mathbf{o}_p = [1 \quad f(\mathbf{n}_{\mathrm{p}})] \tag{3.26}$$

where $1 \leq p \leq N_v$ and $f(n)$ is an activation function such as rectify linear unit (ReLU) applied element wise to the each element of n defined as

$$f(n) = \begin{cases} n, & n \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.27}$$

The outputs of activation $f(n)$ have the same dimension as the inputs. If input n is a scalar, then $f(n)$ is a scalar. Alternatively, if input $\mathbf{n}$ is a vector, then $f(\mathbf{n})$ is also a vector with the same dimension as $\mathbf{n}$. The $\mathbf{R}$ and $\mathbf{C}$ matrices are calculated as

$$\mathbf{R} = \frac{1}{N_v}\mathbf{O}^T\mathbf{O} \tag{3.28}$$

$$\mathbf{C} = \frac{1}{N_v}\mathbf{O}^T\mathbf{T}_{\mathrm{o}} \tag{3.29}$$

Equating the derivative in equation (3.24) to zero we have

$$\mathbf{R}\mathbf{W}_{\mathrm{o}}^T = \mathbf{C} \tag{3.30}$$

where $\mathbf{W}_{\mathrm{o}}$ is the solution to M sets of N_u equations in N_u unknowns. These equations can be solved using any number of methods, but special care must be taken when $\mathbf{R}$ is ill-conditioned. An algorithm which solves equation (3.30) for $\mathbf{W}_{\mathrm{o}}$ can be denoted as output weight optimization (OWO) [6, 51, 76].

OWO is Newton's algorithm for the output weights. We update the output weights as $\mathbf{W}_o \leftarrow \mathbf{W}_o + \mathbf{D}$. We use Newton's algorithm to calculate $\mathbf{D}$. The Hessian and gradient of equation (2.4) with respect to output weights $\mathbf{W}_o$ are $\mathbf{2R}$ and $2\mathbf{R}\mathbf{W}_o^T - 2\mathbf{C}$ respectively. The weight update becomes

$$\begin{aligned} \mathbf{W}_o &\leftarrow \mathbf{W}_o - \mathbf{H}^{-1}\mathbf{g} \\ &= \mathbf{W}_o + (2\mathbf{R})^{-1}2(\mathbf{C} - \mathbf{R}\mathbf{W}_o^T) \\ &= \mathbf{W}_o + \mathbf{R}^{-1}\mathbf{C} - \mathbf{R}^{-1}\mathbf{R}\mathbf{W}_o^T \\ &= \mathbf{R}^{-1}\mathbf{C} \end{aligned} \tag{3.31}$$

This is the same solution as for OWO given in equation (3.30)

3.2.4 Levenberg-Marquardt algorithm

The LM algorithm [44] is a combination of first and second order training methods. Since Newton's Hessian matrix is often ill-conditioned or singular [9, 26], inverting them is problem. Levenberg-Marquard gave a solution by adding constant terms to the Hessian's diagonal as

$$\mathbf{H}_{LM} = \mathbf{H} + \lambda \cdot \mathbf{I} \tag{3.32}$$

where $\mathbf{I}$ is an identity matrix which has the same dimension as $\mathbf{H}$ and λ is the constant. Then the LM's Hessian matrix is nonsingular and its direction vector can be calculated by solving a set of equations

$$\mathbf{H}_{LM}\mathbf{d}_{LM} = \mathbf{g} \tag{3.33}$$

The λ constant is a trade-off value between first and second order for LM method. If λ is small and close to zero, then it does not have much effect on Hessian's matrix and LM method approaches Newton's method. In the opposite, if λ is big enough, it makes the Hessian matrix becomes similar to the identity matrix, then the weights change vector $\mathbf{d}_{LM}$ is close to the gradient $\mathbf{g}$; as a result, LM method approaches steepest descent method. Including steepest descent method makes LM method also lacks of affine invariance. In fact, there is other way to solve the non singular Hessian problem but still maintain affine invariance properties as study in [69]. The LM algorithm can be summarized as algorithm 5.

Most training algorithms, including BP and LM, lack affine invariance. Since CG begins with an iteration of steepest descent, it lacks affine invariance. Newton's algorithm has affine invariance, but it cannot be reliably used to train all of a MLP's

Algorithm 5 LM algorithm

```
1: Initialize $\mathbf{w}$, $N_{it}$ , it$\leftarrow$ 0 and a small value for $\lambda$
2: while it $< N_{it}$ do
3:     Calculate the current error such as the MSE in equation (2.4)
4:     Calculate $\mathbf{g}$ and $\mathbf{H}$ from equation (3.19) and equation (3.18)
5:     Obtain $\mathbf{H}_{LM}$ from equation (3.32)
6:     Compute $\mathbf{d}_{LM}$ from equation (3.33)
7:     Update $\mathbf{w}$ as $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{d}_{LM}$
8:     Re-compute the error $E_{new}$ by using the updated weights.
9:     if $E_{new} < E_{old}$ then
10:        Reduce the value of $\lambda$
11:        goto step 5
12:    else
13:        Increase the value of $\lambda$
14:    end if
15:    it $\leftarrow$ it + 1
16: end while
```

weights. One solution to this problem is to modify Newton's algorithm using regulariztion, resulting in LM. Another approach follows in the next subsections.

3.3 Output reset for classifier design

The mean square error (MSE) cost function works well for the approximation case, since the corresponding outputs are continuous. But it might have a problem in the classification case, when outputs are all discrete numbers. For example, we have a target output 1 for the 4 classes case, so the target output vector is a one-hot encoding such as

$$\mathbf{t_p} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.34}$$

Assuming the output vectors we have are

$$\mathbf{y_{p1}} = \begin{bmatrix} 1.5 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.35}$$

and

$$\mathbf{y_{p2}} = \begin{bmatrix} 1 \\ -0.5 \\ 0 \\ 0 \end{bmatrix} \tag{3.36}$$

there are inconsistent errors in both cases, which cause the MSE to increase but the error percentage does not change. We denote i_c as the correct class and i_d is one of the incorrect ones. In the above example, $i_c = 1$ and $i_d \in \{2, 3, 4\}$. Then the inconsistent errors happen when $y_p(i_c) > t_p(i_c)$ or $y_p(i_d) < t_p(i_d)$. Output reset [79] solves the problem. The idea is modifying the cost function and target outputs as

$$E' = \frac{1}{N_v} \sum_{\mathrm{p}=1}^{N_v} \sum_{\mathrm{i}=1}^{M} [t'_{\mathrm{p}}(\mathrm{i}) - y_{\mathrm{p}}(\mathrm{i})]^2 \tag{3.37}$$

where $t'_p(i)$ is the modified output vectors defined as

$$t'_p(i) = t_p(i) + a_p + d_p(i) \tag{3.38}$$

where a_p and $d_p(i)$ are the values we want to find in order to optimize the modified cost function (3.37). A straight forward way to do that is setting the derivative of E' with respect to a_p to zero, which yields

$$a_p = \frac{1}{M} \sum_{i=1}^{M} [y_p(i) - t'_p(i) - d_p(i)] \tag{3.39}$$

and then we can update $d_p(i)$ using the found a_p and the current $\mathbf{y_p}$, $\mathbf{t'_p}$ as

$$d_p(i) = y_p(i) - t'_p(i) - a_p \tag{3.40}$$

Similar to backpropagation, the process of finding a_p, and updating $\mathbf{d_p}$ and $\mathbf{t'_p}$ can be performed multiple times to get a better $\mathbf{t'_p}$ which does not cause much inconsistent error to the MSE. We make it 3 times in the following algorithm, but it can be any positive integer.

Algorithm 6 Output reset(OR) algorithm

```
Retrieve current $\mathbf{y_p}$ and $\mathbf{t_p}$, zeros initialize $a_p$ and $\mathbf{d_p}$, it$\leftarrow 0$
while it $< 3$ do
    Calculate $a_p$ as in equation (3.39)
    Update $\mathbf{d_p}$ as in equation (3.40)
    Update $\mathbf{t'_p}$ using equation (3.38)
end while
```

An improved version of output reset (OR) has been found [27] which gives the final answer without iteration but requires the ordering of the outputs. Unfortunately this might not work well for large data files. Algorithm (6), used in this dissertation, can be vectorized and run instantly.

Chapter 4

Partial affine invariance in MLP training

One of the principal drawbacks to using MLPs is the sensitivity of training to the initial weight values. If an MLP training algorithm has affine invariance, then the objective function training satisfies the definition (2) for every nonsingular matrix $\mathbf{T}$. This means that training yields equivalent results for an uncountably infinite number of different initial weight vectors. Therefore, using affine invariant training is a first step towards making MLP training insensitive to initial weights. We've developed several MLP training algorithms [3, 14, 15, 47, 48, 61, 68, 69] that use Newton's algorithm in each iteration to find a vector of unknown gains or learning factors. So far, we haven't shown the relationship between this approach and Newton's algorithm for finding all the network's weights. In this section, we show that increasing the number of elements in the unknown vector improves performance. We also show that when $dim(\mathbf{z}) < N_w$, where N_w is the number of network weights, our algorithms have partial affine invariance, rather than affine invariance for all N_w unknowns.

4.1 Error versus learning factor dimensionality

In this section we show that increasing the dimension of the unknown vector $\mathbf{z}$ leads to improved algorithm performance.

Lemma 4.1.1 *Assume $E(\mathbf{w})$ is a quadratic objective function of the N_w dimensional weight vector $\mathbf{w}$ which is divided into k partitions $\mathbf{w}_k$ as $\mathbf{w} = [\mathbf{w}_1^T; \mathbf{w}_2^T; ...\mathbf{w}_k^T]$ and $\mathbf{g}_k = -\frac{\partial E}{\partial \mathbf{w}_k}$. If one iteration of a training algorithm minimizes E with respect to the k-dimenional vector $\mathbf{z}$ yielding an error $E_k = E(\mathbf{w_1} + z_1\mathbf{g}_1, \mathbf{w_2} + z_2\mathbf{g}_2, ...\mathbf{w_k} + z_k\mathbf{g}_k)$ and k increases by splitting one of the existing partitions, then $E_{k+1} \leq E_k$*

Proof:

The error $E(\mathbf{w})$ after updating the weight vector can be modeled as:

$$E(\mathbf{w} + \mathbf{d}) = E_0 + \mathbf{d}^T\mathbf{g} + \frac{1}{2}\mathbf{d}^T\mathbf{H}\mathbf{d} \tag{4.1}$$

where E_0 is the error before updating the weights, $\mathbf{g} = [\mathbf{g}_1^T, \mathbf{g}_2^T, ...\mathbf{g}_k^T]$ denotes the negative gradient vector and its components, $\mathbf{H}$ is the network's Hessian, and $\mathbf{d}$ is the weight change vector of dimension N_w. If $\mathbf{d}$ is found using Newton's method, then

$$\mathbf{d} = \mathbf{H}^{-1}\mathbf{g} \tag{4.2}$$

By contrast, the weight change vector for k groups and k learning factors is

$$\mathbf{d}_k = [z_1\mathbf{g}_1^T; z_2\mathbf{g}_2^T; ...z_k\mathbf{g}_k^T] \tag{4.3}$$

Given $\mathbf{z} = argmin_{\mathbf{z}}(E(\mathbf{w} + \mathbf{d}_k))$ then

$$\mathbf{d}_{k+1} = [z_1\mathbf{g}_1^T; z_2\mathbf{g}_2^T; ...z_{ka}\mathbf{g}_{ka}^T; z_{kb}\mathbf{g}_{kb}^T] \tag{4.4}$$

If $z_{ka} = z_{kb} = z_k$ then $\mathbf{d}_k = \mathbf{d}_{k+1}$ and $E_{k+1} \leq E_k$. However since the $k+1$ elements in $\mathbf{z}$ can be improved by a new stage of Newton's algorithm and E is quadratic, we get $E_{k+1} \leq E_k$

Lemma 4.1.2 *Given that k can only increase by splitting one of the existing partitions, our algorithm becomes Newton's algorithm as k is increased to N_w if $\mathbf{g}$ is not sparse.*

Proof:

When the dimension of $\mathbf{z}$ is exactly number of weights N_w in the MLP, then each group of weights will have only one element or one weight. The error function becomes

$$E_{N_w} = E(w_1 + z_1 g_1, w_2 + z_2 w_2, ...w_{N_w} g_{N_w}) \tag{4.5}$$

But Newton's algorithm will make $z_1 g_1 = d_1, z_2 g_2 = d_2, ...z_{N_w} g_{N_w} = d_{N_w}$. Our learning factor optimization performs Newton's algorithm as long as $\mathbf{g}$ has no elements equal to zero, resulting in $\lim_{k \to N_w} \mathbf{d}_k = \mathbf{d}$.

Lemma 4.1.1 indicates that increasing the dimension of the vector $\mathbf{z}$ leads to a better performance per iteration. Lemma 4.1.2 proves that increasing k until it reaches N_w leads to Newton's method for all network weights

4.2 Multilayer optimal learning factors training

Multiple optimal learning factors (MOLF) [48] is a typical PAI training method which order of $(N_h + 1)^2 / N_w^2$ as mentioned in table 3.1. MOLF uses a PAI methods to update the input weights and use OWO[51] to solve for the optimal output weights.

Consider the same cost function MSE as in equation 2.4. MOLF assigned one learning factor for each hidden unit, so hidden unit k^{th} has a learning factor $z(k)$ as

$$w_i(k,n) \leftarrow w_i(k,n) + z(k) \cdot g_i(k,n) \tag{4.6}$$

The input gradient $\mathbf{g_i}$ can be improved using HWO[86] as described in section 3.1.3. The k^{th} hidden unit net function will change as

$$n_p(k) = \sum_{n=1}^{N+1} [w_i(k,n) + z(k) \cdot g_i(k,n)] \cdot x_p(n) \tag{4.7}$$

The MOLF Jacobian vector has elements

$$j(k) = -\frac{\partial E}{\partial z(k)} = \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} [t_p(i) - y_p(i)] \frac{\partial y_p(i)}{\partial z(k)} \tag{4.8}$$

where

$$\frac{\partial y_p(i)}{\partial z(k)} = w_{oh}(i,k) \cdot o'_p(k) \cdot \sum_{n=1}^{N+1} g(k,n) \cdot x_p(n) \tag{4.9}$$

where $o_p(k)$ is the output activation of $n_p(k)$, and $o'_p(k)$ is the derivative of $o_p(k)$ with respect to the net function $n_p(k)$

$$\begin{aligned} o_p(k) &= f(n_p(k)) \\ o'_p(k) &= \frac{\partial o_p(k)}{\partial n_p(k)} \end{aligned} \tag{4.10}$$

Elements of the MOLF input weight Gaussian-Newton Hessian are given by

$$\begin{aligned} H(k,m) &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}\frac{\partial^2 y_p(i)}{\partial z(k)\partial z(m)} \\ &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}\frac{\partial y_p(i)}{\partial z(k)}\cdot\frac{\partial y_p(i)}{\partial z(m)} \end{aligned} \tag{4.11}$$

The learning factor vector $\mathbf{z}$ can be found by solving

$$\mathbf{H}\mathbf{z} = \mathbf{j} \tag{4.12}$$

The MOLF algorithm can be summarized as algorithm 7

Algorithm 7 Multiple optimal learning factors (MOLF) algorithm

```
Initialize $\mathbf{w}$, $N_{it}$ , it$\leftarrow$ 0
while it $< N_{it}$ do
    Calculate input gradient $\mathbf{g_i}$ and update it using HWO as in equation (3.7)
    Compute Jacobian $\mathbf{j}$ and Hessian $\mathbf{H}$ as equation (4.8) and(4.11)
    Solve equation (4.12) to find the learning factor vector $\mathbf{z}$
    Update the input weights as in (4.6)
    Solving output weights by OWO as in (3.30)
    it $\leftarrow$ it + 1
end while
```

MOLF works really well in many datasets, its performance was published in [48]. But the MOLF's Hessian still has the size of $N_h \times N_h$ which can be a big matrix, and the use of OWO makes MOLF not scale well for deep learning and big data.

Chapter 5

Problems and Proposed work

In this chapter, we explain some serious problems of MLP training, then propose tasks that solve these problems.

5.1 Problems

- *First order methods are not affine invariant*
  Lemmas 3.2.1 and 3.2.2 prove that first order methods such as steepest descent and conjugate gradient are not affine invariant. As a result, they are really sensitive to weights initialization and different affine transforms of the training data. One important advantage of first order training methods is that they are scalable, usually take $O(N_w)$ operations for one training iteration. The scalability makes first order training methods become dominant in deep learning and big data.

- *Newton's method lacks of scalability*
  Lemma 3.2.3 proves that Newton's method is affine invariant. Unfortunately, Newton's method is not scalable due to extensive of computation which makes it applicable only for small networks [9]. Newton's method requires to compute and store a $N_w \times N_w$ Hessian matrix, which is a big number. If the network has thousands of unknowns, then the Hessian matrix has millions elements. In addition, Newton's method also requires to invert the Hessian or solve a big set of linear equations which requires $O(N_w^3)$ operations. All of these burdens makes

Newton's method suitable for only small networks which is not applicable for deep learning.

- *Redundancy in gradients*
  Training using regular steepest descent is slow in general. There are many different ways adjust the gradient to speed up training, such as conjugate gradient, RMSProp [29] and the Adam method [35]. This suggests that the gradient has redundancy and can be further optimized.

- *OWO-Newton method is not stable*
  The OWO-Newton method [69] is a powerful second order training method which can minimize the cost function much faster than all other first and second order training methods . Unfortunetaly, OWO-Newton is not very stable and can fail when the cost function is not quadractic. The problem's details and its solution will be discussed in section 7.4.2

5.2 Objectives and Tasks

Our objectives are to (1) develop highly scalable one-step second order methods for large networks and (2) improve a two-step second order method for smaller networks. The proposed tasks are as follows

- *T1. Develop a scalable second order method*
  First order methods are scalable but not affine invariant. Newton's method is affine invariant but scalable. We want to develop a theory of methods which can take advantage of affine invariance but still be scalable. Part of theory is already published in [61].

- *T2. Develop a scalable gradient method for ReLU networks*
  Rectify linear unit (ReLU) is the dominating activation function in training neural network and deep learning, which motivates us to develop a theory for it. In fact, we finished part of the theory presented in preliminary work.

- *T3. Develop a scalable gradient method for sigmoidal networks*
  Sigmoidal is a traditional activation function which is still widely used. Having a theory for this function will definitely improve our contribution.

- *T4. Improve upon T2 and T3 using HWO*
  Hidden weight optimization [86] improves gradients which can be used in any layer in the network. We will try to take advantage of HWO to improve upon T2 and T3.

- *T5. Improve the OWO-Newton method*
  As mentioned in the problems part, OWO-Newton can fail in some datasets. In chapter 7, we will investigate possible reasons and propose a new method which fixes the failures problem.

Chapter 6

Balanced gradient back propagation

In this section, first we propose a scalable partial affine invariant method that works but fails at some iterations. Then, we analyze the fails iteration which causes by ill-condition Hessian. Solving the ill-conditioned Hessian leads to a novel scalable second order method. The new method can also take advantage of whitening and works well on both of the MLP and the CNN.

6.1 Back propagation with two learning rates (BP2)

The steepest descent algorithm can be improved by using multiple learning rates or learning factors. With shallow networks that have one hidden layer and two weight matrices, it might be better to let each weight matrix have its own learning rate. The two learning rates can also be calculated using Newton's method.

Consider the same MSE cost function as equation (2.4). The output activation $o_p(k)$ and the actual outpt $y_p(i)$ can be re-written as

$$\begin{aligned} o_p(k) &= f\left(\sum_{n=1}^{N+1} x_p(n) \cdot (w_i(k,n) + z_1 \cdot g_i(k,n))\right) \\ y_p(i) &= \sum_{k=1}^{N_h+1} o_p(k) \cdot (w_o(i,k) + z_2 \cdot g_o(i,k)) \end{aligned} \tag{6.1}$$

where $\mathbf{g_i}$ and $\mathbf{g_o}$ are input and output negative gradient defined as

$$\begin{aligned} \mathbf{g_i} &= -\frac{\partial E}{\partial \mathbf{w}_\mathrm{i}} \\ \mathbf{g_o} &= -\frac{\partial E}{\partial \mathbf{w}_\mathrm{o}} \end{aligned} \tag{6.2}$$

There are two unknows z_1 and z_2 which can be stacked as a vector $\mathbf{z} = [z_1, z_2]$. The 2×2 Hessian and two elements gradient vector of the unknown vector $\mathbf{z}$ are calculated as

$$\begin{aligned} h(l,n) &= \frac{2}{N_v} \sum_{\mathrm{p}=1}^{N_v} \sum_{i=1}^{M} \frac{\partial y_\mathrm{p}(i)}{\partial z_l} \cdot \frac{\partial y_\mathrm{p}(i)}{\partial z_n} \\ g(k) &= -\frac{2}{N_v} \sum_{\mathrm{p}=1}^{N_v} \sum_{i=1}^{M} (t_\mathrm{p}(i) - y_\mathrm{p}(i)) \cdot \frac{\partial y_\mathrm{p}(i)}{\partial z_k} \end{aligned} \tag{6.3}$$

The unknown vector is calculated the same as in equation (3.20), then the weight matrices are separately updated as

$$\begin{aligned} \mathbf{w}_\mathrm{i} &= \mathbf{w}_\mathrm{i} + z_1 \cdot \mathbf{g}_\mathrm{i} \\ \mathbf{w}_\mathrm{o} &= \mathbf{w}_\mathrm{o} + z_2 \cdot \mathbf{g}_\mathrm{o} \end{aligned} \tag{6.4}$$

The BP2 algorithm can be summarized as following

Algorithm 8 BP2 algorithm

```
Initialize $\mathbf{w_i}$, $\mathbf{w_o}$, $N_{it}$ , it$\leftarrow$ 0
while it $< N_{it}$ do
    Calculate $\mathbf{g_i}$, $\mathbf{g_o}$
    Compute $z_1$ and $z_2$ by solving equation (3.20)
    Update $\mathbf{w_i}$ and $\mathbf{w_o}$ as $\mathbf{w_i} \leftarrow \mathbf{w_i} + z_i \cdot \mathbf{g_i}$ and $\mathbf{w_o} \leftarrow \mathbf{w_o} + z_o \cdot \mathbf{g_o}$
    it $\leftarrow$ it + 1
end while
```

Clearly, BP2 has a PAI order of $4/N_w^2$. BP2 performs better than regular backpropagation but it is still slow and can be further improved. Fig. 6.1 shows BP2's performance compared with regular backpropagation.

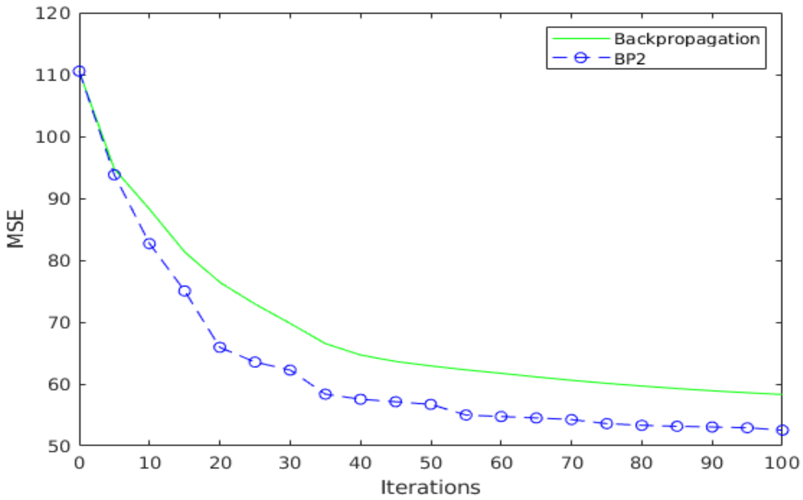

Figure 6.1: Backpropagation and BP2 comparison

6.2 BP2 and ill-conditioned Hessians problem

Ill-conditioned Hessian's is a well-known problem of Newton's method [9], which causes the Hessian to become non-invertible. There are different methods to solve this problem. LM method [44] modifies the Hessian to fix the ill-condition and make it invertible. Orthogonal least square [16, 34] gives acceptable solution without Hessians inverting. In this subsection, we show the effect of Hessian's ill-condition when Newton's method is applied for the case of two learning rate.

We apply the BP2 to the data set oh7.tra [49]. We also obtain the determinant of the BP2's Hessian, which is the 2×2 matrix of equation (**??**). The results are shown in figure 6.2

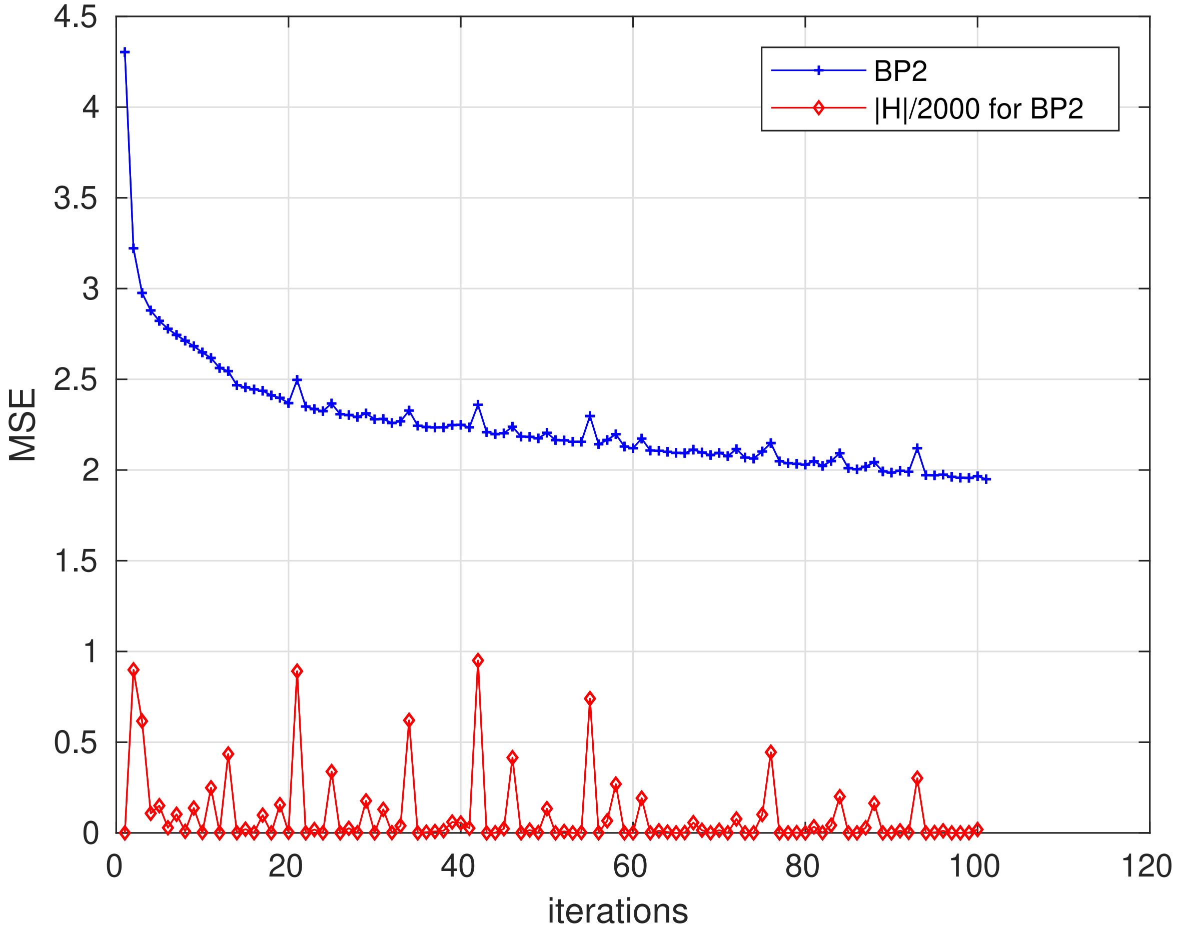

Figure 6.2: Hessian's ill-condition effect on BP2, oh7.tra dataset

The BP2 fails some times which causes its error increases. The iteration when BP2 fails usually has a small Hessian's determinant which can be considered as ill-conditioned Hessian. An interesting thing to note is that, the Hessian's determinant is proportional to the error decrease of the BP2 case. When the Hessian's determinant is small enough, the BP2 algorithm starts to fail.

We want to tackle this problem by optimizing the scaling factor of section 4 followed by the use of an optimal learning factor z.

The scaling factor can be found using Newton's method as follows. Consider cost function $E(z)$ as a function of learning rate z. We have Taylor expansion for the cost function $E(z)$

$$E(z) = E(0) + z\frac{\partial E}{\partial z} + z^2\frac{1}{2}\frac{\partial^2 E}{\partial z^2} \tag{6.5}$$

If z is calculated by Newton's method then

$$z = -\frac{\frac{\partial E}{\partial z}}{\frac{\partial^2 E}{\partial z^2}} \tag{6.6}$$

Substitute to the previous equation, we have

$$E(z) = E(0) - \frac{\frac{\partial E}{\partial z}}{\frac{\partial^2 E}{\partial z^2}}\frac{\partial E}{\partial z} + \left(\frac{\frac{\partial E}{\partial z}}{\frac{\partial^2 E}{\partial z^2}}\right)^2 \frac{1}{2}\frac{\partial^2 E}{\partial z^2} = E(0) - \frac{1}{2}\frac{(\frac{\partial E}{\partial z})^2}{\frac{\partial^2 E}{\partial z^2}} \tag{6.7}$$

So basically

$$E(z) = E(0) - \frac{1}{2}z\frac{\partial E}{\partial z} \tag{6.8}$$

The result is surprisingly simple, in order to minimize $E(z)$, we just need to maximize the product of z and its gradient. If z is a function of some variables, then maximizing $z\frac{\partial E}{\partial z}$ becomes a typical optimization problem. We now consider this optimization problem on two popular neural network architecture, regular fully connected feed forward neural networks and convolutional neural networks (CNNs)

6.3 Non-unique gradient problem

In this section, we show that there are many different gradient based weight changes in addition to the standard negative gradient. Consider scaling the MSE of equation (2.4) as

$$E' = a \cdot E = \frac{a}{N_v}\sum_{\mathrm{p}=1}^{N_v}\sum_{\mathrm{i}=1}^{M}[t_{\mathrm{p}}(\mathrm{i}) - y_{\mathrm{p}}(\mathrm{i})]^2 \tag{6.9}$$

where a is a positive scalar. After scaling the output weights as

$$w'_o(i,k) = a \cdot w_o(i,k) \tag{6.10}$$

the output $y_p(i)$ can be re-written as

$$y_p(i) = \frac{1}{a}\sum_{k=1}^{N_h} o_p(k)w'_o(i,k) \tag{6.11}$$

We have the negative input and output gradient matrices $\mathbf{G}_i$, $\mathbf{G}_o$ as

$$\begin{aligned} \mathbf{G}_i &= -\frac{\partial E}{\partial \mathbf{W}_i} \\ g_o(i,k) &= -\frac{\partial E}{\partial w_o(i,k)} = -\frac{2}{N_v}\sum_{\mathrm{p=1}}^{N_v}\sum_{\mathrm{i=1}}^{M}[t_{\mathrm{p}}(\mathrm{i}) - y_{\mathrm{p}}(\mathrm{i})] \cdot \frac{\partial y_p(i)}{\partial w_o(i,k)} \end{aligned} \tag{6.12}$$

the corresponding input and output gradients are

$$\begin{aligned} \mathbf{G}'_i &= -\frac{\partial E'}{\partial \mathbf{W}'_i} = -\frac{a \cdot \partial E}{\partial \mathbf{W}_i} = a \cdot \mathbf{G}_i \\ g'_o(i,k) &= -\frac{\partial E'}{\partial w'_o(i,k)} = -\frac{2a}{N_v}\sum_{\mathrm{p=1}}^{N_v}\sum_{\mathrm{i=1}}^{M}[t_{\mathrm{p}}(\mathrm{i}) - y_{\mathrm{p}}(\mathrm{i})] \cdot \frac{1}{a}\frac{\partial y_p(i)}{\partial w_o(i,k)} \\ &= g_o(i,k) \end{aligned} \tag{6.13}$$

The weight updates of the equivalent network are

$$\begin{aligned} \mathbf{W'}_i &\leftarrow \mathbf{W'}_i + z \cdot \mathbf{G'}_i \\ \mathbf{W'}_o &\leftarrow \mathbf{W'}_o + z \cdot \mathbf{G'}_o \end{aligned} \tag{6.14}$$

Mapping back to original network, multiplying the input weight update with a and dividing the output weight update by the same a, we have

$$\begin{aligned} \mathbf{W}_i &\leftarrow \mathbf{W}_i + z \cdot a \cdot \mathbf{G}_i \\ \mathbf{W}_o &\leftarrow \mathbf{W}_o + z \cdot \frac{1}{a} \cdot \mathbf{G}_o \end{aligned} \tag{6.15}$$

Using a simple scaling equivalent network, we clearly show that equation (6.15) is a valid weight update and there are infinitely many ways to choose a, which results in an infinite number of valid scaled gradients. The above results show that

- It is valid to update weight matrices with different scaling factors
- a can be found to maximize the error decrease

6.4 Balanced gradient on fully connected neural networks

We consider a shallow neural network with one hidden layer. So there are two weights matrices $\mathbf{W}_i$ and $\mathbf{W}_o$. To make learning rate have a freedom to change, we consider learning rate as a function of a and z. The output activation $o_p(k)$ and the actual output $y_p(i)$ can be written as

$$\begin{aligned} o_p(k) &= f\big(\sum_{n=1}^{N+1} x_p(n)\cdot(w_i(k,n)+z\cdot a\cdot g_i(k,n)\big) \\ y_p(i) &= \sum_{k=1}^{N_h+1} o_p(k)\cdot(w_o(i,k)+\frac{z}{a}g_o(i,k)) \end{aligned} \tag{6.16}$$

where $f()$ is the activation function, z is the learning rate and a is the scaling factor that we want to optimize. For convenience of calculation, we consider the partial derivative of output $y_p(i)$ with respect to learning rate z

$$\begin{aligned} \frac{\partial y_p(i)}{\partial z} &= a\cdot\sum_{k=1}^{N_h+1}[o'_p(k)\cdot\sum_{n=1}^{N+1} x_p(n)\cdot g_i(k,n)]\cdot w_o(i,k)+\frac{1}{a}\cdot\sum_{k=1}^{N_h+1} o_p(k)\cdot g_o(i,k) \\ &= a\cdot m_1(p,i)+\frac{1}{a}\cdot m_2(p,i) \end{aligned} \tag{6.17}$$

where $o'_p(k)$ denotes the first partial derivative of $o_p(k)$ which respect to its net function.To alleviate the notation's complication, we denote

$$\begin{aligned} m_1(p,i) &= \sum_{k=1}^{N_h+1} o'_p(k)\cdot\sum_{n=1}^{N+1} x_p(n)\cdot g_i(k,n)\cdot w_o(i,k) \\ m_2(p,i) &= \sum_{k=1}^{N_h+1} o_p(k)\cdot g_o(i,k) \\ m_3(p,i) &= t_p(i)-y_p(i) \end{aligned} \tag{6.18}$$

Note that all $\mathbf{M_1}$, $\mathbf{M_2}$ and $\mathbf{M_3}$ are $N_v\times M$ dimensional matrices. In order to find the optimal values of z and a, as a result from equation (6.8) we maximize the value $\frac{(\frac{\partial E}{\partial z})^2}{\frac{\partial^2 E}{\partial z^2}}$

The numerator

$$\begin{aligned}\left(\frac{\partial E}{\partial z}\right)^2 &= \left(\sum_{p=1}^{N_v}\sum_{n=1}^{M}\left(\frac{2}{N_v}\cdot m_3(p,n)\big(a\cdot m_1(p,n)+\frac{1}{a}\cdot m_2(p,n)\big)\right)\right)^2\\ &= \frac{4}{a^2\cdot N_v^2}\cdot\left(\sum_{p=1}^{N_v}\sum_{n=1}^{M}a^2\cdot(m_3(p,n)m_1(p,n)+\sum_{p=1}^{N_v}\sum_{n=1}^{M}m_3(p,n)m_2(p,n)\right)^2\\ &= \frac{1}{a^2}\cdot(a^2\cdot T_1+T_2)^2\end{aligned} \tag{6.19}$$

where scalars T_1 and T_2 are

$$\begin{aligned}T_1 &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{n=1}^{M}m_3(p,n)m_1(p,n)\\ T_2 &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{n=1}^{M}m_3(p,n)m_2(p,n)\end{aligned} \tag{6.20}$$

The denominator:

$$\begin{aligned}\frac{\partial^2 E}{\partial z^2} &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{n=1}^{M}\left(\left(a\cdot m_1(p,n)+\frac{1}{a}\cdot m_2(p,n)\right)\cdot\left(a\cdot m_1(p,n)+\frac{1}{a}\cdot m_2(p,n)\right)\right)\\ &= \frac{2}{a^2N_v}\Bigg(\sum_{p=1}^{N_v}\sum_{n=1}^{M}a^4\cdot m_1(p,n)m_1(p,n)+2a^2\sum_{p=1}^{N_v}\sum_{n=1}^{M}m_1(p,n)m_2(p,n)\\ &+\sum_{p=1}^{N_v}\sum_{n=1}^{M}m_2(p,n)m_2(p,n)\Bigg)\\ &= \frac{1}{a^2}(a^4T_3+2a^2T_4+T_5)\end{aligned} \tag{6.21}$$

where scalars T_3, T_4 and T_5 are

$$\begin{aligned} T_3 &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{n=1}^{M} m_1(p,n)m_1(p,n) \\ T_4 &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{n=1}^{M} m_1(p,n)m_2(p,n) \\ T_5 &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{n=1}^{M} m_2(p,n)m_2(p,n) \end{aligned} \tag{6.22}$$

Then, the value we want to maximize becomes

$$fz = \frac{(\frac{\partial E}{\partial z})^2}{\frac{\partial^2 E}{\partial z^2}} = \frac{(b\cdot T_1 + T_2)^2}{b^2\cdot T_3 + 2\cdot b\cdot T_4 + T_5} \tag{6.23}$$

where $b = a^2$, and all T_1, T_2, T_3, T_4, T_5 are scalars. Reducing cost function turn out to be a optimization problem $max(f_z)$ with respect to b, and constrain $b > 0$. The optimization problem can be solved by taking derivative of fz with respect to b and set the numerator to zero. The numerator of the derivative then turns out to be

$$num = (2bT_1^2 + 2T_1T_2)(b^2T_3 + 2bT_4 + T_5) - (2bT_3 + 2T_4)(bT_1 + T_2)^2 = 0 \tag{6.24}$$

reducing the equation we have

$$b^2(T_1^2T_4 - T_1T_2T_3) + b(T_1^2T_5 - T_3T_2^2) + (T_1T_2T_5 - T_4T_2^2) = 0 \tag{6.25}$$

The solution turns out to be a second order equation. So it has at most 2 roots. By evaluating the roots, we can see which one gives the most error decreasing, and then use the chosen root to calculate learning factor following Newton's method. Combine with equations

$$z = \frac{\frac{\partial E}{\partial z}}{\frac{\partial^2 E}{\partial z^2}} = \frac{b\cdot T_1 + T_2}{b^2\cdot T_3 + 2T_4\cdot b + T_5} \tag{6.26}$$

Due to the fact that $b = a^2$, we only consider positive roots. The b value can go anywhere from very close to 0 to infinity. Re-considering equation (6.23), if b is close to 0, then the scaled gradient of the input weights is also close to 0, and $fz \to \frac{T_2^2}{T_5}$. Therefore, we should only update the output weights with learning rate $z = \frac{T_2}{T_5}$. Alternately, if

b is close to infinity, then the scaled gradient of the output weights is also close to 0, and $fz \to \frac{T_1^2}{T_3}$. In this case we should only update the input weights with learning rate $z = \frac{T_1}{T_3}$. The new algorithm not only scales the gradient but also updates only one weight matrix when that is best.

Algorithm 9 Balanced gradient algorithm

```
Initialize $\mathbf{W}_i, \mathbf{W}_o$, $N_{it}$ , it$\leftarrow$ 0
while it $< N_{it}$ do
    Calculate negative gradients $\mathbf{G}_i, \mathbf{G}_o$
    Update $\mathbf{G}i = \mathbf{G}i_{\text{hwo}}$ and $\mathbf{G}o = \mathbf{G}\text{o}_{\text{hwo}}$ as in equation (3.7) and (3.10).
    Compute roots of equation (6.25)
    Find the $b$ value that maximizes equation (6.23), which $b \in \{0, \infty, roots\}$
    If $b_{max} = 0$, update $\mathbf{W}_i$ only as $\mathbf{W}_i \leftarrow \mathbf{W}_i + \frac{T_1}{T_3} \cdot \mathbf{G}_i$
    If $b_{max} = \infty$, update $\mathbf{W}_o$ only as $\mathbf{W}_o \leftarrow \mathbf{W}_o + \frac{T_2}{T_5} \cdot \mathbf{G}_o$
    If $b_{max} \in roots$, compute $z$ from equation (6.26), update both $\mathbf{W}_i$ and $\mathbf{W}_o$ as $\mathbf{W}_i \leftarrow \mathbf{W}_i + z \cdot \sqrt{b} \cdot \mathbf{G}_i$ and $\mathbf{W}_o \leftarrow \mathbf{W}_o + \frac{z}{\sqrt{b}} \cdot \mathbf{G}_o$
    Optain validation error from validation data set
    it $\leftarrow$ it + 1
end while
Choose the final network as the one which lowest validation error
```

We apply balanced gradient to the dataset oh7.tra which causes BP2 to fail and update the figure 6.2 with the training curve of balanced gradient. The result shows in figure 6.3, balanced gradient does not have any fail iteration in the same training data. We observed that at iteration 19^{th}, when the Hessian is ill-condition, the balanced gradient only update the input weight matrix, which enable it to avoid the failure aht BP2 has. With the flexibility of choosing which weight matrix to update, balanced gradient can easily solve the singular Hessian's problem of Newton's method.

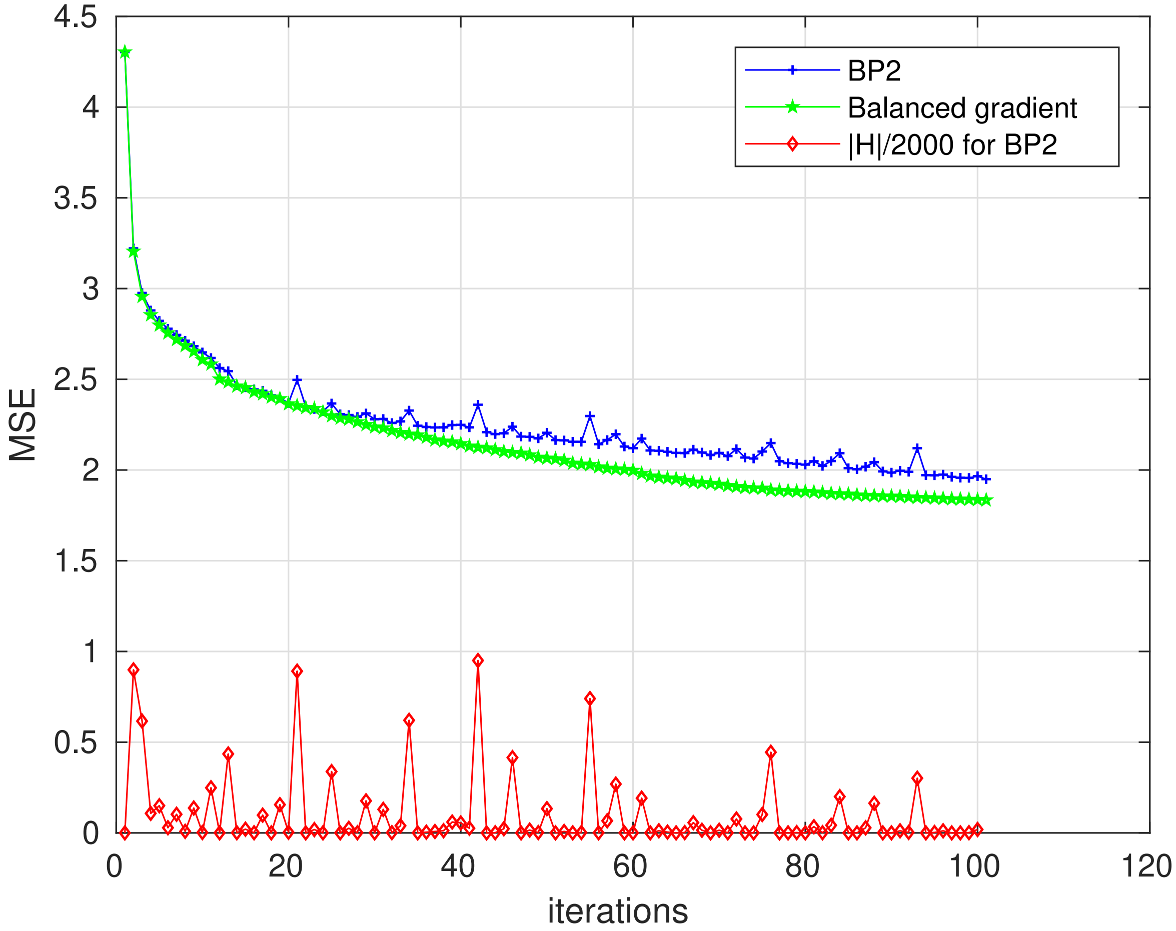

Figure 6.3: Hessian's ill-condition effect on BP2 and balanced gradient, oh7.tra dataset

Now, we do simulations to see the performance of the balanced gradient back propagation compared with conjugate gradient (CG) and LM. We do 10-folds validation and testing in all three algorithms in different datasets with the same initialized weights. Table 6.1 gives a short description of these datasets.

In each simulation, data is equally divided to 10 folds. Each fold becomes testing data once. In the remaining 9 folds, 8 folds are used for training and 1 fold is for validation. Validation error is calculated at every training iteration. The network which gives the smallest validation error is used for testing. In each simulation, we compare algorithm's MSE over iterations as traditional method. Due to balanced gradient and CG having very light computation effort compare with those of LM, we also compare MSE over multiplies. We also collect number the percentage of iterations which balanced gradient updates both input and output weights matrix.

Table 6.1: Data set descriptions

Data Sets	N	N_h	M	N_v
Rosenbrock	10	12	1	10000
Inverse 9	9	12	9	10000
Cover types	54	20	7	581012
super conductivity	81	20	1	21263
Ozone forecast	71	50	3	72050

6.4.1 Rosenbrock function dataset

The Rosenbrock function [71] is a well known non-convex highly non-linear function used to test the performance of optimization algorithms. The data file has 10000 samples with 10 inputs and one output.

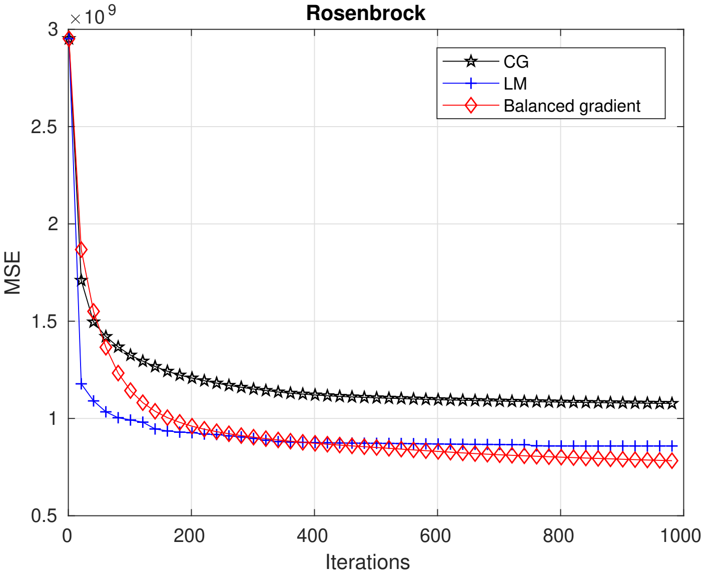

Figure 6.4: Rosenbrock dataset, training MSE vs iterations

In that classic problem, figure (6.4) and (6.5) shows that balanced gradient is superior on when comparing MSE versus both iterations and multiplies. One interesting thing in figure (6.5) to notice is that, conjugate gradient has many troubles when training starts. It has to backtrack many times before it can reduce the error which causes CG's accumulate multiplies is much more than that of balanced gradient. In the opposite, balanced gradient seems to have no problem in decreasing error when training starts.

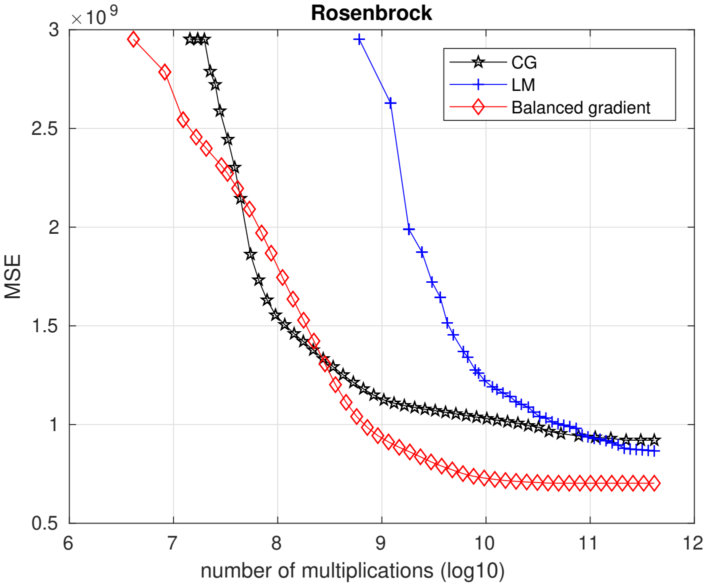

Figure 6.5: Rosenbrock dataset, training MSE vs multiplies

6.4.2 Inverse 9 dataset

The inverse 3×3 training dataset was created by randomly selecting 3×3 as input and calculate its invert matrix as output. So the dataset has 9 inputs and 9 outputs with 10000 samples. The purpose of this dataset is creating a highly non-linear training data to test performance of training algorithms.

In figure (6.6), it looks like that LM is much better than both balanced gradient and conjugate gradient. But it's much different in figure (6.7), when the real calculation burden is considered. Balanced gradient is actually better than both LM and CG. It is clear that one iteration of LM needs a lot more calculation than one iteration of CG or balanced gradient. It is the reason why we consider mean square error (MSE) versus multiplies as main comparison.

6.4.3 Cover types dataset

This dataset [10] is contains forest cover type for a given observation (30 x 30 meter cell) that was determined from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. Independent variables were derived from data originally obtained from US Geological Survey (USGS) and USFS data. Data is in raw form (not scaled) and contains binary (0 or 1) columns of data for qualitative independent variables

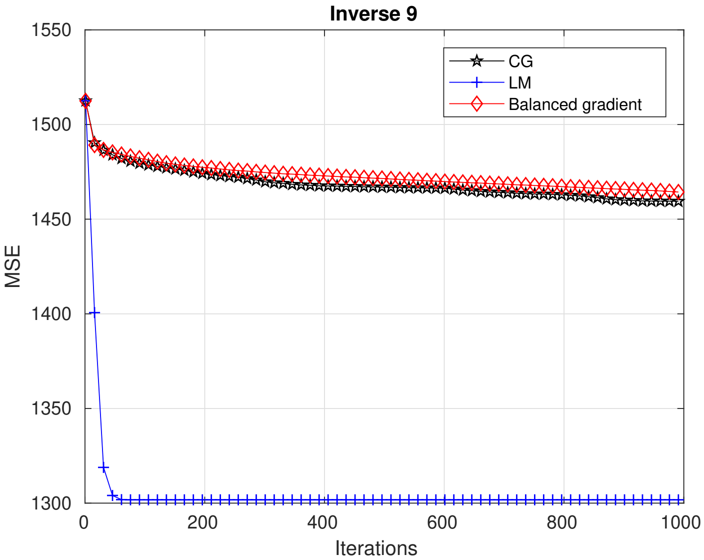

Figure 6.6: Inverse 9 dataset, training MSE vs iterations

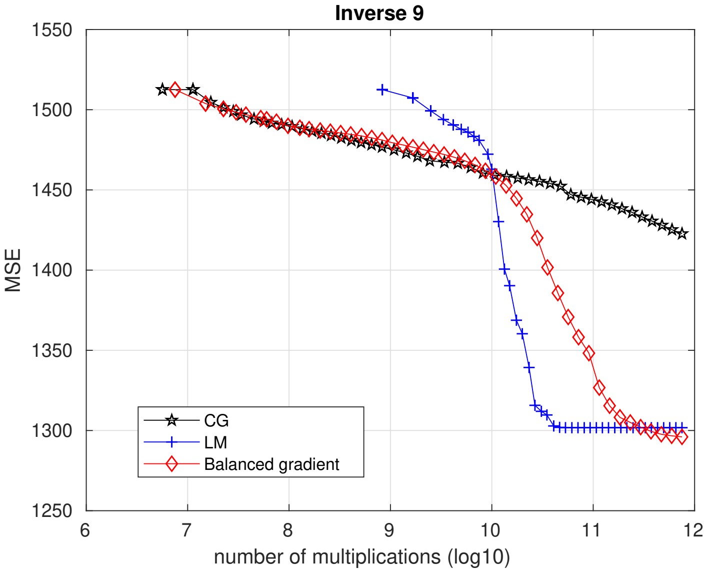

Figure 6.7: Inverse 9 dataset, training MSE vs multiplies

(wilderness areas and soil types). Figure (6.8) shows the superior of LM when it reduces the MSE so fast. Balanced gradient is still much better than conjugate gradient, and just a little bit less than that of LM.

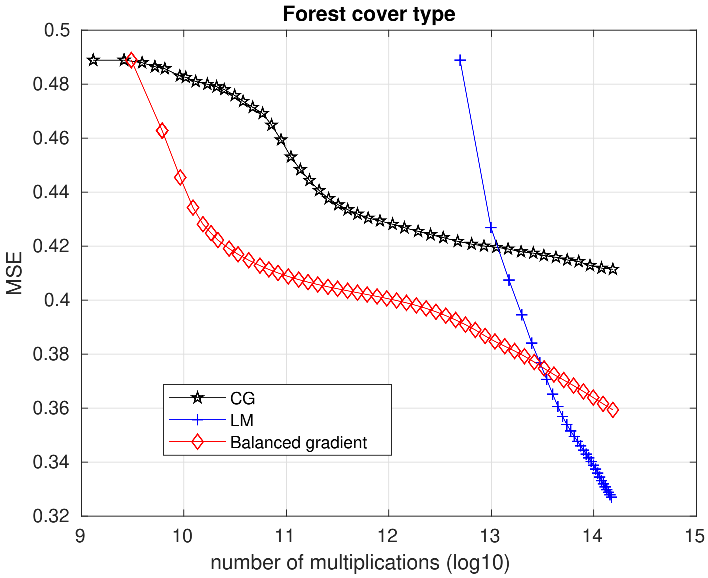

Figure 6.8: Cover type dataset, training MSE vs multiplies

6.4.4 Superconductivity dataset

The Superconductivity dataset [28] contains 81 features extracted from 21263 superconductors along with the critical temperature. So it has 81 inputs and 1 output. The gold is to predict temperature based on the features extracted. Again, as seen in figure (6.9), balanced gradient is much better than CG and ends up to have the same performance as LM.

6.4.5 Ozone forecast dataset

The Ozone forecasting data file [21] was made from years 2010 to 2013, it has 71 inputs and 3 outputs. First 4 inputs are time inputs (encoded in continuous form); Inputs 5 to 8 are spatial variables (latitude, longitude) that indicate the monitoring site/station and city the pattern comes from; Inputs 9 to 71 comprise time delayed data up to 3 days of Daily Mean, Daily Min, and Daily Max values of meteorological variables

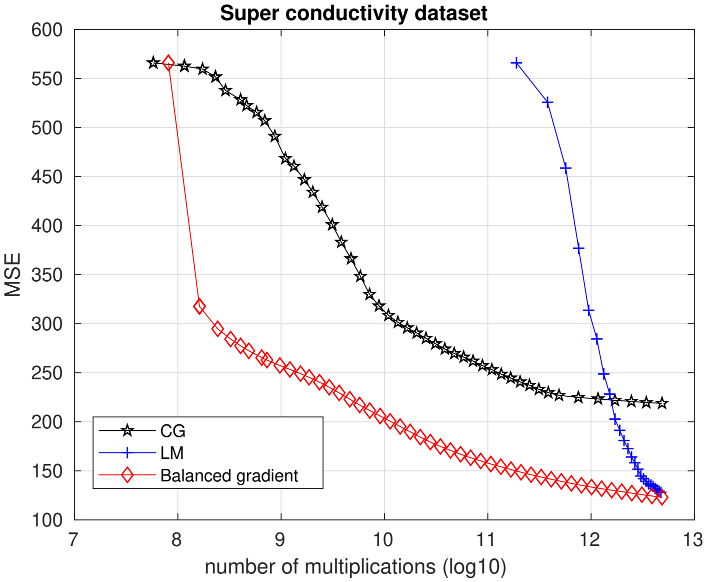

Figure 6.9: Super conductivity dataset, training MSE vs multiplies

(temperature, solar radiation, wind speed and wind direction encoded together in continuous form) and pollutant variables (nitric oxide, nitrogen dioxide, 8 - hour average ozone concentration). Outputs are Daily Maximum 8- hour average ozone concentration up to 3 days ahead. Figure (6.10) clearly shows that balanced gradient is superior than both LM and conjugate gradient.

6.4.6 Testing results

In each simulation, the final network is the one which gives lowest validation error, then this network will be used to obtain the testing result. We are doing 10-fold testing, so the following testing results in table 6.2 are the averages from these 10 networks. Balanced gradient back-propagation is superior than CG in all 5 datasets and better

Table 6.2: Ten-fold testing results

Data Sets	Balanced gradient	CG	LM
Rosenbrock	$\mathbf{7.9 \cdot 10^8}$	$10.24 \cdot 10^8$	$9.62 \cdot 10^8$
Inverse 9	**1501.3**	1502.8	1529.7
Cover types	24.79	28.78	**20.75**
super conductivity	172.03	226.36	**158.24**
Ozone forecast	**288.47**	299.59	299.32

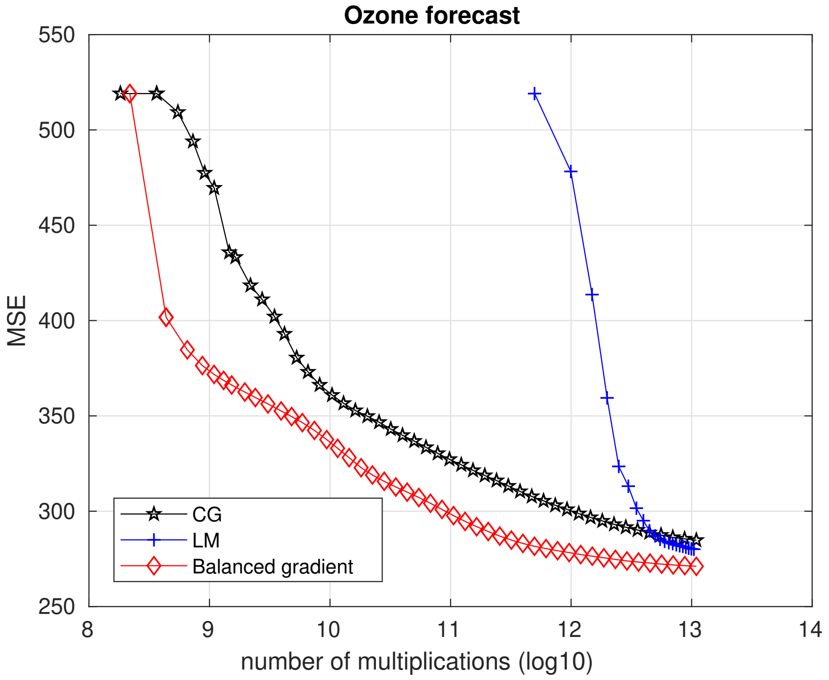

Figure 6.10: Ozone forecast dataset, training MSE vs multiplies

than LM in 3/5 datasets. As seen on table 6.3, more than half of the time, balanced gradient updates only one weight matrix. On average, it updates both weight matrices only on 26.88% of the iterations while LM and CG updates always update all the weights. It's clear that updating all the weights at the same time can be redundant, and each weight matrix should be treated differently. We also collect information about the multiplication needed to get the final network, the network that gives the lowest validation error. Result on table 6.4 is no surprise, balanced gradient is the best in all 5 datasets, it takes a lot less calculation effort to get the final network compare to that of CG and LM.

Table 6.3: Percentage of iteration where all weights are updated by balanced gradient

Data Sets	percentage
Rosenbrock	4.2%
Inverse 9	38.26%
Cover types	46.53%
Super Conductivity	17.55%
Ozone forecast	27.84%

Table 6.4: Multiplications needed for final networks

Data Sets	Balanced gradient	CG	LM
Rosenbrock	$\mathbf{3.44 \cdot 10^{10}}$	$11.03 \cdot 10^{10}$	$22.18 \cdot 10^{10}$
Inverse 9	$\mathbf{6.84 \cdot 10^{8}}$	$1509.7 \cdot 10^{8}$	$141.5 \cdot 10^{8}$
Cover types	$\mathbf{7.58 \cdot 10^{13}}$	$8.23 \cdot 10^{13}$	$44.15 \cdot 10^{13}$
Super conductivity	$\mathbf{2.84 \cdot 10^{12}}$	$12.97 \cdot 10^{12}$	$11.6 \cdot 10^{12}$
Ozone forecast	$\mathbf{7.81 \cdot 10^{12}}$	$10.24 \cdot 10^{12}$	$25.37 \cdot 10^{12}$

6.5 Balanced gradient on convolutional neural networks(CNNs)

CNNs often have two different types of trainable layers: the convolution layers which perform convolution operations and the fully connected layers which perform matrix multiplication operations. Most current training strategies use one heuristic global learning rate for all layers in the neural network even though different mathematical operations are used in each layer. It is natural to question whether or not balanced gradient will provide a benefit in CNN training.

In this sub-section, we apply the idea of balanced gradient to CNNs to answer the question and also to see if the proposed method improves CNNs' performance In the CNNs case, we also investigate a simple model with one convolution layer, 32 5x5 filters, one max-pooling layer, and one fully connected layer. The output activation image $\mathbf{O}_p(k)$ of the k^{th} filter can be written as

$$\mathbf{O}_p(k) = f\Bigg(\mathbf{X}_p * \big(\mathbf{W}_i(k) + z \cdot a \cdot \mathbf{G}_i(k)\big)\Bigg) \tag{6.27}$$

where $\mathbf{X}_p$ is the p^{th} image of the training data, $\mathbf{W}_i(k)$ is the weights of the k^{th} filter, $\mathbf{G}_i(k)$ is the gradient of the cost function with respect to $\mathbf{W}_i(k)$, and $(*)$ is the convolution operation.

Then we apply pooling $pool()$ and flatten $vec()$ [26] to $\mathbf{O}_p$ to make it becomes a vector, so we can feed it to the later fully connected layer.

$$\mathbf{o}_p = vec(pool(\mathbf{O}_p)) \tag{6.28}$$

The output $y_p(i)$ is still the same as the second part of equation (6.16). As a result, only calculation of the $\mathbf{M}_1$ matrix is different, which is

$$m_1(p,i) = \mathbf{w}_o(i) \cdot \Big(vec\big(pool\big(\mathbf{O'}_p \odot (\mathbf{X}_p * \mathbf{G}_i)\big)\big)\Big) \tag{6.29}$$

where $\mathbf{O'}_p$ denostes the first partial derivative of $\mathbf{O}_p$ which respect to its net function, $(\odot)$ is element-wise product operation and $\mathbf{w}_o(i)$ is the i^{th} row of output weight matrix $\mathbf{W}_o$.

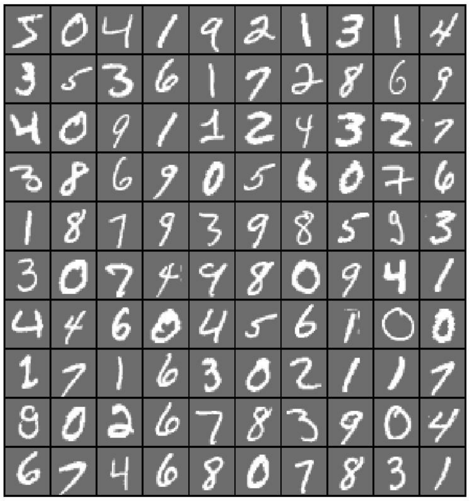

Figure 6.11: Samples of MNIST dataset

All the other calculations are still the same as the fully connected case. So the training algorithm is generally still the same.
We use the MNIST[40] , SVHN[59] and CIFAR10[36] datasets in this simulation. With a very simple CNN model comprising of one convolution layer with 32 5x5 filters, a max pooling layer, and a fully connected layer. We use minibatch size of 500 samples per iteration. There is no data augmentation involved in this simulation. Figure 6.11 and 6.12 shows some samples of these two datasets. In addition, we run simulation on Scrap dataset, one of our lab project dataset which classifies scrap and wrought.

Due to the scalable, we do not apply LM in this simulation. Conjugate gradient is supposed to work in batch-mode, but the study in [39] shows that CG can still work

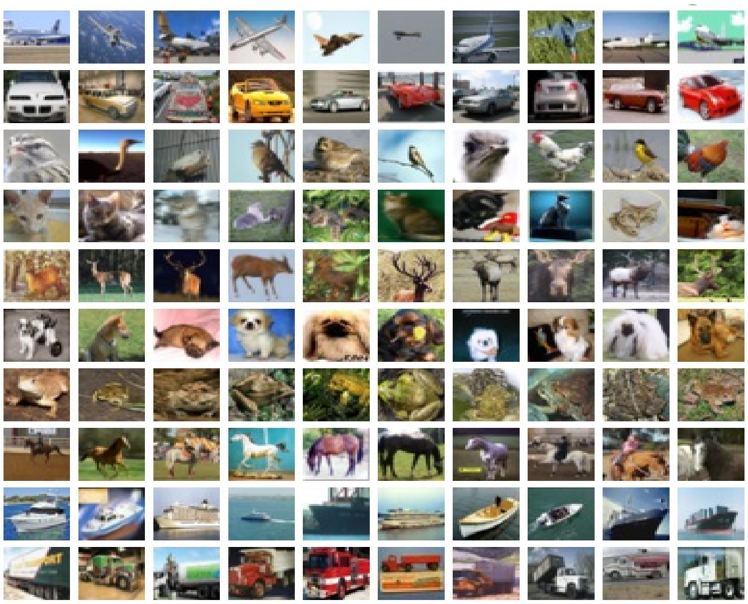

Figure 6.12: Samples of CIFAR10 dataset

well in mini-batch case with a suitable learning rate. In addition, we apply the basic idea of output-reset described in [79] to make the mean square error cost function work better with classification data. The output-reset was applied to both CG and balanced gradient. Because of scaling problem, we do not apply HWO whitening

Due to training in mini-batches, the current gradient in one mini-batch is just an estimate of the whole batch's gradient. Many approaches use momentum or accumulate gradients to have a better estimation. As described in algorithm 2, CG uses a combination of the current and previous gradient to update the weights, follows a learning rate. In this simulation, to guarantee fairness, we use the same gradient combination scheme as CG does, but with a learning rate found from the proposed balanced gradient approach. So the learning rate is the only difference between CG and balanced gradient in these simulations.

Figure (6.13) shows the learning rates which balanced gradient uses for convolutional layer and fully connected layer in MNIST dataset. Learning rates for convolution layers on average are roughly 3 times bigger than the learning rates for the fully connected one. Using a better learning rate, balanced gradient reduces error rate much faster than conjugate gradient in term of iterations or epochs, as shown in figure (6.15).

The same trend shows in Scrap, SVHN and CIFAR10 dataset in figure (6.14),(6.16) and figure (6.17). It is important to note that, evenwhen both methods use first derivative information, given the same epochs of training, balanced gradient often finishes training faster than conjugate gradient at roughly 20% of training time. It is because while balanced gradient has constant times pass through each mini-batch, conjugate gradient has to have additional passes for backtracking and line search to find a suitable learning rate which adds up computations to its training.

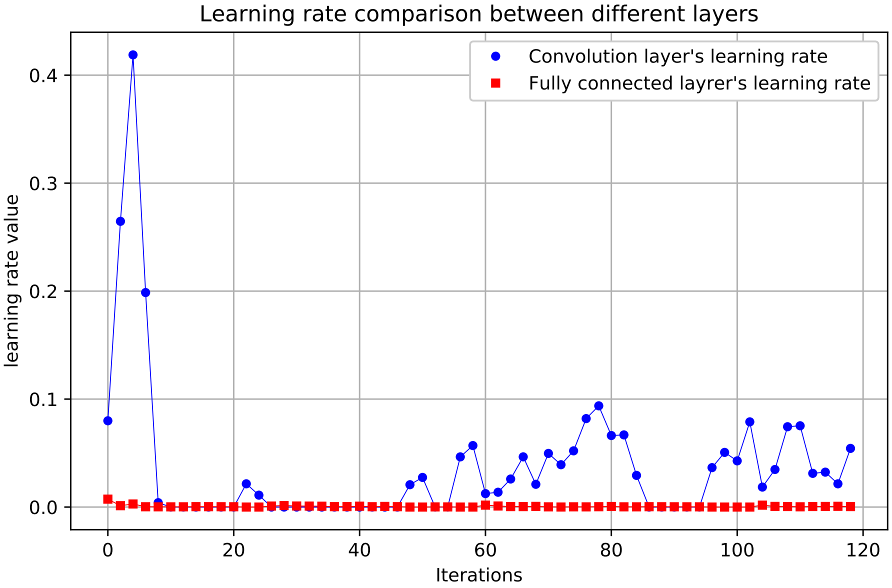

Figure 6.13: Difference in learning rate values of convolution and fully connected layers

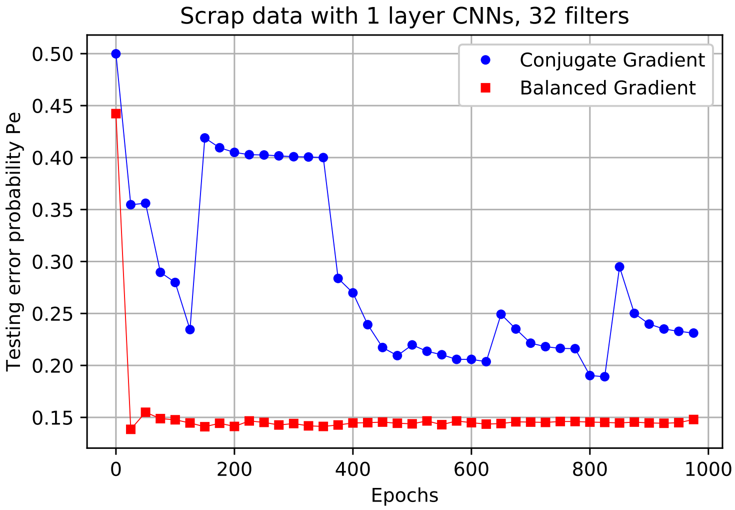

Figure 6.14: scrap testing error Pe of balanced gradient and conjugate gradient

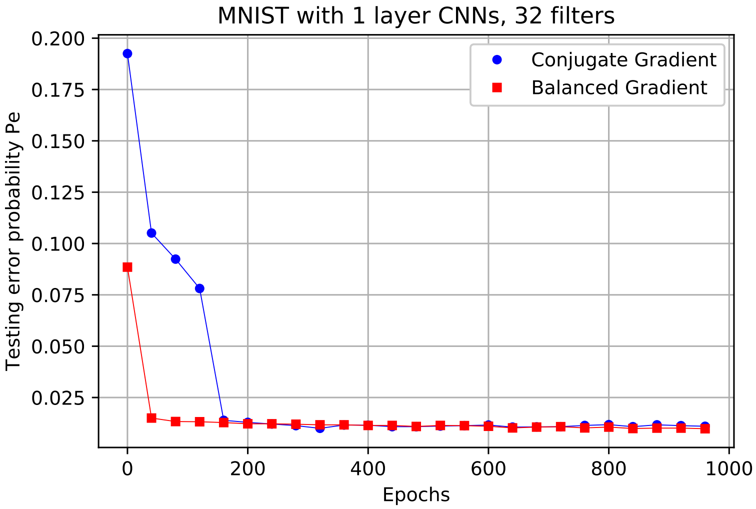

Figure 6.15: MNIST dataset testing error Pe of balanced gradient and conjugate gradient

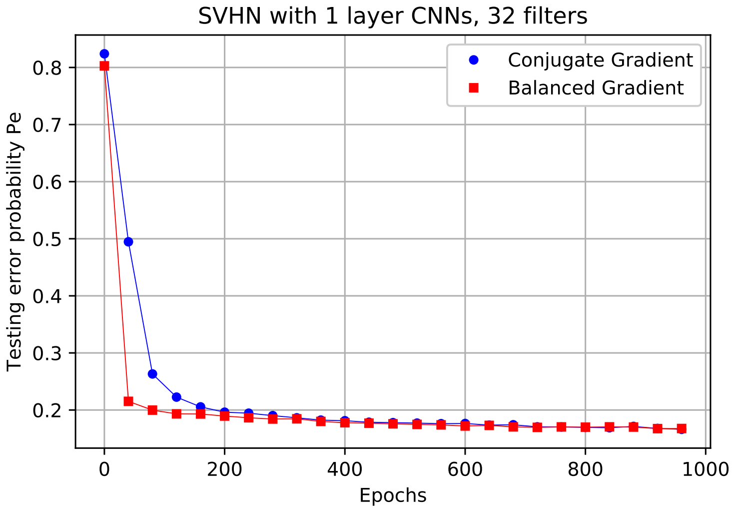

Figure 6.16: SVHN testing error Pe of balanced gradient and conjugate gradient

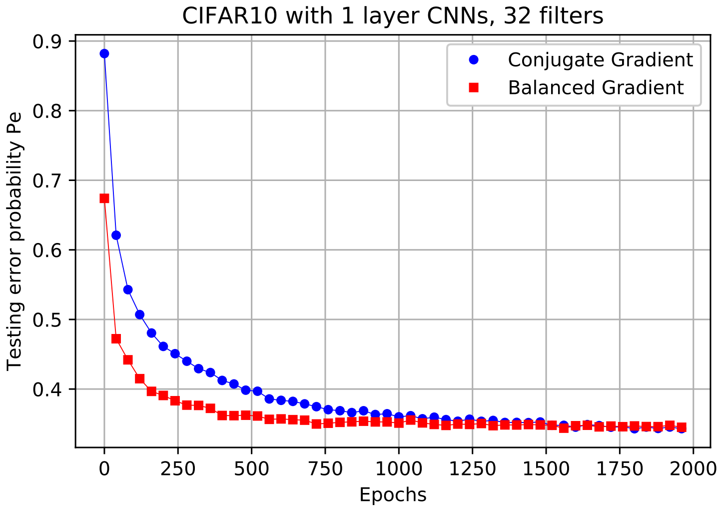

Figure 6.17: CIFAR10 testing error Pe of balanced gradient and conjugate gradient

Having a lighter computation burden than conjugate gradient, balanced gradient reduces error rate and converges much faster than conjugate gradient in all three datasets.

Chapter 7

OWO-Newton method

This chapter purely focuses on second order training methods. First, we investigate the reason why Newton's method is not stable in training neural networks. Then, we propose a method which solves this problem, making the algorithm more stable. After that, we propose a novel method which is much more stable and still has the fast convergence speed of second order methods.

7.1 Problems with the MLP Hessian

For fast convergence we would like to use Newton's method to train our MLP, but the Hessian $\mathbf{H}$ for the network is singular [83]. An alternative to overcome this problem is to modify the Hessian matrix as in the Levenberg-Marquardt (LM) algorithm. Another alternative is to use two-step methods such as layer by layer training [43]. Newton's method is derived from a 2^{nd} order Taylor series approximation to an objective function [78]. Applying this principle to equation (2.4) gives us the quadratic approximation

$$E(\mathbf{w}) \approx E_o - (\mathbf{w} - \tilde{\mathbf{w}})^T \mathbf{g} + \frac{1}{2}(\mathbf{w} - \tilde{\mathbf{w}})^T \mathbf{H}(\mathbf{w} - \tilde{\mathbf{w}}) \tag{7.1}$$

where $\tilde{\mathbf{w}}$ is $\mathbf{w}$ from the previous iteration, and is fixed. Also E_o is shorthand for $E(\tilde{\mathbf{w}})$.

In this Section we investigate the assumptions used by Newton's method and present the implications. When applied to the MSE as in equation (2.4), Newton's algorithm assumes that

- (A1) $E(\mathbf{w})$ in (2.4) is approximately quadratic as in 7.1.

- (A2) In each pattern, $\mathbf{y}_p$ is well approximated as a first degree function of $\mathbf{w}$.

Note that (A2) follows immediately from (A1)

7.2 Piecewise affine model of a single hidden layer MLP

We investigate whether (A2) is a valid assumption by constructing a first order model for $y_p(i)$. A model that yields the same Hessian and gradient as $E(\mathbf{w})$ is

$$\tilde{E}(\mathbf{w}) = \frac{1}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}[t_p(i) - \tilde{y}_p(i)]^2 \tag{7.2}$$

where $\tilde{y}_p(i)$ is

$$\begin{aligned}\tilde{y}_p(i) = \sum_{n=1}^{N+1} w_{oi}(i,n)x_p(n) + \sum_{k=1}^{N_h} w_{oh}(i,k)[O_p(k)+ \\ O'_p(k)(n_p(k) - \tilde{n}_p(k))]\end{aligned} \tag{7.3}$$

and

$$O'_p(k) = \frac{\partial O_p(k)}{\partial n_p(k)}\,\Big|_{\,n_p(k)=\tilde{n}_p(k)} \tag{7.4}$$

$$\tilde{n}_p(k) = \sum_{n=1}^{N+1} \tilde{w}_i(k,n)x_p(n) \tag{7.5}$$

In (7.3), we have used a first order Taylor series for each hidden unit activation for each pattern in the training file. Since we have a different model for each pattern, which is first degree in $\mathbf{x}_p$, we can term $\tilde{y}_p(i)$ a piecewise affine model of $y_p(i)$. The validity of the piecewise affine model is demonstrated by,

$$E(\mathbf{w}) = \tilde{E}(\mathbf{w}), \tag{7.6}$$

$$\frac{\partial E}{\partial w(u,v)} = \frac{\partial \tilde{E}}{\partial w(u,v)}, \tag{7.7}$$

and

$$\frac{\partial^2 E}{\partial w(u,v)\partial w(m,j)} = \frac{\partial^2 \tilde{E}}{\partial w(u,v)\partial w(m,j)} \tag{7.8}$$

Also the corresponding errors for each model, $t_p(i) - y_p(i)$ and $t_p(i) - \tilde{y}_p(i)$ are equal for $n_p(k) = \tilde{n}_p(k)$ since

$$\frac{\partial y_p(i)}{\partial w_i(u,v)} = w_{oh}(i,j)O'_p(u)x_p(v) = \frac{\partial \tilde{y}_p(i)}{\partial w_i(u,v)} \tag{7.9}$$

When the vector $\mathbf{w}$ includes all the network weights contained in $\mathbf{W}_i, \mathbf{W}_{oh}$ and $\mathbf{W}_{oi}$, $\tilde{y}_p(i)$ is not a first degree function of $\mathbf{w}$. To show this, we note that the exact expression for the output vector $\tilde{\mathbf{y}}_p$ for our network is

$$\tilde{\mathbf{y}}_p = [\mathbf{W}_{oi} + \mathbf{W}_{oh} diag(\mathbf{O}'_p)\mathbf{W}]\mathbf{x}_p + \mathbf{W}_{oh}[\mathbf{O}_p - diag(\mathbf{O}'_p)\tilde{\mathbf{n}}_p] \tag{7.10}$$

where $\mathbf{O}'_p$ denotes a vector whose k^{th} element is the derivative $f'(n_p(k))$. The model output $\tilde{y}_p(i)$ has products $w_{oh}(i,k)w(k,n)$. If all network weights can simultaneously vary then $\tilde{y}_p(i)$ is second degree in the unknowns, $\tilde{E}(\mathbf{w})$ is a fourth degree model in $\mathbf{w}$ and assumptions (A1) and (A2) are violated.

Clearly there is a discrepancy between $E_H(\mathbf{w})$ in equation (7.1) and $\tilde{E}(\mathbf{w})$ in (7.2). Since the products $w_{oh}(i,k)w(k,n)$ cause this discrepancy, the corresponding cross terms in blocks $\mathbf{H}_{oi}$ and $\mathbf{H}_{oi}^T$ of the network Hessian

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_R & \mathbf{H}_{oi}^T \\ \mathbf{H}_{oi} & \mathbf{H}_o \end{bmatrix} \tag{7.11}$$

are sources of error in training a MLP using Newton"s method.

7.3 Implications for MLP training

If we use Newton's algorithm for output weights, the block diagonal output weight Gauss-Newton Hessian matrix $\mathbf{H_o}$ is specified as

$$\mathbf{H}_o = \begin{bmatrix} 2\mathbf{R} & 0 & 0 & \cdots & 0 \\ 0 & 2\mathbf{R} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & 0 & 2\mathbf{R} & 0 \\ 0 & 0 & 0 & 0 & 2\mathbf{R} \end{bmatrix} \tag{7.12}$$

where $\mathbf{R}$ is the autocorrelation matrix given in equation (3.28). Later we show in detail that OWO is Newton's algorithm for output weights. The elements of the Gauss-Newton input weight Hessian, $\mathbf{H}_R$, are given by

$$\frac{\partial^2 E}{\partial w(j,k)\partial w(l,m)} = \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}\frac{\partial y_p(i)}{\partial w(j,k)}\cdot\frac{\partial y_p(i)}{\partial w(l,m)} \tag{7.13}$$

The elements of $\mathbf{H}_{oi}$ are calculated by

$$\frac{\partial^2 E}{\partial w_o(j,k)\partial w(l,m)} = \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}\frac{\partial y_p(i)}{\partial w(j,k)}\cdot\frac{\partial y_p(i)}{\partial w_o(l,m)} \tag{7.14}$$

The implications are

- When Newton's algorithm solves for all weights in $\mathbf{w}$ simultaneously, there can be multiple solutions for $\tilde{y}(i)$ and singular $\mathbf{H}$, making LM[44] a possible option.

- If we solve for elements for $\mathbf{w}$ one layer at a time in a two-step approach, $E_H(\mathbf{w}) = \tilde{E}(\mathbf{w})$, the cross terms in (7.10) are first degree in $\mathbf{w}$ and the discrepancy vanishes as seen for the input weight case in equations (7.6-7.8).

- The solution for $\mathbf{W}_o$ in the two-step approach is OWO which is described earlier. As M increases, the OWO algorithm's efficiency far outstrips that of the one-step approach.

7.4 OWO-Newton

Based upon the implications of subsection 7.3, we propose a two-step block coordinate descent (BCD) [78] approach that uses Newton's algorithm to alternately update input weights $\mathbf{W}_i$ and output weights $\mathbf{W}_o$. We give details of this method in the following.

7.4.1 Initial two step Newton's algorithm

In this section, our initial goal is to use Newton's algorithm to update $\mathbf{W}_i$ as

$$\mathbf{W}_i \leftarrow \mathbf{W}_i + \mathbf{D} \tag{7.15}$$

The steps for calculating $\mathbf{D}$ are as follows. Taking the negative first derivative of E with respect to $\mathbf{D}$, we have elements of the Jacobian matrix as:

$$g(k,n) = -\frac{\partial E}{\partial d(k,n)} = \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}[t_p(i) - y_p(i)] \cdot \frac{\partial y_p(i)}{\partial d(k,n)} \tag{7.16}$$

where

$$n_p(k) = \sum_{n=1}^{N+1}[w(k,n) + d(k,n)] \cdot x_p(n) \tag{7.17}$$

and

$$\frac{\partial y_p(i)}{\partial d(k,n)} = w_{oh}(i,k) \cdot O'_p(k) \cdot x_p(n) \tag{7.18}$$

The elements of the four dimensional Gauss-Newton input weight Hessian, $\mathbf{H}_4$ are given by

$$\begin{aligned} h_4(k,j,m,l) &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}\frac{\partial^2 y_p(i)}{\partial d(k,j)\partial d(m,l)} \\ &= \frac{2}{N_v}\sum_{p=1}^{N_v}\sum_{i=1}^{M}\frac{\partial y_p(i)}{\partial d(k,j)} \cdot \frac{\partial y_p(i)}{\partial d(m,l)} \end{aligned} \tag{7.19}$$

where $1 \leq k,m \leq N_h$ and $1 \leq j,l \leq N+1$. Elements of $\mathbf{H}_4$ can be mapped to a two-dimensional Hessian $\mathbf{H}_R$ as $h_R((j-1) \cdot N_h + k, (l-1) \cdot N_h + m) = h_4(k,j,m,l)$. Similarly elements of $\mathbf{g}$ are found from $\mathbf{G}$ as $g((l-1) \cdot N_h + m) = g(m,l)$

After obtaining $\mathbf{H}_R$ and $\mathbf{g}$ we apply orthogonal least squares [16] to find $\mathbf{d}$ from

$$\mathbf{H}_R\mathbf{d} = \mathbf{g} \tag{7.20}$$

Generating the $N_h \times (N+1)$ matrix $\mathbf{D}$ as $\mathbf{D} = vec^{-1}\{\mathbf{d}\}$, we update the input weight matrix $\mathbf{W}_i$ as in equation (7.15). Non-quadratic objective functions often require a line search. In this work, we use the dichotomous search [2], so equation (7.15) is modified as

$$\mathbf{W}_i \leftarrow \mathbf{W}_i + z \cdot \mathbf{D} \tag{7.21}$$

Our initial version of OWO-Newton alternately improves $\mathbf{W}_i$ and $\mathbf{W}_o$.

7.4.2 Problems with OWO-Newton

OWO-Newton approach can fail when assumption A1 is violated, resulting in an increase in E. When failure happens, we backtrack and substitute Multiple Optimal Learning Factors (MOLF) [48] for the input weight Newton step. MOLF has a smaller Hessian matrix which makes it more stable in reducing the MSE. The smaller Hessian results a average learning rate for a group of weight, details can be found in [61]. In the appendix, its Hessian matrix $\mathbf{H}_m$ and gradient vector $\mathbf{g}_m$ are calculated from $\mathbf{H}_R$. Using orthogonal least squares to solve

$$\mathbf{H}_m \cdot \mathbf{z} = \mathbf{g}_m \tag{7.22}$$

for $\mathbf{z}$, we update the input weight matrix $\mathbf{W}_i$ as explained in the appendix.

7.5 Partial affine invariance of OWO-Newton method

The partially affine invariance of OWO-Newton makes the algorithm perform indentically for equivalent initial networks. Unfortunately, approximately second order methods such as LM and BFGS do not have this property since we can't construct a matrix $\mathbf{T}$ for them.

We performed an experiment to illustrate partial affine invariance in OWO-Newton and to show its absence in LM[44] and BFGS[62]. The three algorithms were applied to the Rosenbrock [71] data sets. From the randomly initialized network, we created two other equivalent networks in which inputs are linear combinations of the original

Algorithm 10 OWO-Newton

```
Require: Iterations > 0
Initialize $\mathbf{W}_i$
Perform OWO
for k=1 to Iterations do
    Calculate $\mathbf{g}$ and $\mathbf{H}_R$
    Find $\mathbf{d}$ and $\mathbf{D}$, then update $\mathbf{W}_i$
    Perform OWO
    if the error increases then
        Back track input weights: $\mathbf{W}_i \leftarrow \mathbf{W}_i - z \cdot \mathbf{D}$
        Calculate $\mathbf{g}_m$ and $\mathbf{H}_m$
        Solve equation 7.22 and update $\mathbf{W}_i$ as
            $\mathbf{W}_i \leftarrow \mathbf{W}_i - diag(z) \cdot \mathbf{G}$
        Perform OWO
    end if
end for
```

inputs. Given a dataset and arrays $\mathbf{W}_\mathrm{i}$ and $\mathbf{W}_\mathrm{o}$ of an initial network, we construct a new dataset and an equivalent network as follows

(1) Find a nonsingular matrix $\mathbf{A}$ and derive the input stages of the equivalent network as

$$\begin{aligned}\mathbf{n}_\mathrm{p} =& \mathbf{W}_i \cdot \mathbf{x}_\mathrm{p} \\ =& \mathbf{W}_\mathrm{i} \cdot \mathbf{A}^{-1} \cdot \mathbf{A} \cdot \mathbf{x}_\mathrm{p} \\ =& \mathbf{W}'_\mathrm{i} \cdot \mathbf{x}'_\mathrm{p} \\ =& \mathbf{n}'_\mathrm{p}\end{aligned} \tag{7.23}$$

Note that $\mathbf{O}_\mathrm{p} = \mathbf{O}'_\mathrm{p}$ if $\mathbf{A}$ is non-singular. Now $\mathbf{W}'_\mathrm{i} = \mathbf{W}_\mathrm{i} \cdot \mathbf{A}^{-1}$ and $\mathbf{x}'_\mathrm{p} = \mathbf{A} \cdot \mathbf{x}_\mathrm{p}$.

(2) We can similarly show that the output weight matrices satisfy

$$\mathbf{W}'_\mathrm{o} = \mathbf{W}_\mathrm{o} \cdot \mathbf{A}^{-1} \tag{7.24}$$

In this way we generated three equivalent randomly initialized networks for each datafile. Then each algorithm was used to train the three equivalent initial networks. In figures 5 through 7, the three curves begin at the same MSE value, providing evidence that the networks start out equivalent.

For the Rosenbrock dataset, the training error curves for OWO-Newton overlay each other while the curves for LM diverge a great deal and the curves for BFGS diverge a lot more.

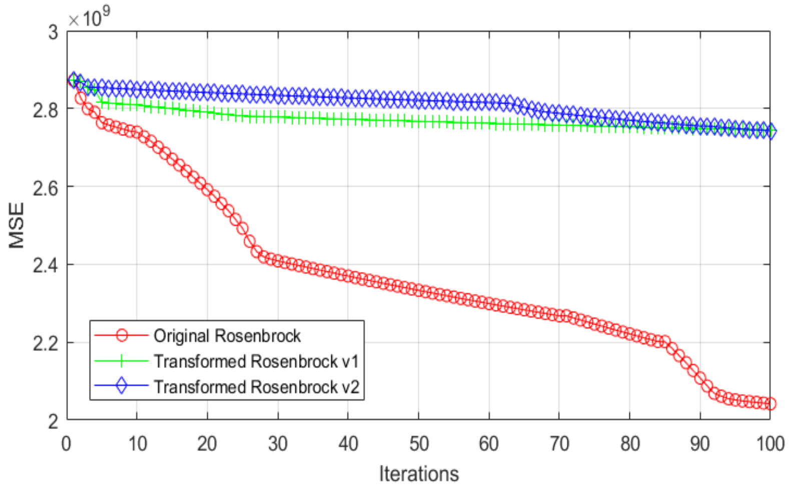

Figure 7.1: BFGS applied to transformed Rosenbrock datasets

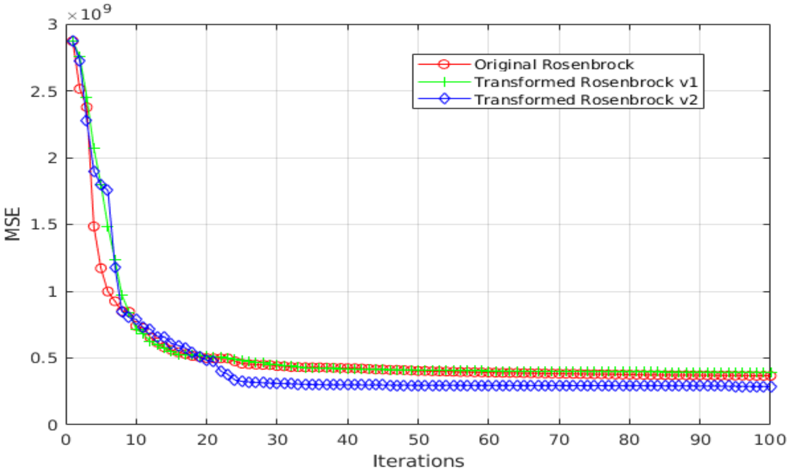

Figure 7.2: LM applied to transformed Rosenbrock data sets

In figures 7.1 through 7.3, we see that OWO-Newton performs consistently well on equivalent initial networks due to its partial affine invariance. In contrast, LM and BFGS perform unpredictably for the equivalent networks.

Newton's method is affine invariant but it requires the solution of a large set of linear equations which is expensive and suitable only for small networks [26].

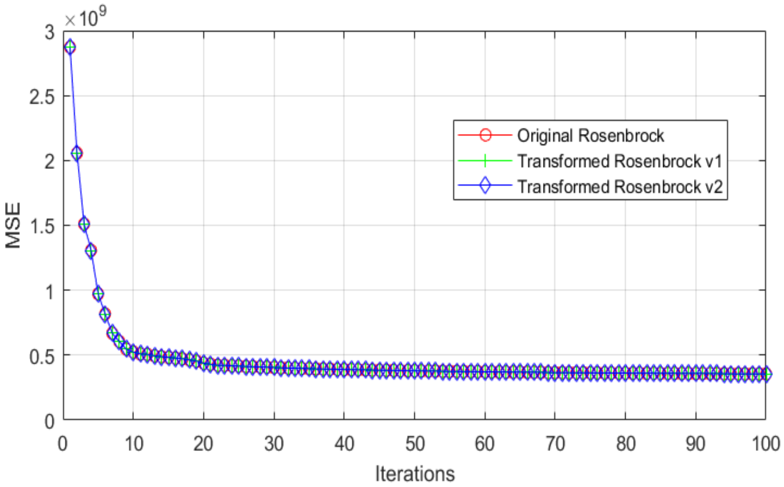

Figure 7.3: OWO-Newton applied to transformed Rosenbrock data sets

Chapter 8

Conclusions

In this dissertation, we have developed scalable partially affine invariant training algorithms for the MLP and the CNN, denoted as BP2 and balanced gradient. Balanced gradient solves the ill-conditioned Hessian problem in BP2 by developing a rational learning factor.

Balanced gradient only requires first order derivative information which makes it scalable for big datasets. We have shown that balanced gradient can work well with both regular fully connected networks and convolutional neural networks. In the CNN, balanced gradient is even faster than conjugate gradient. Applying balanced gradient makes the learning rate in convolutional neural networks no longer a heuristic, but optimal. The ability to alternatively update all or just one weight matrix makes balanced gradient effectively a two step training algorithm. In addition, balanced gradient outperforms conjugate gradient in first iteration and reaches the final network faster on both the MLP and the CNN. Balanced gradient has better MSE error or Pe in all simulations.

We have also found a way to solve the failed iteration problem in OWO-Newton which makes it much more stable. Simulations show that the improved OWO-Newton algorithm can return the same error curves given different linear transformations of input data.

Appendix A

Matrix derivative

Let $\mathbf{x} \in R^n$ (a column vector) and let $\mathbf{f} : R^n \rightarrow R^m$.The derivative of $\mathbf{f}$ with respect to $\mathbf{x}$ is the $m \times n$ matrix:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f(x)_1}{\partial x_1} & \cdots & \frac{\partial f(x)_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f(x)_m}{\partial x_1} & \cdots & \frac{\partial f(x)_m}{\partial x_n} \end{bmatrix} \tag{A.1}$$

Let $\mathbf{T} \in R^{m \times n}$ and $\mathbf{x} \in R^n$. Let $\mathbf{t}_1^T, ..., \mathbf{t}_n^T$ be the rows of $\mathbf{T}$

$$\mathbf{Tx} = \begin{bmatrix} \mathbf{t}_1^T \\ \mathbf{t}_2^T \\ \vdots \\ \mathbf{t}_m^T \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{t}_1^T\mathbf{x} \\ \mathbf{t}_2^T\mathbf{x} \\ \vdots \\ \mathbf{t}_m^T\mathbf{x} \end{bmatrix} \tag{A.2}$$

$$\frac{\partial \mathbf{Tx}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{t}_1^T\mathbf{x}}{\partial \mathbf{x}} \\ \frac{\partial \mathbf{t}_2^T\mathbf{x}}{\partial \mathbf{x}} \\ \vdots \\ \frac{\partial \mathbf{t}_m^T\mathbf{x}}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathbf{t}_1^T \\ \mathbf{t}_2^T \\ \vdots \\ \mathbf{t}_m^T \end{bmatrix} \tag{A.3}$$

so

$$\frac{\partial \mathbf{Tx}}{\partial \mathbf{x}} = \mathbf{T} \tag{A.4}$$

Consider function $E(\mathbf{w})$ which returns a scalar, column vectors $\mathbf{w}, \mathbf{w}' \in R^N$ and square matrix $\mathbf{T} \in R^{N \times N}$ where $\mathbf{w} = \mathbf{T}\mathbf{w}'$. From above proof, we have

$$\frac{\partial \mathbf{w}}{\partial \mathbf{w}'} = \mathbf{T} \tag{A.5}$$

Consider the derivative

$$\frac{\partial E}{\partial \mathbf{w}'} = \begin{bmatrix} \frac{\partial E}{\partial w'_1} \\ \frac{\partial E}{\partial w'_2} \\ \vdots \\ \frac{\partial E}{\partial w'_N} \end{bmatrix} \tag{A.6}$$

Each w'_i is a function of all elements of vector $\mathbf{w}$, where $1 \leq i \leq N$, so

$$\frac{\partial E}{\partial w'_i} = \sum_{j=1}^{N} \frac{\partial w_j}{\partial w'_i} \frac{\partial E}{\partial w_j} \tag{A.7}$$

which is equivalent to

$$\frac{\partial E}{\partial \mathbf{w}'} = \begin{bmatrix} \frac{\partial w_1}{\partial w'_1} & \cdots & \frac{\partial w_N}{\partial w'_1} \\ \vdots & & \vdots \\ \frac{\partial w_1}{\partial w'_N} & \cdots & \frac{\partial w_N}{\partial w'_N} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_N} \end{bmatrix} \tag{A.8}$$

or

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{w}'} &= \left(\frac{\partial \mathbf{w}}{\partial \mathbf{w}'} \right)^T \frac{\partial E}{\partial \mathbf{w}} \\ &= \mathbf{T}^T \frac{\partial E}{\partial \mathbf{w}} \end{aligned} \tag{A.9}$$

Appendix B

Converting Newton method to MOLF

In multiple optimal learning factors (MOLF) training algorithm, each hidden unit has one learning factor which will be used to update input weights as [48]:

$$w_i(k,n) \leftarrow w_i(k,n) + z(k) \cdot g(k,n) \tag{B.1}$$

The k^{th} hidden unit net function will change as

$$n_p(k) = \sum_{n=1}^{N+1} [w_i(k,n) + z(k) \cdot g(k,n)] \cdot x_p(n) \tag{B.2}$$

The MOLF Jacobian vector has elements

$$g_m(k) = -\frac{\partial E}{\partial z(k)} = \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} [t_p(i) - y_p(i)] \frac{\partial y_p(i)}{\partial z(k)} \tag{B.3}$$

$$\frac{\partial y_p(i)}{\partial z(k)} = w_{oh}(i,k) \cdot O'_p(k) \cdot \sum_{n=1}^{N+1} g(k,n) \cdot x_p(n) \tag{B.4}$$

recall equation (7.18)

$$\frac{\partial y_p(i)}{\partial d(k,n)} = w_{oh}(i,k) \cdot O'_p(k) \cdot x_p(n) \tag{B.5}$$

then, equation (B.4) becomes

$$\frac{\partial y_p(i)}{\partial z(k)} = \sum_{n=1}^{N+1} \frac{\partial y_p(i)}{\partial d(k,n)} \cdot g(k,n) \tag{B.6}$$

We can see that $\mathbf{g}_m$ can be calculated from OWO-Newton's Jacobian as

$$g_m(k) = \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} [t_p(i) - y_p(i)] \sum_{n=1}^{N+1} \frac{\partial y_p(i)}{\partial d(k,n)} \cdot g(k,n) \tag{B.7}$$

Elements of the OWO-MOLF input weight Gaussian-Newton Hessian are given by

$$\begin{aligned} h_m(k,m) &= \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} \frac{\partial^2 y_p(i)}{\partial z(k) \partial z(m)} \\ &= \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} \frac{\partial y_p(i)}{\partial z(k)} \cdot \frac{\partial y_p(i)}{\partial z(m)} \end{aligned} \tag{B.8}$$

Combining equations (B.4) and (B.8)

$$\begin{aligned} h_m(k,m) = \frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} [\sum_{n=1}^{N+1} \frac{\partial y_p(i)}{\partial d(k,n)} \cdot g(k,n)] \\ \cdot [\sum_{q=1}^{N+1} \frac{\partial y_p(i)}{\partial d(m,q)} \cdot g(m,q)] \end{aligned} \tag{B.9}$$

Comparing with equation (7.19)

$$\begin{aligned} h_m(k,m) = \sum_{j=1}^{N+1} \sum_{l=1}^{N+1} [\frac{2}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} \frac{\partial y_p(i)}{\partial d(k,j)} \cdot \frac{\partial y_p(i)}{\partial d(m,l)}] \\ \cdot g(k,j) \cdot g(m,l) \end{aligned} \tag{B.10}$$

or

$$h_m(k,m) = \sum_{j=1}^{N+1} \sum_{l=1}^{N+1} h_4(k,j,m,l) \cdot g(k,j) \cdot g(m,l) \tag{B.11}$$

Equation (B.7) gives the relationship between the OWO-Newton Jacobian and the MOLF Jacobian, while equation (B.11) relates the MOLF Hessian to the input weight

Hessian. To get the Hessian and Jacobian of MOLF, we just need to use information from Newton"s Hessian and Jacobian that are already available. From that Hessian and Jacobian, we can find the learning factor **z** and update the input weights as in equation (B.1).

Bibliography

[1] M Usman Akram and Anam Usman. Computer aided system for brain tumor detection and segmentation. In *2011 International Conference on Computer Networks and Information Technology (ICCNIT)*, pages 299–302. IEEE, 2011.

[2] Andreas Antoniou and Wu-Sheng Lu. *Practical optimization: algorithms and engineering applications.* Springer Science & Business Media, 2007.

[3] Babu Hemanth Kumar Aswathappa. Optimal output gain algorithm for feed-forward network training. 2011.

[4] Peter M Atkinson and ARL Tatnall. Introduction neural networks in remote sensing. *International Journal of remote sensing*, 18(4):699–709, 1997.

[5] Soumitro Swapan Auddy, Kanishka Tyagi, Son Nguyen, and Michael Manry. Discriminant vector tranformations in neural network classifiers. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 1780–1786. IEEE, 2016.

[6] Simon A Barton. A matrix method for optimizing a neural network. *Neural Computation*, 3(3):450–459, 1991.

[7] William G Baxt. Use of an artificial neural network for the diagnosis of myocardial infarction. *Annals of internal medicine*, 115(11):843–848, 1991.

[8] Suchendra M Bhandarkar, Jean Koh, and Minsoo Suk. Multiscale image segmentation using a hierarchical self-organizing map. *Neurocomputing*, 14(3):241–272, 1997.

[9] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[10] Jock A Blackard and Denis J Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and electronics in agriculture*, 24(3):131–151, 1999.

[11] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[12] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[13] Rüdiger W Brause. Medical analysis and diagnosis by neural networks. In *International Symposium on Medical Data Analysis*, pages 1–13. Springer, 2001.

[14] Xun Cai, Kanishka Tyagi, and Michael T Manry. An optimal construction and training of second order rbf network for approximation and illumination invariant image segmentation. In *The 2011 International Joint Conference on Neural Networks (IJCNN)*, pages 3120–3126. IEEE, 2011.

[15] Xun Cai, Kanishka Tyagi, Michael T Manry, and Zhi Chen. An efficient conjugate gradient based learning algorithm for multiple optimal learning factors of multilayer perceptron neural network. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 1093–1099. IEEE, 2014.

[16] Sheng Chen, Stephen A Billings, and Wan Luo. Orthogonal least squares methods and their application to non-linear system identification. *International Journal of control*, 50(5):1873–1896, 1989.

[17] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[18] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[19] Thierry Denoeux and Régis Lengellé. Initializing back propagation networks with prototypes. *Neural Networks*, 6(3):351–363, 1993.

[20] Gian Paolo Drago and Sandro Ridella. Statistically controlled activation weight initialization (scawi). *IEEE Transactions on Neural Networks*, 3(4):627–631, 1992.

[21] Gautam Raghavendra Eapi. *Comprehensive neural network forecasting system for ground level ozone in multiple regions*. PhD thesis, 2015.

[22] G-PK Economou, C Spiropoulos, NM Economopoulos, N Charokopos, D Lymberopoulos, M Spiliopoulou, E Haralambopulu, and CE Goutis. Medical diagnosis and artificial neural networks: a medical expert system applied to pulmonary diseases. In *Neural Networks for Signal Processing [1994] IV. Proceedings of the 1994 IEEE Workshop*, pages 482–489. IEEE, 1994.

[23] Michael Egmont-Petersen, Dick de Ridder, and Heinz Handels. Image processing with neural networks a review. *Pattern recognition*, 35(10):2279–2301, 2002.

[24] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *Icml*, volume 96, pages 148–156. Citeseer, 1996.

[25] Philip E Gill and Walter Murray. Conjugate-gradient methods for large-scale nonlinear optimization. Technical report, STANFORD UNIV CALIF SYSTEMS OPTIMIZATION LAB, 1979.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[27] RG Gore, Jiang Li, Michael T Manry, Li-Min Liu, Changhua Yu, and John Wei. Iterative design of neural network classifiers through regression. *International Journal on Artificial Intelligence Tools*, 14(01n02):281–301, 2005.

[28] Kam Hamidieh. A data-driven statistical model for predicting the critical temperature of a superconductor. *Computational Materials Science*, 154:346–354, 2018.

[29] Geoffrey Hinton, Nitsh Srivastava, and Kevin Swersky. Neural networks for machine learning. *Coursera, video lectures*, 264, 2012.

[30] Geoffrey E Hinton. To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547, 2007.

[31] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[32] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. 2003.

[33] Taskin Kavzoglu and Paul M Mather. Pruning artificial neural networks: an example using land cover classification of multi-sensor images. *International Journal of Remote Sensing*, 20(14):2787–2803, 1999.

[34] Parastoo Kheirkhah, Kanishka Tyagi, Son Nguyen, and Michael T Manry. Structural adaptation for sparsely connected mlp using newton's method. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4467–4473. IEEE, 2017.

[35] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[36] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[38] Sinem Kulluk, Lale Ozbakir, and Adil Baykasoglu. Training neural networks with harmony search algorithms for classification problems. *Engineering Applications of Artificial Intelligence*, 25(1):11–19, 2012.

[39] Quoc V Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress, 2011.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[41] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[42] KY Lee, YT Cha, and JH Park. Short-term load forecasting using an artificial neural network. *IEEE Transactions on Power Systems*, 7(1):124–132, 1992.

[43] Régis Lengellé and Thierry Denoeux. Training mlps layer by layer using an objective function for internal representations. *Neural Networks*, 9(1):83–97, 1996.

[44] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[45] K Liu, S Subbarayan, RR Shoults, MT Manry, C Kwan, FI Lewis, and J Naccarino. Comparison of very short-term load forecasting techniques. *IEEE Transactions on power systems*, 11(2):877–882, 1996.

[46] JT Luxhøj. An artificial neural network for nonlinear estimation of the turbine flow-meter coefficient. *Engineering Applications of Artificial Intelligence*, 11(6):723–734, 1998.

[47] Sanjeev S Malalur and Michael Manry. Feed-forward network training using optimal input gains. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 1953–1960. IEEE, 2009.

[48] Sanjeev S Malalur, Michael T Manry, and Praveen Jesudhas. Multiple optimal learning factors for the multi-layer perceptron. *Neurocomputing*, 149:1490–1501, 2015.

[49] M. T. Manry. Image processing and neural network laboratory, the university of texas at arlington, college of engineering, 2016. `http://www.uta.edu/faculty/manry/new_training.html`.

[50] MT Manry, MS Dawson, AK Fung, SJ Apollo, LS Allen, WD Lyle, and W Gong. Fast training of neural networks for remote sensing. *Remote sensing reviews*, 9(1-2):77–96, 1994.

[51] MT Manry, X Guan, SJ Apollo, LS Allen, WD Lyle, and W Gong. Output weight optimization for the multi-layer perceptron. In *Conference Record of The*

Twenty-Sixth Asilomar Conference on Signals, Systems and Computers, 1992., pages 502–506. IEEE, 1992.

[52] Jean-Pierre Martens. A stochastically motivated random initialization of pattern classifying mlps. *Neural Processing Letters*, 3(1):23–29, 1996.

[53] James L McClelland, David E Rumelhart, PDP Research Group, et al. Parallel distributed processing. *Explorations in the Microstructure of Cognition*, 2:216–271, 1986.

[54] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černockỳ. Strategies for training large scale neural network language models. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 196–201. IEEE, 2011.

[55] James Moor. The dartmouth college artificial intelligence conference: The next fifty years. *Ai Magazine*, 27(4):87, 2006.

[56] Nelson Morgan and Herve A Bourlard. Neural networks for statistical recognition of continuous speech. *Proceedings of the IEEE*, 83(5):742–772, 1995.

[57] Kevin P Murphy. *Machine learning: a probabilistic perspective.* Cambridge, MA, 2012.

[58] Shahrin Azuan Nazeer, Nazaruddin Omar, and Marzuki Khalid. Face recognition system using artificial neural networks approach. In *2007 International Conference on Signal Processing, Communications and Networking*, pages 420–425. IEEE, 2007.

[59] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[60] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 21–26. IEEE, 1990.

[61] Son Nguyen, Kanishka Tyagi, Parastoo Kheirkhah, and Michael Manry. Partially affine invariant back propagation. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 811–818. IEEE, 2016.

[62] Jorge Nocedal and Stephen Wright. *Numerical optimization*, chapter *6. Quasi-Newton Methods*, pages 135–163. Springer Science & Business Media, 2nd edition, 2006.

[63] Cemil Oz and Ming C Leu. American sign language word recognition with a sensory glove using artificial neural networks. *Engineering Applications of Artificial Intelligence*, 24(7):1204–1213, 2011.

[64] T Parisini and R Zoppoli. Neural networks for nonlinear state estimation. *International Journal of Robust and Nonlinear Control*, 4(2):231–248, 1994.

[65] Jagdish Chandra Patra, Ganapati Panda, and Rameswar Baliarsingh. Artificial neural network-based nonlinearity estimation of pressure sensors. *IEEE Transactions on Instrumentation and Measurement*, 43(6):874–881, 1994.

[66] Sebastian Polak, Agnieszka Skowron, Jerzy Brandys, and Aleksander Mendyk. Artificial neural networks based modeling for pharmacoeconomics application. *Applied Mathematics and Computation*, 203(2):482–492, 2008.

[67] Michael D Richard and Richard P Lippmann. Neural network classifiers estimate bayesiana posterioriprobabilities. *Neural computation*, 3(4):461–483, 1991.

[68] Melvin D Robinson and Michael T Manry. Partially affine invariant training using dense transform matrices. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE, 2013.

[69] Melvin D Robinson and Michael T Manry. Two-stage second order training in feedforward neural networks. In *FLAIRS Conference*, 2013.

[70] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[71] HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, 1960.

[72] Yong Rui and AA El-Keib. A review of ann-based short-term load forecasting models. In *Proceedings of the Twenty-Seventh Southeastern Symposium on System Theory, 1995.*, pages 78–82. IEEE, 1995.

[73] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[74] Yousuf Saifullah and Michael T Manry. Classification-based segmentation of zip codes. *IEEE transactions on systems, man, and cybernetics*, 23(5):1437–1443, 1993.

[75] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[76] Michael A Sartori and Panos J Antsaklis. A simple method to derive bounds on the size and to train multilayer neural networks. *IEEE transactions on neural networks*, 2(4):467–471, 1991.

[77] Adrian J Shepherd. *Second-order methods for neural networks: Fast and reliable training methods for multi-layer perceptrons*, chapter *1. Multi-Layer Perceptron Training*, pages 1–22. Springer Science & Business Media, 1st edition, 1996.

[78] Paul Tseng. Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications*, 109(3):475–494, 2001.

[79] Kanishka Tyagi and Michael Manry. Multi-step training of a generalized linear classifier. *Neural Processing Letters*, pages 1–20, 2018.

[80] Marotesa Voultsidou, Silke Dodel, and J Michael Herrmann. Neural networks approach to clustering of activity in fmri data. *IEEE transactions on medical imaging*, 24(8):987–996, 2005.

[81] Jin Wang and Jie Huang. Neural network enhanced output regulation in nonlinear systems. *Automatica*, 37(8):1189–1200, 2001.

[82] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[83] Jorg Wille. On the structure of the hessian matrix in feedforward networks and second derivative methods. In *International Conference on Neural Networks, 1997.*, volume 3, pages 1851–1855. IEEE, 1997.

[84] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[85] Jim YF Yam and Tommy WS Chow. A weight initialization method for improving training speed in feedforward neural network. *Neurocomputing*, 30(1):219–232, 2000.

[86] Changhua Yu and Michael T Manry. A modified hidden weight optimization algorithm for feedforward neural networks. In *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on*, volume 2, pages 1034–1038. IEEE, 2002.