THESIS

A RECURSIVE LEAST SQUARES TRAINING APPROACH FOR CONVOLUTIONAL NEURAL NETWORKS

Submitted by

Yifan Yang

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2022

ABSTRACT

A RECURSIVE LEAST SQUARES TRAINING APPROACH FOR CONVOLUTIONAL NEURAL NETWORKS

This thesis aims to come up with a fast method to train convolutional neural networks (CNNs) using the application of the recursive least squares (RLS) algorithm in conjunction with the back-propagation learning. In the training phase, the mean squared error (MSE) between the actual and desired outputs is iteratively minimized. The recursive updating equations for CNNs are derived via the back-propagation method and normal equations. This method does not need the choice of a learning rate and hence does not suffer from speed-accuracy trade-off. Additionally, it is much faster than the conventional gradient-based methods in a sense that it needs less epochs to converge. The learning curves of the proposed method together with those of the standard gradient-based methods using the same CNN structure are generated and compared on the MNIST handwritten digits and Fashion-MNIST clothes databases. The simulation results show that the proposed RLS-based training method requires only one epoch to meet the error goal during the training phase while offering comparable accuracy on the testing data sets.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Mahmood R Azimi-Sadjadi, for initiating this research, and commenting and revising my thesis multiple times. He steered me in the right direction whenever I was lost in my pursuit of this research. I would also like to thank everyone in our Digital Signal/Image Processing Laboratory for providing me with valuable feedback on my work. I would like to thank Elliott Forney, who supports the LaTeX template to every graduate student at Colorado State University (CSU) and made my writing life easier. When I was doing simulations and writing my thesis in Spring 2020, Covid-19 was spreading out. During such a tense and stressful time, the Electrical and Computer Engineering (ECE) department at CSU and the Summit High Performance Computer (HPC) system at the University of Colorado Boulder (CU) provided virtual classroom and HPC cluster support that helped me tremendously to run my simulations remotely. Finally, I would like to thank everyone who made efforts in helping CSU community overcome Covid-19 outbreak so that I can concentrate on my thesis instead of worrying too much about the consequences.

DEDICATION

I would like to dedicate this thesis to my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Problem Statement and Motivations

Since the back-propagation learning was successfully implemented on multi-layer feedforward neural networks [1], they have been widely used in many machine learning problems. Based on the back-propagation algorithm and the gradient-based approaches, other types of neural networks, such as convolutional neural networks (CNNs) [2, 3] and recurrent neural networks (RNNs) [4], were developed. In back-propagation neural networks (BPNNs), each layer is fully connected to the next layer via synaptic weights and non-linear activations at neurons (nodes). For training BPNNs or updating the weights, a cost function is formed and then the errors between the desired output and the actual output are back-propagated layer-by-layer through these weights. In CNNs, the convolutional layers are connected via filter weights followed by pooling operation (down-sampling), and the final fully connected layer. The filters have shared weights among the neurons of the previous layer to reduce the number of parameters in the networks. Pooling operation also shrinks the size of images without any additional parameters. The final fully connected layers are similar to those of the standard BPNNs. For training CNNs, the errors are back-propagated through the weights of the fully connected layers, the pooling and the convolutional layers one-by-one. To accomplish this, up-sampling is performed on every pooling layer to back-propagate the errors to the corresponding convolutional layer. Then all the parameters in the network can be updated using the back-propagated errors much like the standard BPNNs.

CNNs have particularly been playing an important role in image recognition problems since they have the advantage of being invariant to scaling and translation [2, 3, 5]. Figure 1.1 shows an example of CNNs, called *LeNet-5* [5], which was designed for object recognition from 32×32 grayscale input images (single channel). There are two convolutional layers in LeNet-5 with 6 and 16 sets of shared filter weights in each and hence the same amount of feature maps with size 28×28

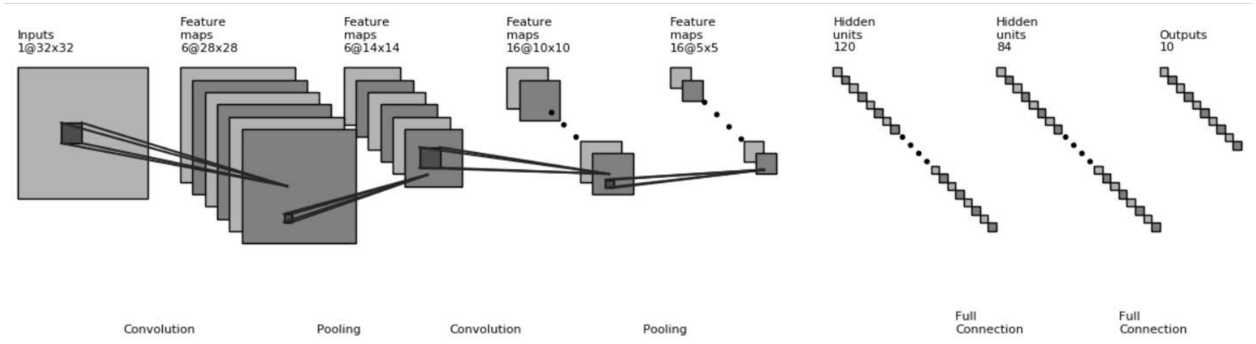
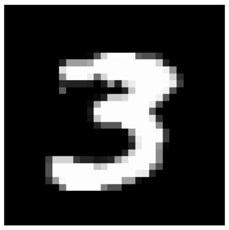
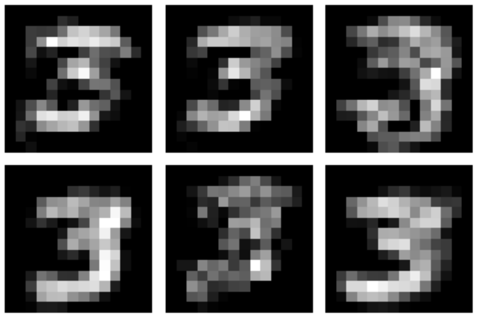


Figure 1.1: An example of CNNs: LeNet-5.

and pooled feature maps with size 10×10 at each layer. The shared weights in the convolutional layers allow the network to extract the more important topology of the objects, e.g., edges, corners, and relative locations, rather than the exact location information [2, 3, 6]. By going through the convolutional layers, the high-level feature representations with their spatial information from the input images can be extracted. At the second convolutional layer of LeNet-5, the feature representations become more abstract features of the input image. Also, distortions and translations can always happen in images. The pooling layers after the convolutional layers can overcome them by reducing the resolution of the images [5] as well as the sensitivity of scaling, shifting, and distortion [7]. In LeNet-5, each convolutional layer is followed by a pooling layer and the feature maps at the convolutional layers are down-sampled with a 2×2 window. The pooling layers introduce the translation-invariance and reduce the size of the feature maps and hence less required number of parameters in the next layer. Finally, at the second pooling layer in LeNet-5, all the neurons are flattened and connected to a two-hidden-layer BPNN with 120 nodes and 80 nodes followed by 10 output neurons.

Examples of the pooled feature maps are shown in Figure 1.2. The image of digit 3 in Figure 1.2 (a) from the MNIST handwritten digit dataset was propagated to a well-trained LeNet-5. After the input image was convolved with 6 filters at the first convolutional layer and passed through the pooling operation, 6 pooled feature maps were extracted as shown in Figure 1.2 (b). It can be


(a) Input image with size 32×32.


(b) Pooled feature maps after the first layer. Each pooled feature map is of size 14×14 pixels.

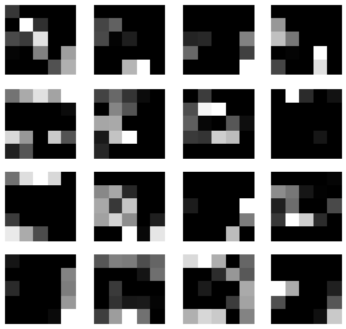

(c) Pooled feature maps after the second layer. Each pooled feature map is of size 4×4 pixels.

Figure 1.2: Examples of the pooled feature maps from LeNet-5 where the input image is a digit 3 from the MNIST dataset. The digit 3 can be roughly observed in the extracted pooled feature maps at the first layer. But the pooled feature maps become abstract at the second layer.

observed that each pooled feature map captured different low-level features while still preserving the shape of digit 3. These pooled feature maps were then applied to the second convolutional layer with 16 sets of filters and the resultant feature maps were pooled leading to the results shown in Figure 1.2 (c). As can be noted that these pooled feature maps at the second layer became higher-level and more abstract. Consequently, the shapes or the outlines of the digit 3 were no longer as obvious as those at the first layer. Comparing with the size of the input image, i.e. $32 \times 32 = 1024$ pixels, the total number of the pixels in these extracted pooled feature maps is only $16 \times 5 \times 5 = 400$ pixels. Thus, the following fully connected layer will have a much smaller number of parameters.

The problem of designing and training CNNs is challenging in a sense that there is room to improve the performance in both training speed and overall accuracy for a given problem. Recursive least squares (RLS) algorithm [8–10] is a well-known adaptive algorithm with fast convergence property and has successfully been implemented for the BPNNs [11, 12]. Results in these papers showed that, compared with the conventional techniques, the RLS-based training approach not only significantly reduced the total number of training iterations, but also achieved high accuracy on the testing data. However, this fast training approach has not yet been explored for CNNs. The reason being the difficulties in deriving the error back-propagation at the convolutional and the pooling layers of CNNs and writing the normal equations in the RLS form.

To this end, this thesis focuses on extending the RLS-based learning to CNNs and similar networks. The training method is designed with the goal that the trained CNNs can not only achieve a comparable accuracy with state-of-the-art training methods, but also provide better convergence and more efficient implementation.

1.2 Literature Review

As far as the training algorithms for CNNs are concerned, stochastic gradient descent (SGD) [2, 3, 5, 13] has shown its effectiveness and wild usage in such optimization problems. SGD is able to converge even when the loss function is not fully differentiable everywhere [13], in which case the gradient is infinity or doesn't exist. Unlike batch gradient descent, which updates the unknown parameters after going through all the training samples once, SGD updates the networks' parameters after every training sample. Thus, overall it is faster than batch gradient descent and more suitable for online learning [14]. For massive neural networks or highly redundant training data, Tieleman & Hinton [15] showed that the mini-batch training is more preferable than the online SGD or full batch gradient descent. The mini-batch gradient descent compromises between SGD and batch gradient descent and updates the unknown parameters every mini-batch of training samples. As a result, less computation is required for training as indicated in [15], and the variance of the weights is reduced hence better convergence property [14]. Besides SGD algorithm, among other efficient and iterative approach is Root Mean Square Propagation (RMSProp) [15], which utilizes the moving average of the squared gradients to adapt the learning rate in order to speed up the training. It outperforms the non-adaptive SGD and the SGD with momentum [16], and also works well in online and non-stationary learning environments [17]. However, RMSProp lacks bias-correction and momentum, in which case the uncorrected bias results in a large step size and possibility of divergence [14]. Adaptive moment estimation (or Adam) [18], which combines RMSProp with bias-correction and momentum, estimates the first and the second order moments to compute learning rates for different parameters. With the added bias correction, Adam outperforms RMSProp towards the end of optimization as the gradients of the parameters become sparser (e.g., the gradient vectors from the ReLU activation function [19]), which benefits Adam by giving faster convergence rate than SGD and RMSProp [14, 18]. Nevertheless, it has been shown that both RMSProp and Adam have a poor generalization ability compared to SGD [20]. They work well in the initial portion of the training phase but get worse toward the later stages of training [20]. In addition, when training deep neural networks, RMSProp and Adam may diverge due to the exponential moving average used in them [21, 22].

In [11], a fast and recursive learning algorithm for BPNNs was developed using RLS adaptive algorithm. However, they assumed the threshold-logic non-linearity and further the method was applied to standard BPNNs. A more general RLS-based training approach for BPNNs was later developed in [12], which was shown to be more efficient than the conventional training methods. The fast convergence property of RLS algorithm can be made use of to speed up the training of other networks like CNNs. On the other hand, introducing more general non-linearity in activation functions and mini-batch training could potentially improve the performance of networks.

1.3 Contributions of the Present Work

The main contributions of this work include: (a) development of a more general recursive learning rule for neural networks with mini-batch extension, (b) application of the proposed RLS-based learning rule for CNNs and the study of its computational complexity and (c) the performance analysis of the proposed method on two datasets, namely MNIST and Fashion-MNIST. The key contribution of this work is to develop the detailed derivation of the general RLS-based learning rule for CNNs and similar networks used in machine learning problems.

In order to introduce the proposed RLS-based method, a detailed review of CNNs and three main training approaches, i.e., SGD [5], RMSProp [15] and Adam [18], are first presented. The review of CNNs includes the explicit forward-pass equations for the convolutional layers, the pooling layers, and the fully connected layers. We then show how the back-propagation learning can be extended to CNNs. Next, we apply the RLS learning rule to CNNs and present the derivation of the recursive updating equations for the convolutional layers and the final fully connected layers. The proposed RLS-based approach on CNNs is then extended to the mini-batch case and compared with other three main learning approaches in terms of their computational complexity and convergence speeds. After that, the proposed RLS-based training method is implemented on two datasets to evaluate the accuracy and convergence speed performance. Finally, observations and conclusions are made based on the simulation results.

1.4 Thesis Organization

This thesis is organized as follows: In Chapter 2, a full review of the CNN structure and the three aforementioned training approaches are presented. The review includes forward pass and the calculation of the gradients of the unknown parameters via error back-propagation method. The updating equations for each approach are then given. Chapter 3 provides the general RLS learning and its application to CNNs. For training CNNs recursively, the derivation follows those in Chapter 2 and the recursive updating equations for each unknown parameter are obtained. Furthermore, the recursive updating equations for mini-batch are derived explicitly and the computational complexity is analyzed. Chapter 4 presents the implementation results and analysis of the trained CNNs using the proposed RLS-based method on two datasets, i.e., MNIST and Fashion-MNIST. Finally, in Chapter 5, conclusions on the proposed method are made and some potential future works are discussed.

Chapter 2

Review of Convolutional Neural Networks and Training Approaches

2.1 Introduction

Image recognition using large fully connected conventional multi-layer back-propagation neural networks (BPNNs) needs massive number of parameters to be found during the learning phase. Training of such a large network with a huge number of parameters requires an overwhelmingly large training dataset and hence increased computational costs. Additionally, large BPNNs with a huge number of parameters generally lead to poor generalization performance on the unseen data. Also, the objects of interest can be anywhere in the images and have different sizes. Although the images can be scaled and centered, this pre-processing cannot always be perfect. Thus, in order to train a BPNN successfully, the training samples need to cover every possible location of the objects in the images. Besides, spatially adjacent pixels are highly correlated. Therefore, when the images are reshaped into vectors and applied as the input to BPNNs, the topology of the objects is completely lost.

On the contrary, convolutional neural networks (CNNs) use a weight-sharing scheme to reduce the number of parameters. These shared weights can be regarded as filter taps, and the filtered images (called *feature maps*) are the convolution results of the input images with the filter taps. Another benefit of CNNs is that, the spatial information can be made use of, thus the properties of the objects (e.g., edges, corners) can be retained. After the convolutions are done or the feature maps have been extracted, the most crucial information is the relative location to other features rather than the exact location in the image [7]. In other words, the precise positions wouldn't be needed to identify the pattern as they may confuse the network since the positions might vary for different features. On the other hand, as indicated in [3], the higher level the features are, the lesser precise location information is need for classification purposes. Moreover, the reduced precision is even advantageous because with the high precision, a slight distortion or translation would be a detriment to the representation. The pooling layers in CNNs can reduce the precise position of the feature maps by performing pooling operations to obtain the *pooled feature maps*. As a result, the resolutions of the feature maps are reduced so the networks become less sensitive to the shifts and distortion. In this way, one can note the similarity between the CNN and the filter banks in the discrete wavelet transform (DWT) [23,24]. They both use filters to extract information of the input and sub-sample the filtered results to obtain feature maps or detail coefficients. But CNNs use training to find the optimal filter parameters while the filters in the wavelet transform are generally pre-defined.

In this chapter, we review the CNNs forward and backward passes. First, the forward pass equations for convolutional, pooling and fully connected layers are provided. Then, the existing gradient-based training approaches for CNNs that use error back-propagation are presented. In particular, we review the three main training approaches, including stochastic gradient descent (SGD) [2,3,5,13], root mean square propagation (RMSProp) [15], and adaptive moment estimation (Adam) [18] and their properties and pros and cons.

2.2 CNN Forward Pass

In this section, the forward pass operations of the CNNs are reviewed. Starting from the input image, we propagate it through multiple convolutional and pooling layers to extract low-, intermediate- and high-level features. Each convolutional layer has a set of filters that generate filtered outputs via local convolution operation. Then, two types of pooling functions may be used to sub-sample the output of each convolutional layer.

A CNN and its layers were shown in Figure 1.1. Given a grayscale input image (single channel case), the convolution is performed using a set of filters and an activation function to extract the features from the input image. Consequently, the total number of such feature maps is equal to the number of filters in the first layer. However, if the input is a color image (red, green, and blue channels), the convolution and activation can still be performed to all the channels independently, and then the results are combined to yield a decision on the color image.

After the convolution and non-linear activation operations, the pooling layer sub-samples each filtered feature map from the previous layer, resulting in smaller size pooled feature maps. This pooling layer doesn't require any weights but reduces the resolution and the size of the feature maps. The outputs (or pooled feature maps) from the first pooling layer are applied as inputs to the next convolutional layer and this process is repeated layer-by-layer until the last pooling layer, the outputs of which are applied to a fully connected network for final decision-making. Note that while the size of the pooled feature maps at the last pooling layer is much smaller than the size of the original input image, the total number of the pooled feature maps becomes larger than the number of them at the previous layers.

The outputs of the last layer are flattened before applying them to the final fully connected layer. Although only two fully connected layers are shown in Figure 1.1, more fully connected layers can be added between the flattened layer and the output layer. During the training phase, the outputs of the fully connected layer are used along with the desired (target) output values to generate the mean squared error (MSE) cost function for minimization.

2.2.1 Convolutional and Pooling Layers

In this section, we derive the convolution and pooling equations for the l^{th} layer where $l \in [1, L]$. Suppose there are A pooled feature maps at the $(l-1)^{\text{th}}$ layer and B feature maps of size $I \times I$ and pooled feature maps of size $J \times J$ at the l^{th} layer, the detailed connections between them are shown in Figure 2.1.

For $t \in [1, T]$, $a \in [1, A]$, $b \in [1, B]$, $i \in [1, I^2]$, at the training sample t, let $net_{b,i}^{(l)}(t)$ be the convolution sum result before the activation function of the i^{th} pixel in the b^{th} feature map at the l^{th} layer, vector $\boldsymbol{z}_{a,i}^{(l-1)}(t)$ be the vectorized i^{th} input image block from the a^{th} pooled feature map at the $(l-1)^{\text{th}}$ layer, and vector $\boldsymbol{k}_{a,b}^{(l)}(t) \in \mathbb{R}^{K_l}$ be the filter coefficients including the bias term, which filters the a^{th} pooled feature map at the $(l-1)^{\text{th}}$ pooling layer to yield the b^{th} feature map at the l^{th}

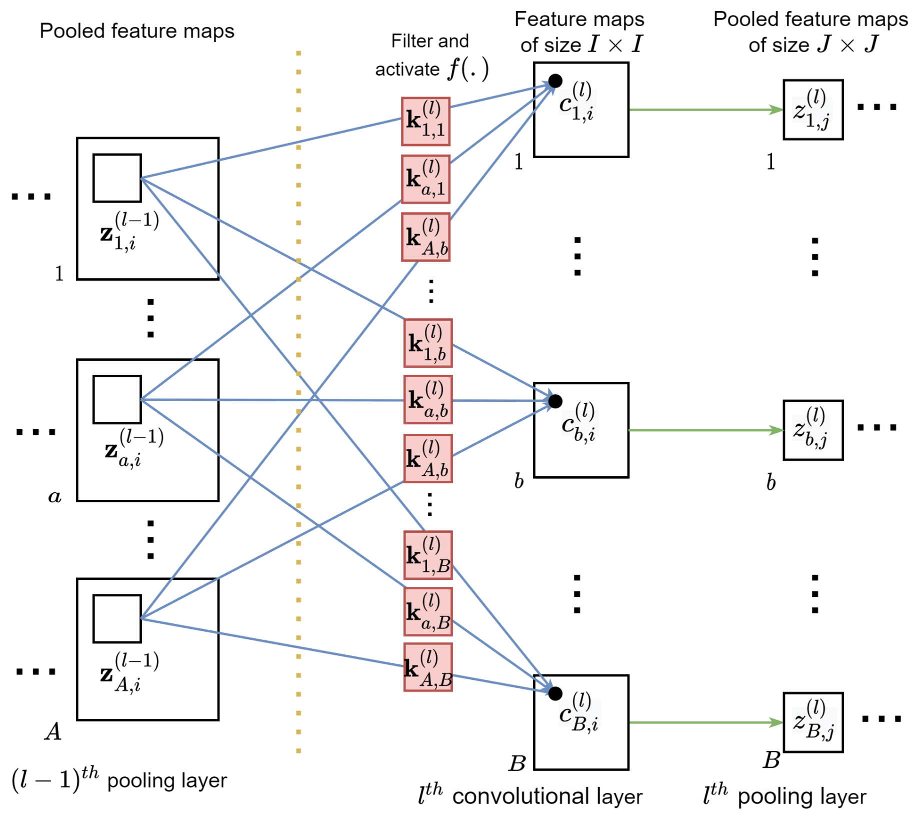


Figure 2.1: Connections between $(l-1)^{\text{th}}$ and l^{th} layers in CNN. The filter $\boldsymbol{k}_{a,b}^{(l)}$ convolves the a^{th} pooled feature map at $(l-1)^{\text{th}}$ pooling layer to yield the b^{th} feature map (after activation function) at the l^{th} convolutional layer. The feature maps are then pooled by a moving mask $\mathcal{S}$ leading to the pooled feature maps.

convolutional layer with K_l being the size of the filter including the bias. Let $f(.)$ be the activation function at nodes, then we have

$$net_{b,i}^{(l)}(t) = \sum_{a=1}^{A} \boldsymbol{z}_{a,i}^{(l-1)^T}(t)\boldsymbol{k}_{a,b}^{(l)}(t), \tag{2.1}$$

and

$$c_{b,i}^{(l)}(t) = f(net_{b,i}^{(l)}(t)), \tag{2.2}$$

where $c_{b,i}^{(l)}(t)$ is the resultant i^{th} pixel of the b^{th} feature map at the l^{th} convolutional layer. For $l = 1$, i.e., the first convolutional layer, and $A = 1$ for the first layer (1 channel), $\boldsymbol{z}_{1,i}^{(0)}(t)$ is the vectorized i^{th} block of the t^{th} training image.

The pooling operation follows the local convolution operations performed at the convolutional layer. The pooling functions could be either max pooling, averaging, and sub-sampling operations. For $i_1, i_2 \in [1, I]$, $j_1, j_2 \in [1, J]$, if the pooling mask of size $S_1 \times S_2$ is denoted by $\mathcal{S}$, then the max pooling is given by

$$z^{(l)}_{b,(j_1,j_2)}(t) = \max_{i_1,i_2 \in \mathcal{S}} \{c^{(l)}_{b,(i_1,i_2)}(t)\}, \tag{2.3}$$

and the average pooling is given by

$$z^{(l)}_{b,(j_1,j_2)}(t) = \frac{1}{|\mathcal{S}|} \sum_{i_1,i_2 \in \mathcal{S}}\sum c^{(l)}_{b,(i_1,i_2)}(t), \tag{2.4}$$

where $c^{(l)}_{b,(i_1,i_2)}(t)$ and $z^{(l)}_{b,(j_1,j_2)}(t)$ are the tripled-indexed $(i_1, i_2)^{\text{th}}$ and $(j_1, j_2)^{\text{th}}$ pixels in the b^{th} feature map and pooled feature map at l^{th} layer, $|\mathcal{S}|$ is the cardinality of $\mathcal{S}$. Typically, $\mathcal{S}$ is of size 2-by-2 and moves across the feature maps. Note that here we used tripled-indexed notation. However, the resulting triple indexing images $c^{(l)}_{b,(i_1,i_2)}(t)$ and $z^{(l)}_{b,(j_1,j_2)}(t)$ can be reshaped into double indexing $c^{(l)}_{b,i}(t)$ and $z^{(l)}_{b,j}(t)$ by $i = I(i_1 - 1) + i_2$, $j = J(j_1 - 1) + j_2$.

Finally, the pooled feature maps at the l^{th} become the inputs to the $(l + 1)^{\text{th}}$ convolutional layer to produce the feature maps and the corresponding pooled feature maps. This procedure can be repeated until the last convolutional and pooling layers, the output of which are flattened into 1-D array before applying to the fully connected layer.

2.2.2 Fully Connected Layer and Cost Function

Let $\boldsymbol{y}$ be the input vector including the bias term of the fully connected layer and suppose there are M feature maps and pooled feature maps of size $N \times N$ at the last convolutional and pooling layer. Define $h := N^2(m - 1) + n$ where $n \in [1, N^2]$, $m \in [1, M]$, then the h^{th} element y_h of vector $\boldsymbol{y}$ is

$$y_h(t) = z^{(L)}_{m,n}(t), \tag{2.5}$$

where $z^{(L)}_{m,n}(t)$ is the n^{th} pixel of the m^{th} pooled feature map at the last pooling layer. In addition, let $y_{MN^2+1}(t) = 1$ for the convenience of vector product with the bias term in the weight vectors.

If we define $F(.)$ to be the concatenating operator, then we can write

$$\boldsymbol{y}(t) = F(\{z_{m,n}^{(L)}(t)\}_{m=1,\cdots,M,\ n=1,\cdots,N^2}). \tag{2.6}$$

As for the reverse process, given $\boldsymbol{y} \in \mathbb{R}^{MN^2+1}$, we first delete the last term 1, then for the other elements,

$$z_{m,n}^{(L)}(t) = F_{m,n}^{-1}(\boldsymbol{y}(t)) = y_{N^2(m-1)+n}(t), m \in [1, m], n \in [1, N^2]. \tag{2.7}$$

For the output of the fully connected layer with total of P nodes, the aggregated input to node $p \in [1, P]$ is

$$net_p^{(L+1)}(t) = \boldsymbol{y}^T(t)\boldsymbol{w}_p(t), \tag{2.8}$$

where $\boldsymbol{w}_p(t) \in \mathbb{R}^{MN^2+1}$ is the weight vector including the bias term. Also, the output of the node p is

$$o_p^{(L+1)}(t) = f(net_p^{(L+1)}(t)). \tag{2.9}$$

Once the aggregated output $o_p^{(L+1)}(t)$'s are generated, we can form the cost function which is minimized (or maximized) during the training to find the unknown filter weights of the convolutional layers as well as the weights of the fully connected layer. Suppose the desired output of the p^{th} neuron is $d_p(t)$. The cost function, that is commonly used to train BPNNs or CNNs, is the mean squared error (MSE) between the desired output $d_p(t)$ and the actual ones $o_p^{(L+1)}(t)$, i.e., after T iterations,

$$\mathcal{J}(T) := \frac{1}{2T}\sum_{t=1}^{T}\sum_{p=1}^{P}(d_p(t) - o_p^{(L+1)}(t))^2 = \frac{1}{2T}\sum_{t=1}^{T}\sum_{p=1}^{P} e_p^{(L+1)^2}(t), \tag{2.10}$$

where

$$e_p^{(L+1)}(t) := d_p(t) - o_p^{(L+1)}(t). \tag{2.11}$$

This cost function is minimized to yield (ideally) minimal MSE estimate of the network parameters. Alternatively, we can write the cost function as

$$\mathcal{J}(T) = \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}(t), \tag{2.12}$$

where

$$\mathcal{L}(t) := \frac{1}{2}\sum_{p=1}^{P} e_p^{(L+1)^2}(t). \tag{2.13}$$

Thus, $\mathcal{L}(t)$ is the sum squared error (SSE) over the single training sample t. Minimizing this cost function corresponds to iterative online learning vs. the "batch learning" in the former case. We will use the cost function in (2.13) to devise different learning algorithms for CNNs along with the error back-propagation, which are described next.

2.3 Error Back-Propagation of CNN

This section demonstrates how to update the unknown parameters of CNN via minimizing the cost function in (2.13) and error back-propagation method [1, 2, 25]. As far as the gradient-based methods are concerned, we need to derive the explicit equations for the partial derivatives of the cost function $\mathcal{L}(t)$ in (2.13) w.r.t. the weight vectors of the fully connected layer and those of the convolutional layers.

Starting from the fully connected layer, taking the partial derivative of $\mathcal{L}(t)$ w.r.t. $\boldsymbol{w}_p(t)$, using equations in (2.8), (2.9), (2.11) and (2.13), and applying the chain rule, we obtain

$$\begin{aligned}
\frac{\partial\mathcal{L}(t)}{\partial\boldsymbol{w}_p(t)} &= \frac{\partial e_p^{(L+1)}(t)}{\partial\boldsymbol{w}_p(t)} e_p^{(L+1)}(t) \\
&= [\frac{\partial e_p^{(L+1)}(t)}{\partial o_p^{(L+1)}(t)} \frac{\partial o_p^{(L+1)}(t)}{\partial net_p^{(L+1)}(t)} \frac{\partial net_p^{(L+1)}(t)}{\partial\boldsymbol{w}_p(t)}] e_p^{(L+1)}(t) \\
&= -f'(net_p^{(L+1)}(t))\boldsymbol{y}(t)e_p^{(L+1)}(t) \\
&= -f'(net_p^{(L+1)}(t))\boldsymbol{y}(t)(d_p(t) - f(\boldsymbol{y}^T(t)\boldsymbol{w}_p(t)),
\end{aligned} \tag{2.14}$$

where $f'(.)$ is the first-order derivative of the activation function.

Next, we back-propagate the errors to the input nodes of the fully connected layer through the weight vector. To do that, we find the partial derivative of $\mathcal{L}(t)$ w.r.t. the input vector of the fully connected layer,

$$\begin{aligned}\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{y}(t)} &= \sum_{p=1}^{P} \frac{\partial e_p^{(L+1)}(t)}{\partial \boldsymbol{y}(t)} e_p^{(L+1)}(t) \\ &= -\sum_{p=1}^{P} f'(net_p^{(L+1)}(t)) \boldsymbol{w}_p(t) e_p^{(L+1)}(t). \end{aligned} \tag{2.15}$$

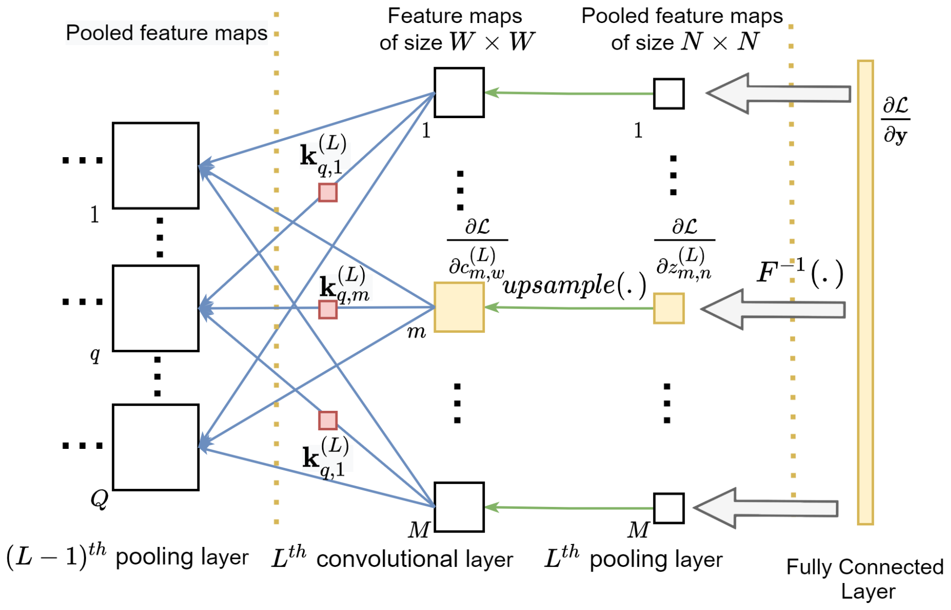

Pooled feature maps
Feature maps of size $W \times W$
Pooled feature maps of size $N \times N$
$\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$
$\mathbf{k}_{q,1}^{(L)}$
$\mathbf{k}_{q,m}^{(L)}$
$\mathbf{k}_{q,1}^{(L)}$
1
1
1
q
Q
m
M
M
$\frac{\partial \mathcal{L}}{\partial c_{m,w}^{(L)}}$
$\frac{\partial \mathcal{L}}{\partial z_{m,n}^{(L)}}$
$upsample(.)$
$F^{-1}(.)$
$(L-1)^{th}$ pooling layer
L^{th} convolutional layer
L^{th} pooling layer
Fully Connected Layer

Figure 2.2: Error back-propagation from the fully connected layer to the last convolutional and the pooling layers. The back-propagated errors from the fully connected layer are first reshaped and then upsampled. We then back-propagate the error to the $(L-1)^{\text{th}}$ pooling layer.

However, $\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{y}(t)}$ is in the form of a flattened long vector from the fully connected layer. Recall that there are M feature maps of size $W \times W$ and pooled feature maps of size $N \times N$ at the last convolutional and pooling layers, respectively as shown in Figure 2.2. Thus, we need to reshape it back as M pooled feature maps of size $N \times N$. According to the reverse concatenation operator in (2.7), the partial derivative of $\mathcal{L}(t)$ w.r.t. the n^{th} pixel in the m^{th} pooled feature map $\frac{\partial \mathcal{L}(t)}{\partial z_{m,n}^{(L)}(t)}$ is

$$\frac{\partial \mathcal{L}(t)}{\partial z_{m,n}^{(L)}(t)} = F_{m,n}^{-1}(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{y}(t)}), m \in [1, m], n \in [1, N^2]. \tag{2.16}$$

where $\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{y}(t)}$ can be obtained via (2.15).

Now, the errors are back-propagated to the L^{th} pooling layer. Since the pooled feature maps are sub-sampled from the feature maps, the partial derivatives of $\mathcal{L}(t)$ w.r.t. the pixels in the convolutional layer can be obtained by upsampling the partial derivatives of $\mathcal{L}(t)$ w.r.t. the pixels in the pooled feature maps.

Specifically, we first reshape $\frac{\partial \mathcal{L}(t)}{\partial z_{m,n}^{(L)}(t)}$ into the triple-indexed form with the $(n_1, n_2)^{\text{th}}$ pixel $\frac{\partial \mathcal{L}(t)}{\partial z_{m,(n_1,n_2)}^{(L)}(t)}$ using $n = N(n_1 - 1) + n_2$, where $n_1, n_2 \in [1, N]$. Recall that the pooling mask $\mathcal{S}$ is of size $S_1 \times S_2$. Then, the partial derivative of $\mathcal{L}(t)$ w.r.t. the $(w_1, w_2)^{\text{th}}$ pixel in the m^{th} feature map $\frac{\partial \mathcal{L}(t)}{\partial c_{m,(w_1,w_2)}^{(L)}(t)}$ is

$$\frac{\partial \mathcal{L}(t)}{\partial c_{m,(w_1,w_2)}^{(L)}(t)} = \frac{\partial \mathcal{L}(t)}{\partial z_{m,(\lceil w_1/S_1 \rceil, \lceil w_2/S_2 \rceil)}^{(L)}(t)}, w_1, w_2 \in [1, W], \tag{2.17}$$

where $\lceil . \rceil$ denotes the ceiling function. Finally, the tripled-indexed $\frac{\partial \mathcal{L}(t)}{\partial c_{m,(w_1,w_2)}^{(L)}(t)}$ can be reshaped back to the double-indexed form for the convenience of the next derivation using $w = W(w_1 - 1) + w_2$. If we define $upsample(.)$ operator to be the upsampling and reshaping process above, then we can write

$$\frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} = upsample(\frac{\partial \mathcal{L}(t)}{\partial z_{m,n}^{(L)}(t)}), n \in [1, N^2], w \in [1, W^2]. \tag{2.18}$$

where $\frac{\partial \mathcal{L}(t)}{\partial z_{m,n}^{(L)}(t)}$ can be found using (2.16).

Now, we are ready to find the partial derivative of $\mathcal{L}(t)$ w.r.t. the filter vector $\boldsymbol{k}_{q,m}^{(L)}(t)$, which connects the q^{th}, $q \in [1, Q]$, pooled feature map at the $(L-1)^{\text{th}}$ layer to yield the m^{th}, $m \in [1, M]$, feature map at the L^{th} layer. To demonstrate it, we write down the forward pass equations from the $(L-1)^{\text{th}}$ pooling layer to the L^{th} convolutional layer

$$net_{m,w}^{(L)}(t) = \sum_{q=1}^{Q} \boldsymbol{z}_{q,w}^{(L-1)^T}(t) \boldsymbol{k}_{q,m}^{(L)}(t), \tag{2.19}$$

where $\boldsymbol{z}_{q,w}^{(L-1)}(t)$ is the vectorized block of the qth pooled feature map at $(L-1)$th pooling layer for the wth pixel of the feature maps at the Lth layer. Also,

$$c_{m,w}^{(L)}(t) = f(net_{m,w}^{(L)}(t)). \tag{2.20}$$

Because of the difficulty in expressing the reshaped and upsampled back-propagated errors in a close form, we make use of the chain rule, and consider $c_{m,w}^{(L)}(t)$'s as intermediate variables to find the partial derivative of $\mathcal{L}(t)$ w.r.t. $\boldsymbol{k}_{q,m}^{(L)}(t)$. Note all $m \in [1, M]$, there are MW^2 such variables in total. Thus, we can write

$$\begin{aligned}
\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{q,m}^{(L)}(t)} &= \sum_{m'=1}^{M} \sum_{w=1}^{W^2} \frac{\partial \mathcal{L}(t)}{\partial c_{m',w}^{(L)}(t)} \frac{\partial c_{m',w}^{(L)}(t)}{\partial \boldsymbol{k}_{q,m}^{(L)}(t)} \\
&= \sum_{w=1}^{W^2} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} \frac{\partial c_{m,w}^{(L)}(t)}{\partial \boldsymbol{k}_{q,m}^{(L)}(t)} \\
&= \sum_{w=1}^{W^2} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} \frac{\partial c_{m,w}^{(L)}(t)}{\partial net_{m,w}^{(L)}(t)} \frac{\partial net_{m,w}^{(L)}(t)}{\partial \boldsymbol{k}_{q,m}^{(L)}(t)} \\
&= \sum_{w=1}^{W^2} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} f'(net_{m,w}^{(L)}(t)) \boldsymbol{z}_{q,w}^{(L-1)}(t).
\end{aligned} \tag{2.21}$$

where $\frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)}$ can be calculated using (2.18). The summation over all M feature maps can be reduced to the mth feature map in the second line since $\boldsymbol{k}_{q,m}^{(L)}(t)$ only leads to the mth feature map and does not connect to other feature maps.

Finally, we back-propagate the error from the Lth convolutional layer to the $(L-1)$th pooling layer thought the filters, so that we can derive the error back-propagation at the $(L-1)$th layer. To do this, we first find the partial derivative of $\mathcal{L}(t)$ w.r.t. the input block $\boldsymbol{z}_{q,w}^{(L-1)}(t)$ by considering $c_{m,w}^{(L)}(t)$'s as the intermediate variables. This gives

$$\begin{aligned}\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{z}_{q,w}^{(L-1)}(t)} &= \sum_{m=1}^{M}\sum_{w'=1}^{W^2} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w'}^{(L)}(t)} \frac{\partial c_{m,w'}^{(L)}(t)}{\partial \boldsymbol{z}_{q,w}^{(L-1)}(t)} \\ &= \sum_{m=1}^{M} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} \frac{\partial c_{m,w}^{(L)}(t)}{\partial \boldsymbol{z}_{q,w}^{(L-1)}(t)} \\ &= \sum_{m=1}^{M} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} \frac{\partial c_{m,w}^{(L)}(t)}{\partial net_{m,w}^{(L)}(t)} \frac{\partial net_{m,w}^{(L)}(t)}{\partial \boldsymbol{z}_{q,w}^{(L-1)}(t)} \\ &= \sum_{m=1}^{M} \frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)} f'(net_{m,w}^{(L)}(t)) \boldsymbol{k}_{q,m}^{(L)}(t). \end{aligned} \tag{2.22}$$

where $\frac{\partial \mathcal{L}(t)}{\partial c_{m,w}^{(L)}(t)}$ can be calculated using (2.18). Note that the summation over all W^2 pixels is reduced to the w^{th} pixels only in each feature map because $\boldsymbol{z}_{q,w}^{(L-1)}(t)$ only leads to $c_{m,w}^{(L)}(t)$.

Then, $\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{z}_{q,w}^{(L-1)}(t)}$ is reshaped into a block at w^{th} location, i.e., the location of the input block corresponding to the reshaped vector $\boldsymbol{z}_{q,w}^{(L-1)}(t)$. Furthermore, after doing it for all w values, the partial derivatives of $\mathcal{L}(t)$ w.r.t. the pixels in the q^{th} pooled feature map at the $(L-1)^{\text{th}}$ pooling layer can be obtained by summing over all W^2 blocks. Finally, by doing it for all q values, the errors are back-propagated from the L^{th} convolutional layer to the $(L-1)^{\text{th}}$ pooling layer, and the error back-propagation is ready to be applied to the $(L-1)^{\text{th}}$ layer.

Next, we focus on the error back-propagation from l^{th} layer, $l \in [1, L-1]$, to $(l-1)^{\text{th}}$ layer as shown in Figure 2.3. Suppose the errors have been back-propagated to the l^{th} pooling layer. In other words, $\frac{\partial \mathcal{L}(t)}{\partial z_{b,j}^{(l)}(t)}$'s have been obtained, $b \in [1, B]$, $j \in [1, J^2]$. According to the $upsample(.)$ operator defined in (2.18), the partial derivative of $\mathcal{L}(t)$ w.r.t. the i^{th} pixel of the b^{th} feature map is

$$\frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)} = upsample(\frac{\partial \mathcal{L}(t)}{\partial z_{b,j}^{(l)}(t)}), i \in [1, I^2]. \tag{2.23}$$

By doing it for all i and b values, the errors can be back-propagated to the l^{th} convolutional layer and we have all $\frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)}$'s.

Next, we derive the partial derivative of $\mathcal{L}(t)$ w.r.t. $\boldsymbol{k}_{a,b}^{(l)}(t)$. Similar to the derivation for the L^{th} layer, we consider $c_{b,i}^{(l)}(t)$'s as in total of BI^2 intermediate variables and use the chain rule, i.e.,

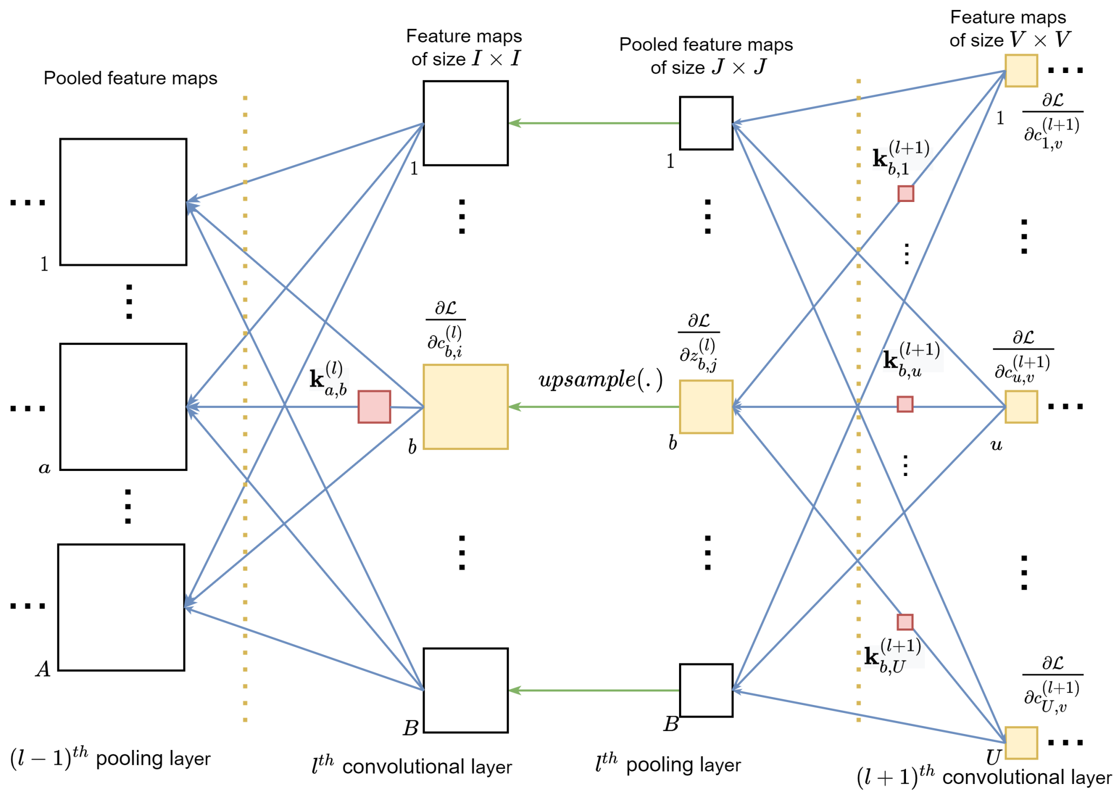

Pooled feature maps
Feature maps of size $I \times I$
Pooled feature maps of size $J \times J$
Feature maps of size $V \times V$
$\mathbf{k}_{a,b}^{(l)}$
$\frac{\partial \mathcal{L}}{\partial c_{b,i}^{(l)}}$
$upsample(.)$
$\frac{\partial \mathcal{L}}{\partial z_{b,j}^{(l)}}$
$\mathbf{k}_{b,1}^{(l+1)}$
$\mathbf{k}_{b,u}^{(l+1)}$
$\mathbf{k}_{b,U}^{(l+1)}$
$\frac{\partial \mathcal{L}}{\partial c_{1,v}^{(l+1)}}$
$\frac{\partial \mathcal{L}}{\partial c_{u,v}^{(l+1)}}$
$\frac{\partial \mathcal{L}}{\partial c_{U,v}^{(l+1)}}$
$(l-1)^{th}$ pooling layer
l^{th} convolutional layer
l^{th} pooling layer
$(l+1)^{th}$ convolutional layer

Figure 2.3: Error back-propagation at the l^{th} layer. Errors at the $(l+1)^{\text{th}}$ convolutional layer are first back-propagated through the filters at this layer to the l^{th} pooling layer, then upsampled to the l^{th} convolutional layer. Finally, they back-propagation to the $(l-1)^{\text{th}}$ pooling layer through the filters at l^{th} layer. All the nodes involving $\boldsymbol{k}_{a,b}^{(l)}(t)$ are colored as orange.

$$\begin{aligned}
\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} &= \sum_{b'=1}^{B}\sum_{i=1}^{I^2} \frac{\partial \mathcal{L}(t)}{\partial c_{b',i}^{(l)}(t)} \frac{\partial c_{b',i}^{(l)}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} \\
&= \sum_{i=1}^{I^2} \frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)} \frac{\partial c_{b,i}^{(l)}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} \\
&= \sum_{i=1}^{I^2} \frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)} \frac{\partial c_{b,i}^{(l)}(t)}{\partial net_{b,i}^{(l)}(t)} \frac{\partial net_{b,i}^{(l)}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} \\
&= \sum_{i=1}^{I^2} \frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)} f'(net_{b,i}^{(l)}(t)) \boldsymbol{z}_{a,i}^{(l-1)}(t), \qquad (2.24)
\end{aligned}$$

where $\frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)}$ can be calculated using (2.23). Again, the summation over B is reduced to b because $\boldsymbol{k}_{a,b}^{(l)}(t)$ only leads to the b^{th} feature map and doesn't connect to other feature maps. Also, $\boldsymbol{z}_{a,i}^{(l-1)}(t)$ is the vectorized block of the a^{th} pooled feature map at $(l-1)^{\text{th}}$ pooling layer for the i^{th} pixel of the l^{th}-layer feature maps. Note that the partial derivative of $\mathcal{L}(t)$ w.r.t. $\boldsymbol{k}_{q,m}^{(L)}(t)$ in (2.21) can be merged here. Therefore, for the next updating methods, we only focus on (2.24) and $l \in [1, L]$.

Finally, if the l^{th} layer is not the first layer of CNN, we need to back-propagate the error from the l^{th} convolutional layer to the $(l-1)^{\text{th}}$ pooling layer through the filters, so that the error back-propagation can be derived at the $(l-1)^{\text{th}}$ layer. This is done in a similar manner as for the previous case. This error back-propagation procedure can be repeated until the first layer to find the partial derivatives of $\mathcal{L}(t)$ w.r.t. all the parameters in CNN.

Although we have derived $\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}$ in (2.24), it is difficult to write it in a close form or expand $\frac{\partial \mathcal{L}(t)}{\partial c_{b,i}^{(l)}(t)}$ explicitly due to the changed indices from upsampling operator, the block summation in the back-propagation from the $(l+1)^{\text{th}}$ convolutional layer to the l^{th} pooling layer, etc. Thus, it is more convenient to back-propagate the errors layer-by-layer and eventually find $\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}$. Furthermore, in the next section, we will introduce how to update these parameters of the convolutional and the fully connected layers based on the partial derivatives w.r.t. them but without expanding them explicitly.

2.4 Training Methods for CNN

After obtaining the partial derivatives of the cost function w.r.t. the unknown parameter vectors in (2.14) and (2.24), we can make use of them and derive the updating equations for training CNN using different methods. In this section, three main learning methods are reviewed, including stochastic gradient descent (SGD) [2,3,5,13], root mean square propagation (RMSProp) [15], and the adaptive moment estimation (Adam) [18]. All these methods need to initialize the parameters randomly at the beginning of the training, and assign a proper learning rate, which is denoted by η in this section. The updating equations are given without expanding the explicit forms of the partial derivatives of $\mathcal{L}(t)$ w.r.t. the unknown parameter vectors due to the reason mentioned above. In addition, these updating equations can be implemented for multiple epochs until convergence.

2.4.1 Stochastic Gradient Descent and Momentum Methods

Stochastic gradient descent (SGD) [2, 3, 5, 13] is the most commonly used method for training CNNs and BPNNs, which updates the networks' parameters after the application of each training sample hence enabling online learning. For $t \in [1, T]$, the updating equations for the parameters of the final fully connected layer and the lth convolutional layer are, respectively

$$\boldsymbol{w}_p(t) = \boldsymbol{w}_p(t-1) - \eta \frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)}, p \in [1, P], \tag{2.25}$$

$$\boldsymbol{k}_{a,b}^{(l)}(t) = \boldsymbol{k}_{a,b}^{(l)}(t-1) - \eta \frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}, a \in [1, A], b \in [1, B], l \in [1, L]. \tag{2.26}$$

The initial $\boldsymbol{w}_p(0)$, $\boldsymbol{k}_{a,b}^{(l)}(0)$ are initialized randomly. Same initialization is done for the next two methods as well.

To accelerate the convergence of SGD, a momentum term can be added to the SGD updating equation [14]. If γ denotes the momentum factor, then the updating equations of the SGD with the momentum term for the parameters of the final fully connected layer and the lth convolutional layer become,

$$\Theta_p^{(L+1)}(t) = \gamma \Theta_p^{(L+1)}(t-1) + \eta \frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)}, \tag{2.27}$$

$$\boldsymbol{w}_p(t) = \boldsymbol{w}_p(t-1) - \Theta_p^{(L+1)}(t), \tag{2.28}$$

$$\Theta_{a,b}^{(l)}(t) = \gamma \Theta_{a,b}^{(l)}(t-1) + \eta \frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}, \tag{2.29}$$

$$\boldsymbol{k}_{a,b}^{(l)}(t) = \boldsymbol{k}_{a,b}^{(l)}(t-1) - \Theta_{a,b}^{(l)}(t), \tag{2.30}$$

respectively, where $\Theta_p^{(L+1)}(0)$ and $\Theta_{a,b}^{(l)}(0)$ can be initialized to zero vectors. The momentum factor γ can be set to 0.9 as recommended in [14]. One can notice that when $\gamma = 0$, the updating equations become the standard SGD updating.

2.4.2 RMSProp Method

RMSProp [15] introduces the moving average of the squared gradients (elementwise squared for vectors) into the updating equations. It is also well-known for its online and non-stationary learning abilities [17]. For $p \in [1, P]$, $a \in [1, A]$, and $b \in [1, B]$, $l \in [1, L]$, the weight updating equations for the final fully connected layer, and the l^{th} convolutional layer of CNN are, respectively

$$M_p^{(L+1)}(t) = \beta M_p^{(L+1)}(t-1) + (1-\beta)(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2, \tag{2.31}$$

$$\boldsymbol{w}_p(t) = \boldsymbol{w}_p(t-1) - \frac{\eta}{\sqrt{M_p^{(L+1)}(t)}} \frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)}, \tag{2.32}$$

$$M_{a,b}^{(l)}(t) = \beta M_{a,b}^{(l)}(t-1) + (1-\beta)(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)})^2, \tag{2.33}$$

$$\boldsymbol{k}_{a,b}^{(l)}(t) = \boldsymbol{k}_{a,b}^{(l)}(t-1) - \frac{\eta}{\sqrt{M_{a,b}^{(l)}(t)}} \frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}, \tag{2.34}$$

where $M_p^{(L+1)}(t)$ and $M_{a,b}^{(l)}(t)$ are the moving average of the squared gradients, which can be initialized as numbers close to zero, and β is known as the discounting factor which determines the weights in the convex-combinations in (2.31) and (2.33). The recommended choice for $\beta = 0.9$ [15].

2.4.3 Adam Method

Adam [18] combines RMSProp with bias-correction and momentum terms, and estimates the first- and the second-order moments of the unknown parameters' gradients to compute learning rates for different parameters via scaling the learning rate by the squared gradients (elementwise squared for vectors) of the unknown parameters. According to [18], for $p \in [1, P]$, $a \in [1, A]$, and $b \in [1, B]$, the updating equations for the final fully connected layer, and the l^{th} convolutional layer l of CNN are, respectively

$$M_p^{(L+1)}(t) = \beta_1 M_p^{(L+1)}(t-1) + (1-\beta_1)\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)}, \tag{2.35}$$

$$V_p^{(L+1)}(t) = \beta_2 V_p^{(L+1)}(t-1) + (1-\beta_2)(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2, \tag{2.36}$$

$$\hat{M}_p^{(L+1)}(t) = \frac{M_p^{(L+1)}(t)}{1-\beta_1^t}, \tag{2.37}$$

$$\hat{V}_p^{(L+1)}(t) = \frac{V_p^{(L+1)}(t)}{1-\beta_2^t}, \tag{2.38}$$

$$\boldsymbol{w}_p(t) = \boldsymbol{w}_p(t-1) - \eta \frac{\hat{M}_p^{(L+1)}(t)}{\sqrt{\hat{V}_p^{(L+1)}(t)} + \epsilon}, \tag{2.39}$$

$$M_{a,b}^{(l)}(t) = \beta_1 M_{a,b}^{(l)}(t-1) + (1-\beta_1)\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}, \tag{2.40}$$

$$V_{a,b}^{(l)}(t) = \beta_2 V_{a,b}^{(l)}(t-1) + (1-\beta_2)(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)})^2, \tag{2.41}$$

$$\hat{M}_{a,b}^{(l)}(t) = \frac{M_{a,b}^{(l)}(t)}{1-\beta_1^t}, \tag{2.42}$$

$$\hat{V}_{a,b}^{(l)}(t) = \frac{V_{a,b}^{(l)}(t)}{1-\beta_2^t}, \tag{2.43}$$

$$\boldsymbol{k}_{a,b}^{(l)}(t) = \boldsymbol{k}_{a,b}^{(l)}(t-1) - \eta \frac{\hat{M}_{a,b}^{(l)}(t)}{\sqrt{\hat{V}_{a,b}^{(l)}(t)} + \epsilon}, \tag{2.44}$$

where $M_p^{(L+1)}(t)$ and $M_{a,b}^{(l)}(t)$ are the estimated first-order moment of gradients; $V_p^{(L+1)}(t)$ and $V_{a,b}^{(l)}(t)$ are the estimated second-order moment of gradients [18]. They can be initialized as zero at the beginning of the training. $\hat{M}_p^{(L+1)}(t)$ and $\hat{M}_{a,b}^{(l)}(t)$ are the bias-corrected first-order moment of gradients; $\hat{V}_p^{(L+1)}(t)$ and $\hat{V}_{a,b}^{(l)}(t)$ are the bias-corrected second-order moment of gradients. Finally, β_1 and β_2 are the exponential decay rates, which are less than 1 and control the exponential decay rates of the moving averages, i.e., the first- and the second-order moment of gradients. In addition, they can be regarded as the convex-combination of the first- and second-order moments with the squared gradients; and ϵ is a small number added for numerical stability. The superscript t in β_1^t and β_2^t implies raising to power t, i.e., the denominators of (2.37), (2.38), (2.42) and (2.43) approach

1 as learning progresses, which implies less bias correction for the moment of gradients and more unbiased estimation. As recommend in [18], β_1 can be set to 0.9 and β_2 can be set to 0.999.

To see how the bias is corrected [18], as an example let $\boldsymbol{v}_w(t)$ be the second-order moment of the weight vector $\boldsymbol{w}_p(t)$. The exponential second-order moving average is

$$\boldsymbol{v}_w(t) = \beta_2 \boldsymbol{v}_w(t-1) + (1-\beta_2)(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2, \tag{2.45}$$

which can be written as a function of the gradients at all previous training samples:

$$\boldsymbol{v}_w(t) = (1-\beta_2)\sum_{\tau=1}^{t} \beta_2^{t-\tau}(\frac{\partial \mathcal{L}(\tau)}{\partial \boldsymbol{w}_p(\tau)})^2. \tag{2.46}$$

In order to know how the expection of the exponential second-order moving average $\mathbb{E}[\boldsymbol{v}_w(t)]$ relates to the true second-order moment $\mathbb{E}[(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2]$, so that we can correct the difference between them, we take $\mathbb{E}[.]$ on both sides, and assume that the accumulated exponential second-order moving average up to sample t can be approximated in terms of the exponential second-order moving average at sample t, i.e.,

$$\begin{aligned}\mathbb{E}[\boldsymbol{v}_w(t)] &= \mathbb{E}[(1-\beta_2)\sum_{\tau=1}^{t} \beta_2^{t-\tau}(\frac{\partial \mathcal{L}(\tau)}{\partial \boldsymbol{w}_p(\tau)})^2] \\ &= \mathbb{E}[(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2](1-\beta_2)\sum_{\tau=1}^{t} \beta_2^{t-\tau} + \xi \\ &= \mathbb{E}[(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2](1-\beta_2^t) + \xi. \end{aligned} \tag{2.47}$$

where ξ is the approximation error. Now, if $\mathbb{E}[(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2]$ is stationary, then $\xi = 0$; otherwise ξ can be kept small by choosing proper β_2 such that the assigned weight $(1-\beta_2)$ to the squared gradients in (2.45) is small. At the same time, we divide by $(1-\beta_2^t)$ on both sides to correct the initialization bias as,

$$\frac{\mathbb{E}[\boldsymbol{v}_w(t)]}{1-\beta_2^t} = \mathbb{E}[(\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{w}_p(t)})^2] + \frac{\xi}{1-\beta_2^t}, \tag{2.48}$$

which is corresponding to the bias correction in (2.38) and (2.43). Analogous derivation can be done for the first-order moment of gradients and the filter weights $\boldsymbol{k}_{a,b}^{(l)}(t)$.

2.5 Conclusion

In this chapter, we reviewed the details of convolutional neural network forward and backward passes as well as three training approaches. The derivation of the learning equations in this thesis is based on the mean squared error (MSE) as the cost function. There are, however, other choices for the cost function, e.g, cross-entropy [26, 27]. Although MSE is more robust to the most optimization scenarios, cross-entropy can lead to faster convergence and more optimal minima [28] as well as better posterior probability estimation of the training samples [29]. The adopted learning methods require obtaining the gradient of the cost function w.r.t. each unknown parameter vectors to be used in the error back-propagation. Also, some learning parameters such as learning rates need to be assigned in advance. The three covered training methods in this chapter are the most commonly used ones nowadays. However, all of them are searching methods, which means they have issues with local minima and trade-off between the speed of convergence and accuracy. Inspired by these issues, we introduce recursive least squares (RLS) learning for CNNs in the next chapter, which doesn't have the local minima or the speed-accuracy trade-off issues.

Chapter 3

Recursive Least Squares Learning for CNNs

3.1 Introduction

Recursive least squares (RLS) algorithm [8–10] is an adaptive algorithm to solve the least squares (LS) problem for transversal filter weights (taps) [8]. Generally, given the LS estimate of the weight vector of the filter at the time t, we would like to estimate the weight vector at time $t+1$ recursively based upon limited memory on the past data.

The RLS algorithm was developed to solve the convergence speed-accuracy trade-off problem of the standard least mean squares (LMS) [8–10] type algorithms. That is, the RLS algorithm provides much faster convergence property while it does not suffer from misadjustment issues [8–11]. However, the price to pay is the increased computational cost of the RLS algorithm [8–10].

In this chapter, a RLS-based training method is developed to train the unknown parameters of convolutional neural networks (CNNs). The chapter is organized as follows. In Section 3.2, the recursive learning rule is applied to CNNs by finding the recursive updating equations for the unknown parameter vectors. Then, Section 3.3 extends the learning rule to the mini-batch case, where the updating of the unknown parameters is based upon a mini-batch of samples instead of a single sample at a time. Section 3.4 presents the analysis of the computational complexity of the proposed RLS-based training approach and its comparison with the commonly used training methods including SGD, RMSProp and Adam covered in Chapter 2. Finally, the conclusion of the proposed RLS-based training approach is made in Section 3.5.

3.2 Training CNN using Recursive Least Squares

The derivation of the RLS algorithm in [8–10] starts from the partial derivatives of the cost function w.r.t. the unknown parameters and solving the normal equations recursively. This is done by re-writing the normal equations in terms of the correlation matrices and cross-correlation vectors and find the innovations and the gain matrices for RLS updating [8–11]. However, the recursive updating for the convolutional layers in CNN require a variation of RLS derivation different than those in [8–11]. This RLS variant is derived in this thesis.

In this section, we first give a summary of the forward pass equations in CNN. After that, we modify the mean squared error (MSE) cost function in (2.10) so that the obtained normal equations can be expressed in terms of the sample correlation matrices and the sample cross-correlation vectors for RLS updating. Next, we take the partial derivatives of the modified cost function w.r.t. each parameter vector in the convolutional and the fully connected layers of a CNN. Although the modified cost function is different than the cost function defined in Chapter 2, most of the calculation follows the same procedure. We then obtain the normal equations and re-write them in terms of the correlation matrices and cross-correlation vectors over the past and present data samples. Finally, we find the recursive updating equation for each parameter vector.

3.2.1 Summary of Forward Pass Equations

Here, for the sake of continuity, we summarize the forward pass equations of CNN presented in Chapter 2. For $t \in [1, T]$, $a \in [1, A]$, $b \in [1, B]$, $i \in [1, I^2]$, $l \in [1, L]$, at the training sample t, the convolution sum $net_{b,i}^{(l)}(t)$ before the activation function is

$$net_{b,i}^{(l)}(t) = \sum_{a=1}^{A} \boldsymbol{z}_{a,i}^{(l-1)^T}(t)\boldsymbol{k}_{a,b}^{(l)}(t), \tag{3.1}$$

and

$$c_{b,i}^{(l)}(t) = f(net_{b,i}^{(l)}(t)), \tag{3.2}$$

where $c_{b,i}^{(l)}(t)$ is the i^{th} pixel of the b^{th} feature map at the l^{th} convolutional layer; $\boldsymbol{z}_{a,i}^{(l-1)}(t)$ is the vectorized i^{th} input image block from the a^{th} pooled feature map at the $(l-1)^{\text{th}}$ layer; $\boldsymbol{k}_{a,b}^{(l)}(t)$ is the filter weight vector from the a^{th} pooled feature map at the $(l-1)^{\text{th}}$ pooling layer to the b^{th} feature map at the l^{th} convolutional layer; $f(.)$ is the activation function. These forward pass process details of CNN are shown in Figure 2.2.1.

The pooling operation follows the local convolution operations performed at the convolutional layer. We reshape the pixels $c_{b,i}^{(l)}(t)$ to $c_{b,(i_1,i_2)}^{(l)}(t)$ using $i = I(i_1 - 1) + i_2$. The reshaping is necessary since we are using 2-dimensional moving masks here. Let $\mathcal{S}$ denote the pooling mask of size $S_1 \times S_2$. For $i_1, i_2 \in [1, I]$, $j_1, j_2 \in [1, J]$, we can then use either max pooling given by

$$z_{b,(j_1,j_2)}^{(l)}(t) = \max_{i_1,i_2 \in \mathcal{S}} \{c_{b,(i_1,i_2)}^{(l)}(t)\}, \tag{3.3}$$

or the average pooling given by

$$z_{b,(j_1,j_2)}^{(l)}(t) = \frac{1}{|\mathcal{S}|} \sum\sum_{i_1,i_2 \in \mathcal{S}} c_{b,(i_1,i_2)}^{(l)}(t), \tag{3.4}$$

where $z_{b,(j_1,j_2)}^{(l)}(t)$ is the $(j_1, j_2)^{\text{th}}$ pixels in the b^{th} pooled feature map at l^{th} layer; $|\mathcal{S}|$ is the cardinality of $\mathcal{S}$. The pixel $s_{b,(j_1,j_2)}^{(l)}(t)$ can be reshaped into double indexing $z_{b,j}^{(l)}(t)$ by $j = J(j_1 - 1) + j_2$. Here, we reshape it back for the convenience of the concatenating operator for the fully connected layer.

Next, according to the concatenating operator $F(.)$ defined in Chapter 2, we have the input $\boldsymbol{y}(t)$ of the fully connected layer as

$$\boldsymbol{y}(t) = F(\{z_{m,n}^{(L)}(t)\}_{m=1,\cdots,M,\ n=1,\cdots,N^2}). \tag{3.5}$$

where $z_{m,n}^{(L)}(t)$ is the n^{th} pixels in the m^{th} pooled feature map at L^{th} layer.

The weighted sum and the output of the p^{th} node $net_p^{(L+1)}(t)$ and $o_p^{(L+1)}(t)$, $p \in [1, P]$, at the fully connected layer are, respectively

$$net_p^{(L+1)}(t) = \boldsymbol{y}^T(t)\boldsymbol{w}_p(t), \tag{3.6}$$

and

$$o_p^{(L+1)}(t) = f(net_p^{(L+1)}(t)). \tag{3.7}$$

We will modify the MSE cost function defined in (2.10) and find the error back-propagation based on the modified cost function in the next subsection.

3.2.2 Error Back-Propagation with the Modified Weighted Cost Function

To apply RLS algorithm to CNNs, we first modify the MSE cost function in (2.10) for iteration t as

$$\tilde{\mathcal{J}}(t) := \frac{1}{2t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{p=1}^{P}(\delta_p^{(L+1)}(\tau) - net_p^{(L+1)}(\tau))^2 = \frac{1}{2t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{p=1}^{P}\tilde{e}_p^{(L+1)^2}(\tau), \tag{3.8}$$

where $0 < \lambda \leq 1$ is called the *forgetting factor* [8], $\tilde{e}_p^{(L+1)}(\tau) := \delta_p^{(L+1)}(\tau) - net_p^{(L+1)}(\tau)$, and $\delta_p^{(L+1)}(\tau) := f^{-1}(d_p(\tau))$.

In other words, the modified weighted MSE cost function is defined before the activation function $f(.)$ between the desired value $\delta_p^{(L+1)}(\tau)$ and the actual weight sum $net_p^{(L+1)}(\tau)$ of each neuron p, $p \in [1, P]$. In practice, if $f^{-1}(d_p(\tau))$ goes to infinite when $d_p(\tau) = 1$, then $f^{-1}(d_p(\tau))$ can be specified as a large number. Also, the modified weighted MSE cost function can be written in terms of the sum squared error (SSE) cost function $\tilde{\mathcal{L}}(\tau)$ at each single training sample as

$$\tilde{\mathcal{J}}(t) = \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\tilde{\mathcal{L}}(\tau), \tag{3.9}$$

where

$$\tilde{\mathcal{L}}(\tau) := \frac{1}{2}\sum_{p=1}^{P}\tilde{e}_p^{(L+1)^2}(\tau). \tag{3.10}$$

Based on this modification and the derivation of $\frac{\partial L(t)}{\partial \boldsymbol{w}_p(t)}$ in (2.14), the partial derivative of the modified cost function $\tilde{\mathcal{J}}(t)$ w.r.t. the weight vector $\boldsymbol{w}_p(t)$ after iteration t is

$$\begin{aligned}\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{w}_p(t)} &= \frac{1}{t}\sum_{\tau=1}^{t} \lambda^{t-\tau} \frac{\partial \tilde{e}_p^{(L+1)}(\tau)}{\partial \boldsymbol{w}_p(t)} \tilde{e}_p^{(L+1)}(\tau) \\ &= -\frac{1}{t}\sum_{\tau=1}^{t} \lambda^{t-\tau} \boldsymbol{y}(\tau) \tilde{e}_p^{(L+1)}(\tau) \\ &= -\frac{1}{t}\sum_{\tau=1}^{t} \lambda^{t-\tau} \boldsymbol{y}(\tau)(\delta_p^{(L+1)}(\tau) - \boldsymbol{y}^T(\tau)\boldsymbol{w}_p(t)). \end{aligned} \tag{3.11}$$

Note that here we are using the most current estimate of the weight vector $w_p(t)$ even for all previous $\tau \in [1, t-1]$. The errors are then back-propagated to the input nodes of the fully connected layer through the weight vector $\boldsymbol{w}_p(t)$. Next, we compute $\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial \boldsymbol{y}(\tau)}$ as in Chapter 2, i.e.,

$$\begin{aligned}\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial \boldsymbol{y}(\tau)} &= \sum_{p=1}^{P} \frac{\partial \tilde{e}_p^{(L+1)}(\tau)}{\partial \boldsymbol{y}(\tau)} \tilde{e}_p^{(L+1)}(\tau) \\ &= \sum_{p=1}^{P} [\frac{\partial \tilde{e}_p^{(L+1)}(\tau)}{\partial net_p^{(L+1)}(\tau)} \frac{\partial net_p^{(L+1)}(\tau)}{\partial \boldsymbol{y}(\tau)}] \tilde{e}_p^{(L+1)}(\tau) \\ &= -\sum_{p=1}^{P} \boldsymbol{w}_p(t) \tilde{e}_p^{(L+1)}(\tau). \end{aligned} \tag{3.12}$$

The error back-propagated from the fully connected layer to the Lth pooling layer remains the same except they are based on the modified cost function $\tilde{\mathcal{L}}(\tau)$, i.e.,

$$\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial z_{m,n}^{(L)}(\tau)} = F_{m,n}^{-1}(\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial \boldsymbol{y}(\tau)}), m \in [1, m], n \in [1, N^2], \tag{3.13}$$

where $\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial \boldsymbol{y}(\tau)}$ can be obtained using (3.12).

Once the errors have been back-propagated to the Lth pooling layers, same error back-propagation can be applied to the previous layers for the feature maps and the pooled feature maps except using the modified cost function $\tilde{\mathcal{L}}(\tau)$. That is, for the lth convolutional layer, $l \in [1, L]$,

$$\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial c_{b,i}^{(l)}(\tau)} = upsample(\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial z_{b,j}^{(l)}(\tau)}), i \in [1, I^2], j \in [1, J^2], b \in [1, B], \tag{3.14}$$

where $\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial z_{b,j}^{(l)}(\tau)}$ can be derived layer-by-layer using the same process as in Chapter 2. Also, $\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}$, $a \in [1, A]$, can be derived using

$$\begin{aligned}\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} &= \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} \\ &= \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial c_{b,i}^{(l)}(\tau)}\frac{\partial c_{b,i}^{(l)}(\tau)}{\partial net_{b,i}^{(l)}(\tau)}\frac{\partial net_{b,i}^{(l)}(\tau)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} \\ &= \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial c_{b,i}^{(l)}(\tau)}f'(net_{b,i}^{(l)}(\tau))\boldsymbol{z}_{a,i}^{(l-1)}(\tau), \end{aligned} \tag{3.15}$$

where $\frac{\partial \tilde{\mathcal{L}}(\tau)}{\partial c_{b,i}^{(l)}(\tau)}$ can be found using (3.14). With the partial derivatives of the modified cost function w.r.t. the unknown parameters in CNN, we are now ready to derive the RLS updating rule for each weight vector, which is introduced next.

3.2.3 RLS Updating Rule for the Fully Connected Layer

This section demonstrates how to find the RLS updating rule for the fully connected layer. This is done by first setting the partial derivatives of the cost function w.r.t. weight vectors to zero and obtaining the normal equations. Then, we re-write these equations in terms of the correlation matrix and cross-correlation vector, which will match the forms of the sample correlation matrices and the sample cross-correlation vector in the general RLS learning rule [8–10].

We first set the partial derivative of the modified cost function w.r.t. the weight vector at the fully connected layer in (3.11) to zero, which gives

$$\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{w}_p(t)} = -\frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\boldsymbol{y}(\tau)(\delta_p^{(L+1)}(\tau) - \boldsymbol{y}^T(\tau)\boldsymbol{w}_p(t)) = 0. \tag{3.16}$$

Now, define the weighted sample correlation matrix and cross-correlation vector of the fully connected layer as

$$R^{(L+1)}(t) := \sum_{\tau=1}^{t}\lambda^{t-\tau}\boldsymbol{y}(\tau)\boldsymbol{y}^T(\tau), \tag{3.17}$$

and

$$\Delta_p^{(L+1)}(t) := \sum_{\tau=1}^{t} \lambda^{t-\tau} \boldsymbol{y}(\tau)\delta_p^{(L+1)}(\tau), \tag{3.18}$$

respectively. Thus, the normal equation of the fully connected layer becomes

$$R^{(L+1)}(t)\boldsymbol{w}_p(t) = \Delta_p^{(L+1)}(t). \tag{3.19}$$

From the definitions in (3.17) and (3.18), it is clear that we have the following recursive equations for $R^{(L+1)}(t)$ and $\Delta_p^{(L+1)}(t)$:

$$R^{(L+1)}(t) = \lambda R^{(L+1)}(t-1) + \boldsymbol{y}(t)\boldsymbol{y}^T(t), \tag{3.20}$$

and

$$\Delta_p^{(L+1)}(t) = \lambda\Delta_p^{(L+1)}(t-1) + \boldsymbol{y}(t)\delta_p^{(L+1)}(t), p \in [1, P]. \tag{3.21}$$

Assuming ${R^{(L+1)}}^{-1}(t)$ exists, the updating equation for $\boldsymbol{w}_p(t)$ becomes

$$\begin{aligned}
\boldsymbol{w}_p(t) &= {R^{(L+1)}}^{-1}(t)\Delta_p^{(L+1)}(t) \\
&= {R^{(L+1)}}^{-1}(t)[\lambda\Delta_p^{(L+1)}(t-1) + \delta_p^{(L+1)}(t)\boldsymbol{y}(t)] \\
&= {R^{(L+1)}}^{-1}(t)\{\lambda\frac{1}{\lambda}[R^{(L+1)}(t) - \boldsymbol{y}(t)\boldsymbol{y}^T(t)]\boldsymbol{w}_p(t-1) + \delta_p^{(L+1)}(t)\boldsymbol{y}(t)\} \\
&= \boldsymbol{w}_p(t-1) + P^{(L+1)}(t)\boldsymbol{y}(t)\tilde{e}_p^{(L+1)}(t),
\end{aligned} \tag{3.22}$$

where $\tilde{e}_p^{(L+1)}(t) := \delta_p^{(L+1)}(t) - net_p^{(L+1)}(t)$, and $P^{(L+1)}(t) := {R^{(L+1)}}^{-1}(t)$.

However, one thing left here is to come up with a recursive equation for the matrix ${R^{(L+1)}}^{-1}(t)$ or $P^{(L+1)}(t)$. As with the standard RLS method [8–10], we use the matrix inversion lemma [30], which yields

$$R^{(L+1)}(t)^{-1} = \frac{1}{\lambda}{R^{(L+1)}}^{-1}(t-1) - \frac{1}{\lambda}\frac{{R^{(L+1)}}^{-1}(t-1)\boldsymbol{y}(t)\boldsymbol{y}^T(t){R^{(L+1)}}^{-1}(t-1)}{\lambda + \boldsymbol{y}^T(t){R^{(L+1)}}^{-1}(t-1)^{-1}\boldsymbol{y}(t)}, \tag{3.23}$$

or

$$P^{(L+1)}(t) = \frac{1}{\lambda}[I - P^{(L+1)}(t-1)\frac{\boldsymbol{y}(t)\boldsymbol{y}^T(t)}{\lambda + \boldsymbol{y}^T(t)P^{(L+1)}(t-1)\boldsymbol{y}(t)}]P^{(L+1)}(t-1). \tag{3.24}$$

Define

$$G^{(L+1)}(t) := \frac{P^{(L+1)}(t-1)\boldsymbol{y}(t)}{\lambda + \boldsymbol{y}^T(t)P^{(L+1)}(t-1)\boldsymbol{y}(t)}. \tag{3.25}$$

Thus, we get

$$P^{(L+1)}(t) = \frac{1}{\lambda}[I - G^{(L+1)}(t)\boldsymbol{y}^T(t)]P^{(L+1)}(t-1), \tag{3.26}$$

which is the recursive equation for $P^{(L+1)}(t)$ with initial matrix $P^{(L+1)}(0) = I$.

3.2.4 RLS Updating Rule for the Convolutional Layers

Next, we derive the updating rule for the l^{th} convolutional layer, where $l \in [1, L]$. To achieve this, we need to find the sample correlation matrix and cross-correlation vector as well as the normal equation of the filter weights. Assuming that the errors have been back-propagated to the feature maps at the l^{th} convolutional layer, we define

$$e_{b,i}^{(l)}(\tau) := \frac{\partial\tilde{\mathcal{L}}(\tau)}{\partial c_{b,i}^{(l)}(\tau)} f'(net_{b,i}^{(l)}(\tau)), b \in [1, B], i \in [1, I^2], \tag{3.27}$$

which is the back-propagated error at i^{th} pixel of b^{th} feature map before the activation function. We also define

$$\delta_{b,i}^{(l)}(\tau) := net_{b,i}^{(l)}(\tau) + e_{b,i}^{(l)}(\tau), \tag{3.28}$$

which is the desired value before the activation function. Then, we can re-write (3.15) as, for $a \in [1, A], b \in [b, B]$,

$$\begin{aligned}\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} &= \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)(\delta_{b,i}^{(l)}(\tau) - net_{b,i}^{(l)}(\tau)) \\ &= \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)(\delta_{b,i}^{(l)}(\tau) - \sum_{a'=1}^{A}\boldsymbol{z}_{a',i}^{(l-1)^T}(\tau)\boldsymbol{k}_{a',b}^{(l)}(t)) \\ &= \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)(\delta_{b,i}^{(l)}(\tau) - \mu_{i,a,b}^{(l)}(\tau) - \boldsymbol{z}_{a,i}^{(l-1)^T}(\tau)\boldsymbol{k}_{a,b}^{(l)}(t)), \end{aligned} \tag{3.29}$$

where

$$\mu_{i,a,b}^{(l)}(\tau) := \sum_{\substack{a'=1 \\ a'\neq a}}^{A}\boldsymbol{z}_{a',i}^{(l-1)^T}(\tau)\boldsymbol{k}_{a',b}^{(l)}(t). \tag{3.30}$$

Then, setting $\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)}$ to zero, we have normal equation

$$\frac{\partial \tilde{\mathcal{J}}(t)}{\partial \boldsymbol{k}_{a,b}^{(l)}(t)} = \frac{1}{t}\sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)(\delta_{b,i}^{(l)}(\tau) - \mu_{i,a,b}^{(l)}(\tau) - \boldsymbol{z}_{a,i}^{(l-1)^T}(\tau)\boldsymbol{k}_{a,b}^{(l)}(t)) = 0. \tag{3.31}$$

Now, we define the weighted sample correlation matrix and cross-correlation vector at the l^{th} convolutional layer as

$$R_a^{(l)}(t) := \sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)\boldsymbol{z}_{a,i}^{(l-1)^T}(\tau), \tag{3.32}$$

and

$$\Delta_{a,b}^{(l)}(t) := \sum_{\tau=1}^{t}\lambda^{t-\tau}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)(\delta_{b,i}^{(l)}(\tau) - \mu_{i,a,b}^{(l)}(\tau)), \tag{3.33}$$

respectively. Thus, the normal equation of the l^{th} convolutional layer becomes

$$R_a^{(l)}(t)\boldsymbol{k}_{a,b}^{(l)}(t) = \Delta_{a,b}^{(l)}(t), a \in [1, A], b \in [b, B]. \tag{3.34}$$

In addition, the weighted sample correlation matrix and cross-correlation vector at the training sample t can be expressed as

$$R_a^{(l)}(t) = \lambda R_a^{(l)}(t-1) + \sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(t)\boldsymbol{z}_{a,i}^{(l-1)^T}(t), \tag{3.35}$$

and

$$\Delta_{a,b}^{(l)}(t) = \lambda \Delta_{a,b}^{(l)}(t-1) + \sum_{i=1}^{I^2} \boldsymbol{z}_{a,i}^{(l-1)}(t)(\delta_{b,i}^{(l)}(t) - \mu_{i,a,b}^{(l)}(t)). \tag{3.36}$$

Using a similar process as before and assuming $R_a^{(l)}(t)^{-1}$ exists, the updating equation for $\boldsymbol{k}_{a,b}^{(l)}(t)$ is

$$\begin{aligned}
\boldsymbol{k}_{a,b}^{(l)}(t) &= R_a^{(l)}(t)^{-1}\Delta_{a,b}^{(l)}(t) \\
&= R_a^{(l)}(t)^{-1}[\lambda \Delta_{a,b}^{(l)}(t-1) + \sum_{i=1}^{I^2}(\delta_{b,i}^{(l)}(t) - \mu_{i,a,b}^{(l)}(t))\boldsymbol{z}_{a,i}^{(l-1)}(t)] \\
&= R_a^{(l)}(t)^{-1}\{\lambda \frac{1}{\lambda}[R_a^{(l)}(t) - \sum_{i=1}^{I^2} \boldsymbol{z}_{a,i}^{(l-1)}(t)\boldsymbol{z}_{a,i}^{(l-1)^T}(t)]\boldsymbol{k}_{a,b}^{(l)}(t-1) \\
&\qquad\qquad\qquad\qquad + \sum_{i=1}^{I^2}[\delta_{b,i}^{(l)}(t) - \mu_{i,a,b}^{(l)}(t)]\boldsymbol{z}_{a,i}^{(l-1)}(t)\} \\
&= \boldsymbol{k}_{a,b}^{(l)}(t-1) + P_a^{(l)}(t)\{\sum_{i=1}^{I^2} \boldsymbol{z}_{a,i}^{(l-1)}(t)[\delta_{b,i}^{(l)}(t) - \mu_{i,a,b}^{(l)}(t) - \boldsymbol{z}_{a,i}^{(l-1)^T}(t)\boldsymbol{k}_{a,b}^{(l)}(t-1)]\},
\end{aligned} \tag{3.37}$$

where $P_a^{(l)}(t) := R_a^{(l)^{-1}}(t)$. Again, we seek a recursive equation for $P_a^{(l)}(t)$. To achieve this, we can re-write $R_a^{(l)}(t)$ for the l^{th} convolutional layer in (3.35) as

$$R_a^{(l)}(t) = \lambda R_a^{(l)}(t-1) + \sum_{i=1}^{I^2} Z_{a,i}^{(l-1)}(t), \tag{3.38}$$

where $Z_{a,i}^{(l-1)}(t) := \boldsymbol{z}_{a,i}^{(l-1)}(t)\boldsymbol{z}_{a,i}^{(l-1)^T}(t)$. Alternatively, we define the summation over the first i terms of $Z_{a,i}^{(l)}(t)$, $i \in [1, I^2]$, as

$$R_{a,i+1}^{(l)}(t) := \lambda R_a^{(l)}(t-1) + \sum_{i'=1}^{i} Z_{a,i'}^{(l-1)}(t). \tag{3.39}$$

It should be noted that $R_a^{(l)}(t) = R_{a,I^2+1}^{(l)}(t)$. Equation (3.39) can also be written in a recursive form as

$$R_{a,i+1}^{(l)}(t) = R_{a,i}^{(l)}(t) + Z_{a,i}^{(l-1)}(t), \tag{3.40}$$

with the initial $R_{a,1}^{(l)}(t) = \lambda R_a^{(l)}(t-1)$. Next, let us define $P_{a,i}^{(l)}(t) := {R_{a,i}^{(l)}}^{-1}(t)$. Thus, we have $P_a^{(l)}(t) = {R_a^{(l)}}^{-1}(t) = {R_{a,I^2+1}^{(l)}}^{-1}(t) = P_{a,I^2+1}^{(l)}(t)$, and $P_{a,1}^{(l)}(t) = {R_{a,1}^{(l)}}^{-1}(t) = \frac{1}{\lambda}{R_a^{(l)}}^{-1}(t-1) = \frac{1}{\lambda}P_a^{(l)}(t-1)$.

Again, as before we apply the matrix inversion lemma [30] to (3.40), which yields

$$P_{a,i+1}^{(l)}(t) = [I - G_{a,i}^{(l)}(t){\boldsymbol{z}_{a,i}^{(l-1)}}^T(t)]P_{a,i}^{(l)}(t), \tag{3.41}$$

where

$$G_{a,i}^{(l)}(t) := \frac{P_{a,i}^{(l)}(t)\boldsymbol{z}_{a,i}^{(l-1)}(t)}{1 + {\boldsymbol{z}_{a,i}^{(l-1)}}^T(t)P_{a,i}^{(l)}(t)\boldsymbol{z}_{a,i}^{(l-1)}(t)}. \tag{3.42}$$

Note we have two recursive equations here: one is over index i; and the other one is over time t. To find $P_a^{(l)}(t)$ at time t, we need to run the recursive equation (3.41) over index i for I^2 iterations. After all I^2 recursions, we obtain $P_a^{(l)}(t) = P_{a,I^2+1}^{(l)}(t)$. The forgetting factor λ enters into the recursive equation when we drive the recursive equation from time $t-1$ to time t, i.e., when we initialize $P_{a,1}^{(l)}(t) = \frac{1}{\lambda}P_a^{(l)}(t-1)$. At the beginning of the training, $P_{a,1}^{(l)}(0)$ can be initialized as the identity matrix. The key steps for training CNNs using the developed RLS method with single batch are summarized in Algorithm 1.

Moreover, in this thesis, we apply the matrix inversion lemma [30] to (3.43) recursively to solve for the inverse. This recursive calculation can also be derived using the inverse of the sum of matrices theorem presented in [31].

3.3 Mini-Batch Extension

To introduce a balance point between the stochastic gradient descent and the batch gradient descent, the mini-batch gradient descent was proposed in [15, 32, 33]. The mini-batch gradient descent presents a trade-off between the robust convergence and the computational efficiency by

Algorithm 1 Single Batch RLS training for CNN

```
Require: Training image and label pairs $(\boldsymbol{z}, d)$, activation function $f(.)$, forgetting factor $\lambda$.
 1: Initialize: weight vectors at fully connected layer $\boldsymbol{w}_p$, filter vectors at convolutional layer $\boldsymbol{k}_{a,b}^{(l)}$.
 2: while not the end of epoch or error goal is not reached do
 3:     Initialize $P^{(L+1)}$ and $P_a^{(l)}$ as identity matrices, $\forall l \in [1, L], a \in [1, A]$
 4:     while not the end of the training data do
 5:         for $p \in [1, P]$, $l \in [1, L]$, $a \in [1, A]$, $b \in [1, B]$, $i \in [1, I^2]$ do
 6:             $\boldsymbol{z}_{a,i}^{(l-1)}, net_{b,i}^{(l)}, \boldsymbol{y}, net_p^{(L+1)}, d_p \leftarrow feedforward(\boldsymbol{z}, d)$          ▷ Forward Pass
 7:             $\frac{\partial \tilde{\mathcal{L}}}{\partial c_{b,i}^{(l)}} \leftarrow backpropagate(\boldsymbol{z}, d)$          ▷ Back-Propagation
 8:         end for
 9:         $G^{(L+1)} \leftarrow \frac{P^{(L+1)}\boldsymbol{y}}{\lambda + \boldsymbol{y}^T P^{(L+1)}\boldsymbol{y}}$          ▷ Update $\boldsymbol{w}_p$
10:         $P^{(L+1)} \leftarrow \frac{1}{\lambda}[I - G^{(L+1)}\boldsymbol{y}^T]P^{(L+1)}$
11:         for $p \in [1, P]$ do
12:             $\delta_p^{(L+1)} \leftarrow f^{-1}(d_p)$
13:             $\tilde{e}_p^{(L+1)} \leftarrow \delta_p^{(L+1)} - net_p^{(L+1)}$
14:             $\boldsymbol{w}_p \leftarrow \boldsymbol{w}_p + P^{(L+1)}\boldsymbol{y}\tilde{e}_p^{(L+1)}$
15:         end for
16:         for $l \in [1, L]$, $a \in [1, A]$, $b \in [1, B]$, $i \in [1, I^2]$ do          ▷ Update $\boldsymbol{k}_{a,b}^{(l)}$
17:             $e_{b,i}^{(l)} \leftarrow \frac{\partial \tilde{\mathcal{L}}}{\partial c_{b,i}^{(l)}} f'(net_{b,i}^{(l)})$
18:             $\delta_{b,i}^{(l)} \leftarrow net_{b,i}^{(l)} + e_{b,i}^{(l)}$
19:             $\mu_{i,a,b}^{(l)} \leftarrow \sum_{\substack{a'=1 \\ a' \neq a}}^{A} \boldsymbol{z}_{a',i}^{(l-1)^T} \boldsymbol{k}_{a',b}^{(l)}$
20:             $P_{a,1}^{(l)} \leftarrow \frac{1}{\lambda} P_a^{(l)}$
21:             $G_{a,i}^{(l)} \leftarrow \frac{P_{a,i}^{(l)} \boldsymbol{z}_{a,i}^{(l-1)}}{1 + \boldsymbol{z}_{a,i}^{(l-1)^T} P_{a,i}^{(l)} \boldsymbol{z}_{a,i}^{(l-1)}}$
22:             $P_{a,i+1}^{(l)} \leftarrow [I - G_{a,i}^{(l)} \boldsymbol{z}_{a,i}^{(l-1)^T}] P_{a,i}^{(l)}$
23:             $P_a^{(l)} \leftarrow P_{a,I^2+1}^{(l)}$
24:             $\boldsymbol{k}_{a,b}^{(l)} \leftarrow \boldsymbol{k}_{a,b}^{(l)} + P_a^{(l)} \{\sum_{i=1}^{I^2} \boldsymbol{z}_{a,i}^{(l-1)} [\delta_{b,i}^{(l)} - \mu_{i,a,b}^{(l)} - \boldsymbol{z}_{a,i}^{(l-1)^T} \boldsymbol{k}_{a,b}^{(l)}]\}$
25:         end for
26:     end while
27: end while
```

updating the parameters after each mini-batch number of samples. It has been shown in [33] that a small batch size can improve the generalization ability and has a smaller computational memory. Inspired from this idea, a general mini-batch RLS training is developed in this section, and applied to weight updating in CNNs.

3.3.1 RLS Updating Rules for Mini-Batch CNN Training

Let D be the mini-batch size and T be the total number of training samples as before. Then, the unknown parameters are updated every D samples and therefore the total count of the updating is $\tilde{T} = \frac{T}{D}$ (here, we assume T is divisible by D). The weighted sample correlation matrix and the sample cross-correlation vector are also updated every D samples, i.e., total of $\tilde{T}$ update times. Let us define the weighted sample correlation matrix $\tilde{R}(\tilde{t})$ and the sample cross-correlation vector $\tilde{\Delta}(\tilde{t})$ at time $\tilde{t}$ over D mini-batch of samples as,

$$\tilde{R}(\tilde{t}) := \lambda \tilde{R}(\tilde{t}-1) + \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \boldsymbol{z}_i(\tau)\boldsymbol{z}_i^T(\tau), \tag{3.43}$$

and

$$\tilde{\Delta}(\tilde{t}) := \lambda \tilde{\Delta}(\tilde{t}-1) + \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \delta_i(\tau)\boldsymbol{z}_i(\tau), \tag{3.44}$$

recursively. Note that each sample has total of I^2 input vectors $\boldsymbol{z}_i(\tau)$, $i \in [1, I^2]$. Also, we drop the indices of layers and feature maps to derive the general updating equation first, as well as added tildes in the notation to distinguish these from the single batch case. In addition, let $\boldsymbol{k}(\tilde{t})$ and $\boldsymbol{k}(\tilde{t}-1)$ be the parameter vectors at mini-batch times $\tilde{t}$ and $\tilde{t}-1$, respectively. Thus, the normal equations for these are:

$$\tilde{\Delta}(\tilde{t}) = \tilde{R}(\tilde{t})\boldsymbol{k}(\tilde{t}), \tag{3.45}$$

and

$$\tilde{\Delta}(\tilde{t}-1) = \tilde{R}(\tilde{t}-1)\boldsymbol{k}(\tilde{t}-1). \tag{3.46}$$

As before, we re-write $\tilde{R}(\tilde{t}-1)$ in terms of $\tilde{R}(\tilde{t})$, i.e.,

$$\tilde{\Delta}(\tilde{t}-1) = \tilde{R}(\tilde{t}-1)\boldsymbol{k}(\tilde{t}-1) = \frac{1}{\lambda}[\tilde{R}(\tilde{t}) - \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \boldsymbol{z}_i(\tau)\boldsymbol{z}_i^T(\tau)]\boldsymbol{k}(\tilde{t}-1). \tag{3.47}$$

Then, assuming $\tilde{R}^{-1}(\tilde{t})$ exists, we get

$$\begin{aligned}
\boldsymbol{k}(\tilde{t}) &= \tilde{R}^{-1}(\tilde{t})\tilde{\Delta}(\tilde{t}) = \tilde{P}(\tilde{t})\tilde{\Delta}(\tilde{t}) \\
&= \tilde{R}^{-1}(\tilde{t})[\lambda\tilde{\Delta}(\tilde{t}-1) + \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \delta_i(\tau)\boldsymbol{z}_i(\tau)] \\
&= \tilde{R}^{-1}(\tilde{t})\{\lambda\frac{1}{\lambda}[\tilde{R}(\tilde{t}) - \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \boldsymbol{z}_i(\tau)\boldsymbol{z}_\tau^T(t)]\boldsymbol{k}(\tilde{t}-1) + \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \delta_i(\tau)\boldsymbol{z}_i(\tau)\} \\
&= \boldsymbol{k}(\tilde{t}-1) + \tilde{P}(\tilde{t})[\sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \delta_i(\tau)\boldsymbol{z}_i(\tau) - (\sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \boldsymbol{z}_i(\tau)\boldsymbol{z}_i^T(\tau))\boldsymbol{k}(\tilde{t}-1)] \\
&= \boldsymbol{k}(\tilde{t}-1) + \tilde{P}(\tilde{t})\{\sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \sum_{i=1}^{I^2} \boldsymbol{z}_i(t)[\delta_i(t) - \boldsymbol{z}_i^T(t)\boldsymbol{k}(\tilde{t}-1)]\}.
\end{aligned} \tag{3.48}$$

Thus, for each update, $\sum_{i=1}^{I^2} \boldsymbol{z}_i(t)\boldsymbol{z}_i^T(t)$ and $\sum_{i=1}^{I^2} \delta_i(t)\boldsymbol{z}_i(t)$ are summed over all D samples. However, $\tilde{P}(\tilde{t})$ still needs to be calculated every sample like before regardless of the mini-batch. Finally, the unknown parameter $\boldsymbol{k}(\tilde{t})$ is updated every D samples as well.

Now, using the above general RLS mini-batch updating equation, for $\tilde{t} \in [1, \tilde{T}]$, $\tilde{T} = \frac{T}{D}$, the updating equation with mini-batch D of the weight vectors at the fully connected layer in CNN is

$$\boldsymbol{w}_p(\tilde{t}) = \boldsymbol{w}_p(\tilde{t}-1) + \tilde{P}^{(L+1)}(\tilde{t})\{\sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}} \boldsymbol{y}(\tau)[\delta_p^{(L+1)}(t) - \boldsymbol{y}^T(\tau)\boldsymbol{w}_p(\tilde{t}-1)]\}. \tag{3.49}$$

For matrix $\tilde{P}^{(L+1)}(\tilde{t})$, we can apply the matrix inversion lemma [30] to (3.43) recursively again. That is, for $\tau \in [D\tilde{t}-D+1, D\tilde{t}]$,

$$\tilde{G}_\tau^{(L+1)}(\tilde{t}) = \frac{\tilde{P}_\tau^{(L+1)}(\tilde{t})\boldsymbol{y}(\tau)}{1 + \boldsymbol{y}^T(\tau)\tilde{P}_\tau^{(L+1)}(\tilde{t})\boldsymbol{y}(\tau)}, \tag{3.50}$$

and the recursive equation for matrix $\tilde{P}_{\tau+1}^{(L+1)}(\tilde{t})$ is

$$\tilde{P}_{\tau+1}^{(L+1)}(\tilde{t}) = [I - \tilde{G}_{\tau}^{(L+1)}(t)\boldsymbol{y}^T(\tau)]\tilde{P}_{\tau}^{(L+1)}(\tilde{t}). \tag{3.51}$$

Note that the recursive equation in (3.51) is over index τ instead of $\tilde{t}$. At the first recursion, $\tilde{P}_{D\tilde{t}-D+1}^{(L+1)}(\tilde{t})$ is initialized as $\tilde{P}_{D\tilde{t}-D+1}^{(L+1)}(\tilde{t}) = \frac{1}{\lambda}\tilde{P}^{(L+1)}(\tilde{t}-1)$. And at the last recursion, we obtain $\tilde{P}^{(L+1)}(\tilde{t}) = \tilde{P}_{D\tilde{t}+1}^{(L+1)}(\tilde{t})$. At the beginning of the training, $P^{(L+1)}(0)$ can be initialized as identity matrix.

For the filter weight vector $\boldsymbol{k}_{a,b}^{(l)}(\tilde{t})$ at the l^{th} convolutional layer, the $\tilde{t}^{\text{th}}$ update of $\boldsymbol{k}_{a,b}^{(l)}(\tilde{t})$ is

$$\boldsymbol{k}_{a,b}^{(l)}(\tilde{t}) = \boldsymbol{k}_{a,b}^{(l)}(\tilde{t}-1) + \tilde{P}_a^{(l)}(\tilde{t})\{\sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)[\delta_{b,i}^{(l)}(\tau) - \mu_{i,a,b}^{(l)}(\tau) - \boldsymbol{z}_{a,i}^{(l-1)^T}(\tau)\boldsymbol{k}_{a,b}^{(l)}(\tilde{t}-1)]\}, \tag{3.52}$$

where

$$\mu_{i,a,b}^{(l)}(\tau) := \sum_{\substack{a'=1\\a'\neq a}}^{A}\boldsymbol{z}_{a',i}^{(l-1)}(\tau)^T\boldsymbol{k}_{a',b}^{(l)}(\tilde{t})). \tag{3.53}$$

Similarly, we can use the matrix inversion lemma [30] recursively again for calculating $\tilde{P}_a^{(l)}(\tilde{t})$. We first write $\tilde{R}_a^{(l)}(\tilde{t})$ as

$$\begin{aligned}\tilde{R}_a^{(l)}(\tilde{t}) &= \lambda\tilde{R}_a^{(l)}(\tilde{t}-1) + \sum_{\tau=D\tilde{t}-D+1}^{D\tilde{t}}\sum_{i=1}^{I^2}\boldsymbol{z}_{a,i}^{(l-1)}(\tau)\boldsymbol{z}_{a,i}^{(l-1)^T}(\tau)\\ &= \lambda\tilde{R}_a^{(l)}(\tilde{t}-1) + \sum_{\tau'=1}^{DI^2}\boldsymbol{z}_a^{(l-1)}(\tau')\boldsymbol{z}_a^{(l-1)^T}(\tau').\end{aligned} \tag{3.54}$$

Then , for $\tau' \in [1, DI^2]$, the recursive updating equations for $\tilde{P}_{a,\tau'}^{(l)}(\tilde{t})$ are

$$\tilde{G}_{a,\tau'}^{(l)}(\tilde{t}) = \frac{\tilde{P}_{a,\tau'}^{(l)}(\tilde{t})\boldsymbol{z}_a^{(l-1)}(\tau')}{1 + \boldsymbol{z}_a^{(l-1)^T}(\tau')\tilde{P}_{a,\tau'}^{(l)}(\tilde{t})\boldsymbol{z}_a^{(l-1)}(\tau')}, \tag{3.55}$$

and

$$\tilde{P}^{(l)}_{a,\tau'+1}(\tilde{t}) = [I - \frac{1}{\lambda}\tilde{G}^{(l)}_{a,\tau'}(\tilde{t})\boldsymbol{z}_a^{(l-1)^T}(\tau')]\tilde{P}^{(l)}_{a,\tau'}(\tilde{t}). \tag{3.56}$$

When $\tau' = 1$, we initialize $\tilde{P}^{(l)}_{a,1}(\tilde{t}) = \frac{1}{\lambda}\tilde{P}^{(l)}_a(\tilde{t}-1)$. And at the last recursion, we obtain $\tilde{P}^{(l)}_a(\tilde{t}) = \tilde{P}^{(l)}_{a,DI^2+1}(\tilde{t})$. At the beginning of the training, $\tilde{P}^{(l+1)}_a(0)$ can be initialized as identity matrix. The main steps for training CNN using the developed RLS method with the mini-batch D are summarized in Algorithm 2. In (3.52), for each update, we take the summation over DI^2 terms and multiply it by $\tilde{P}^{(l)}_a(\tilde{t})$. Therefore, in Algorithm 2, we introduce variables $batch\_sum^{(L+1)}_p$ and $batch\_sum^{(l)}_{a,b}$, $p \in [1,P], l \in [1,L], a \in [1,A], b \in [1,B]$, to improve the computational efficiency.

3.4 Computational Complexity Analysis

In this section, the computational complexity of the proposed RLS-based training approach as well as the three training approaches reviewed in Chapter 2 is analyzed. Here, the computational complexity of the forward pass is ignored since all methods do the same forward propagation. In addition, the analysis is based the same CNN structure and single batch training. Recall that the dimension of the weight vector (including bias) $\boldsymbol{w}_p$ at the fully connected layer is H, the dimension of the filter weight vector (including bias) $\boldsymbol{k}^{(l)}_{a,b}$ at the l^{th} convolutional layer is K_l, and the size of the l^{th}-layer feature maps is $I \times I$. The computational complexity for each training approach is compared and summarized in Table 3.1.

Table 3.1: Summary of the Computational Complexity.

	Proposed Method	SGD	Adam	RMSprop
$\boldsymbol{w}_p$	$O(H^2)$	$O(H)$	$O(H)$	$O(H)$
$\boldsymbol{k}^{(l)}_{a,b}$	$O(I^2K_l^2)$	$O(K_l)$	$O(K_l)$	$O(K_l)$

Algorithm 2 Mini-Batch RLS training for CNN

```
Require: Total of $T$ training data pairs $(\boldsymbol{z}, d)$, activ. func. $f(.)$, forgetting factor $\lambda$, mini-batch $D$.
1: Initialize: weight vectors at fully connected layer $\boldsymbol{w}_p$, filter vectors at convolutional layer $\boldsymbol{k}_{a,b}^{(l)}$.
2: while not the end of epoch or error goal is not reached do
3:     Initialize $\tilde{P}^{(L+1)}$ and $\tilde{P}_a^{(l)}$ as identity matrices, $\forall l \in [1, L], a \in [1, A]$
4:     for batch $\tilde{t} \in [1, \tilde{T} = T/D]$ do
5:         for $p \in [1, P], l \in [1, L], a \in [1, A], b \in [1, B]$ do
6:             $\tilde{P}_{D\tilde{t}-D+1}^{(L+1)} \leftarrow \frac{1}{\lambda}\tilde{P}^{(L+1)}$
7:             $\tilde{P}_{a,1}^{(l)} \leftarrow \frac{1}{\lambda}\tilde{P}_a^{(l)}$
8:             $batch\_sum_p^{(L+1)} \leftarrow 0$
9:             $batch\_sum_{a,b}^{(l)} \leftarrow 0$
10:            for training sample $\boldsymbol{z}_\tau$, $\tau \in [D\tilde{t} - D + 1, D\tilde{t}]$, $i \in [1, I^2]$, $\tau' \in [1, DI^2]$, do
11:                $\boldsymbol{z}_{a,i}^{(l-1)}, net_{b,i}^{(l)}, \boldsymbol{z}_{a,\tau'}^{(l-1)}, \boldsymbol{y}_\tau, net_p^{(L+1)}, d_p \leftarrow feedforward(\boldsymbol{z}, d)$    ▷ Forward Pass
12:                $\frac{\partial \tilde{\mathcal{L}}}{\partial c_{b,i}^{(l)}} \leftarrow backpropagate(\boldsymbol{z}, d)$    ▷ Back-Propagation
13:                $\tilde{G}_\tau^{(L+1)} \leftarrow \frac{\tilde{P}_\tau^{(L+1)}\boldsymbol{y}_\tau}{1 + \boldsymbol{y}_\tau^T \tilde{P}_\tau^{(L+1)}\boldsymbol{y}_\tau}$    ▷ Prepare for updating $\boldsymbol{w}_p$
14:                $\tilde{P}_{\tau+1}^{(L+1)} \leftarrow [I - \tilde{G}_\tau^{(L+1)}\boldsymbol{y}_\tau^T]\tilde{P}_\tau^{(L+1)}$
15:                $\delta_p^{(L+1)} \leftarrow f^{-1}(d_p)$
16:                $batch\_sum_p^{(L+1)} \leftarrow batch\_sum_p^{(L+1)} + \boldsymbol{y}_\tau(\delta_p^{(L+1)} - \boldsymbol{y}_\tau^T \boldsymbol{w}_p)$
17:                $\tilde{G}_{a,\tau'}^{(l)} \leftarrow \frac{\tilde{P}_{a,\tau'}^{(l)}\boldsymbol{z}_{a,\tau'}^{(l-1)}}{1 + \boldsymbol{z}_{a,\tau'}^{(l-1)^T}\tilde{P}_{a,\tau'}^{(l)}\boldsymbol{z}_{a,\tau'}^{(l-1)}}$    ▷ Prepare for updating $\boldsymbol{k}_{a,b}^{(l)}$
18:                $\tilde{P}_{a,\tau'+1}^{(l)} \leftarrow [I - \tilde{G}_{a,\tau'}^{(l)}\boldsymbol{z}_{a,\tau'}^{(l-1)^T}]\tilde{P}_{a,\tau'}^{(l)}$
19:                $\delta_{b,i}^{(l)} \leftarrow net_{b,i}^{(l)} + \frac{\partial \tilde{\mathcal{L}}}{\partial c_{b,i}^{(l)}} f'(net_{b,i}^{(l)})$
20:                $\mu_{i,a,b}^{(l)} \leftarrow \sum_{\substack{a'=1 \\ a' \neq a}}^{A} \boldsymbol{z}_{a',i}^{(l-1)^T}\boldsymbol{k}_{a',b}^{(l)}$
21:                $batch\_sum_{a,b}^{(l)} \leftarrow batch\_sum_{a,b}^{(l)} + \boldsymbol{z}_{a,\tau'}^{(l-1)}(\delta_{b,i}^{(l)} - \mu_{i,a,b}^{(l)} - \boldsymbol{z}_{a,\tau'}^{(l-1)^T}\boldsymbol{k}_{a,b}^{(l)})$
22:            end for
23:            $\tilde{P}^{(L+1)} \leftarrow \tilde{P}_{D\tilde{t}+1}^{(L+1)}$
24:            $\tilde{P}_a^{(l)} \leftarrow \tilde{P}_{a,DI^2+1}^{(l)}$
25:            $\boldsymbol{w}_p \leftarrow \boldsymbol{w}_p + \tilde{P}^{(L+1)}batch\_sum_p^{(L+1)}$
26:            $\boldsymbol{k}_{a,b}^{(l)} \leftarrow \boldsymbol{k}_{a,b}^{(l)} + \tilde{P}_a^{(l)}batch\_sum_{a,b}^{(l)}$
27:        end for
28:    end for
29: end while
```

For the fully connected layer, updating $P^{(L+1)}(t)$ in (3.26) and the weight vectors in (3.22) requires the multiplications of $H \times H$ matrices and H-dimensional vectors. Thus, the computational complexity for calculating the weight vector $\boldsymbol{w}_p$ at the fully connected layer is $O(H^2)$. For the lth convolutional layer, calculating $P_a^{(l)}(t)$ in (3.41) and (3.42) requires the multiplications of matrices and vectors for I^2 times, while updating $\boldsymbol{k}_{a,b}^{(l)}(t)$ in (3.37) only requires multiplication of matrices and vectors once. Since the size of the matrices are $K_l \times K_l$ and the vectors are K_l-dimensional in these equations, the computational complexity for the lth convolutional layer is $O(I^2 K_l^2)$.

From the updating equations in Section 2.4, we can also obtain the computational complexities of SGD, RMSProp, and Adam. Since the updating equations of the SGD only involve calculating the gradient of each parameter, the computational complexities for $\boldsymbol{w}_p$ and $\boldsymbol{k}_{a,b}^{(l)}$ are $O(H)$ and $O(K_l)$, respectively. For the RMSProp and the Adam, the updating equations need to calculate not only the gradient of each parameters, but also the H-dimensional and the K_l-dimensional element-wise square of the gradients once for both. Thus, the computational complexities of $\boldsymbol{w}_p$ and $\boldsymbol{k}_{a,b}^{(l)}$ are $O(H)$ and $O(K_l)$, respectively.

In comparison, one can notice that the price to pay for the fast convergence property of the propose RLS-based method is the high computational complexity. The other three methods have low computational complexity but do not converge as fast as the RLS-based method. This analysis also gives suggestions on how to design the CNN structure when using the RLS-based approach. That is, for the fully connected layer, reducing the number of nodes can decrease the computational complexity. As for the convolutional layers, decreasing the size of filters K_l would result in lager size of feature maps, i.e., smaller K_l but lager I^2 in $O(I^2 K_l^2)$. Thus, one needs to consider the computational complexity of the convolutional layer and finds a balance point between the size of the filters K_l and the size of the $I \times I$ feature maps.

3.5 Conclusion

In this chapter, we introduced a new RLS-based training approach for the convolutional neural networks. The unknown parameters are updated recursively for every new training sample.

A mini-batch version of the algorithm was also developed. Although the proposed method has a much higher computational complexity than SGD, RMSProp and Adam, it converges much faster than these conventional methods. It must be mentioned that the updating equations for the convolutional layers present a new extension of the standard RLS algorithm for neural networks. Finally, although the RLS method has fast convergent property, the speed of the training CNNs using the proposed approach still needs to be determined experimentally by considering the number of needed epochs to converge and the computational cost of a single epoch. In the next chapter, we will implement the proposed on two datasets to determine the convergent speed and computational complexity numerically and compare them with those of the algorithms in Chapter 2.

Chapter 4

Implementations

4.1 Introduction

Since the error back-propagation and the gradient-based training methods were developed for the convolutional neural network (CNNs), they have achieved excellent performance in many applications, including but not limited to document recognition [34], digits recognition [2,3,5], facial recognition [35], weather prediction [36,37], and even for 1-dimensional acoustic data and speech recognition [38,39]. For example, the task of digits recognition is to determine the individual digits from the handwritten digit images, which is considered as an benchmark for comparing different classifiers and training methods. For digits recognition, a well-trained CNN reached over 98% accuracy using LeNet-5 on MNIST dataset in [5].

In this chapter, the proposed RLS-based training algorithm was implemented and evaluated on two datasets. One is MNIST dataset [5]; the other one is Fashion-MNIST dataset [40]. First, the two datasets were introduced briefly. Then, the structure of the CNNs for each dataset was described and the CNNs were trained using the proposed RLS-based method. The performance of the proposed method was evaluated in terms of the training convergent speed, validation, and overall testing accuracy on the two datasets. Comparison with other training approaches mentioned in Chapter 2, including stochastic gradient descent (SGD) with momentum [2,3,5,13], root mean square propagation (RMSProp) [15], and adaptive moment estimation (Adam) [18], was also provided. Finally, the results were analyzed and the conclusion was made for the proposed method.

Chapter organization is as follows. In Section 4.2, MNIST and MNIST-Fashion datasets are introduced briefly. Section 4.3 sets up the structure of CNN for training. Section 4.4 provides the implementation results, observations, and performance evaluation of the proposed RLS-based training approach for CNNs. Finally, in Section 4.5, conclusions of the proposed approach are made based on the implementations, and potential improvements for the proposed method are came up with.

4.2 Dataset Description and Preparations

4.2.1 MNIST

MNIST is a handwritten digit database [5] and commonly used for image classification problems. This database includes 70,000 samples of single digit images from 0 to 9, in which the pixel values are from 0 to 255. The digit images in this database are gray-scale and have already been centered and fit into the proper size of 28×28 pixels, where the 24×24 centered digit sub-images are 4 pixel zero-padded on each side of the sub-images. Some digit examples of these images from the MNIST dataset are shown in Figure 4.1.

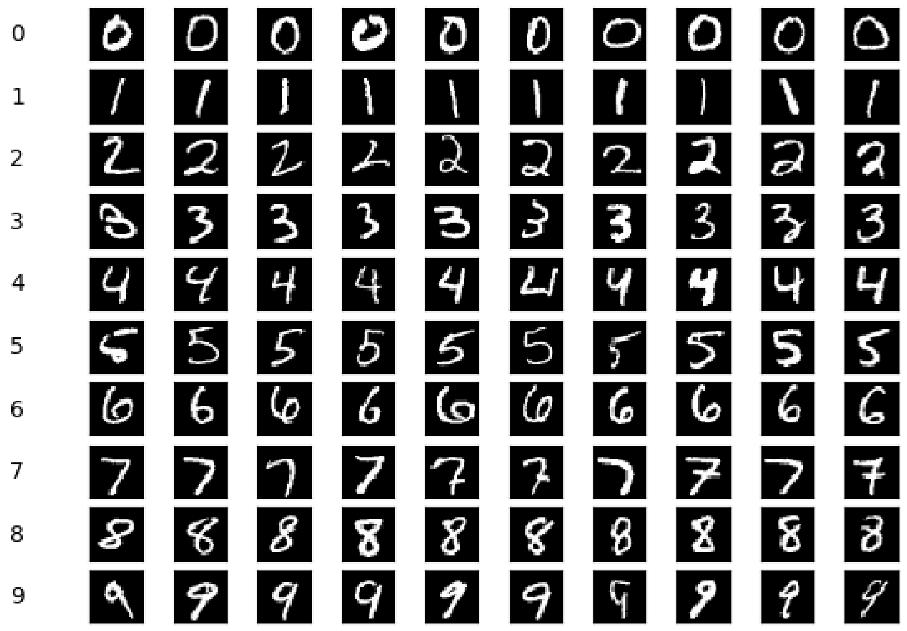


Figure 4.1: Some example images in the MNIST dataset. The labels of digits are at very left column.

In this simulation, a subset of the database consisting of 20,000 uniformly distributed samples (i.e., each class has same number of samples), was chosen for training CNNs. The validation and testing subsets contained 20,000 and 10,000 samples, respectively. The image pixel values of all subsets were normalized between 0 and 1 before being applied to CNNs.

4.2.2 Fashion-MNIST

Fashion-MNIST database [40] is another image database for classification problems, in which the objects are fashion items from 10 classes, including T-shirt, dress, sneaker and other fashion products. For better generalization ability, the items in Fashion-MNIST cover different groups: men, women, kids and unisex. The item pictures were first taken by professional photographers in a light-gray background. Particularly, white-color items are exclusive in the dataset due to the low contrast to the background. Then, the item pictures were trimmed, centered, gray-scaled and finally padded to generate the Fashion-MNIST database. This Fashion-MNIST dataset is considered to serve as a replacement for the original MNIST dataset for benchmarking machine learning algorithms because, as indicated in [40], MNIST has been overused and is not representative for modern image classification problems.

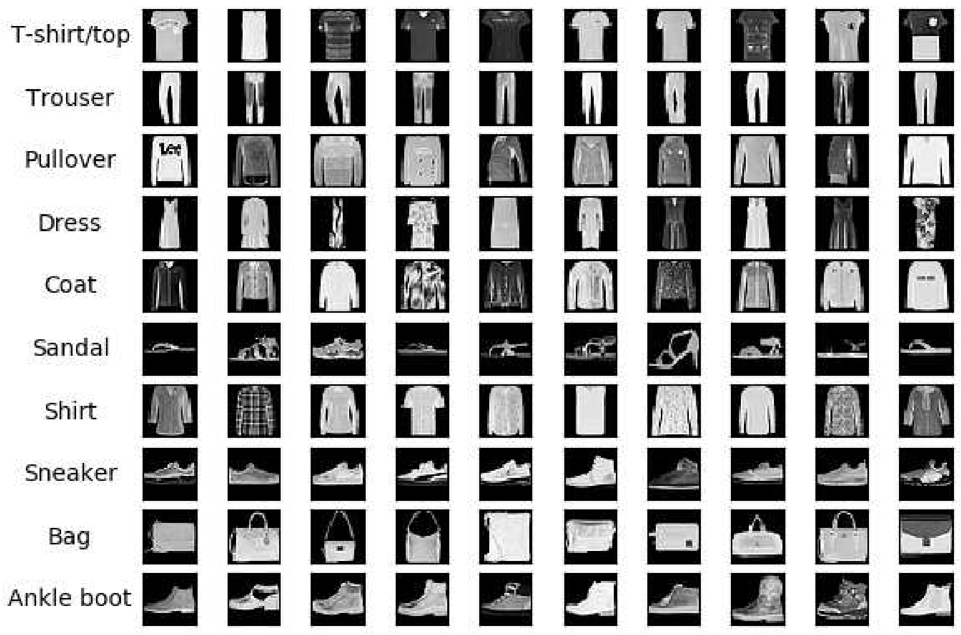


Figure 4.2: Some example images in the Fashion-MNIST dataset. The labels are at very left column.

Fashion-MNIST database consists of 70,000 samples. Each sample image is of size 28×28 and the pixel values are from 0 to 255. Some examples of these fashion items are shown in Figure 4.2. In this simulation, a subset of the database consisting of 20,000 uniformly distributed samples (i.e., each class has same number of samples), was chosen for training the CNNs. The validation and testing subsets contained 20,000 and 10,000 samples, respectively. All the samples were pre-processed by normalizing the pixel values between 0 and 1.

4.3 CNN Structure

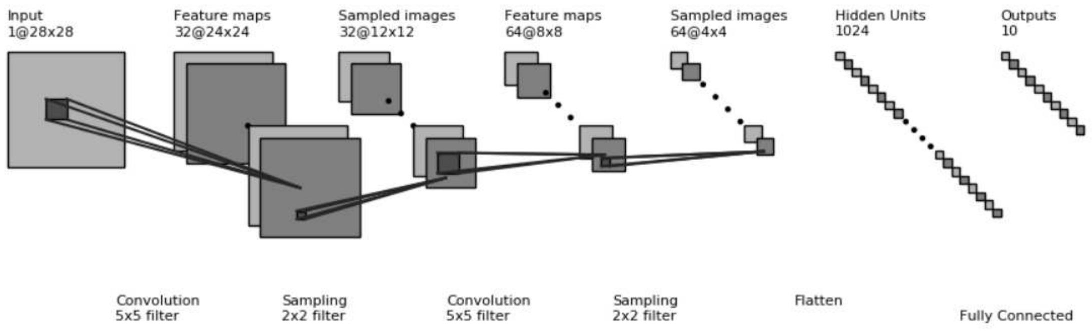


Figure 4.3: The CNN structure used on the MNIST and the Fashion-MNIST.

In this thesis, CNNs with the same structure was implemented for both MNIST and Fashion-MNIST datasets. Specifically, a CNN with two convolutional and two pooling layers, and one fully connected layer was implemented as shown in Figure 4.3. At the first convolutional layer, it contains 32 filters of size 5×5 and 32 feature maps of size 24×24, average pooled using a 2×2 mask and followed by 32 pooled feature maps of size 12×12 at the first pooling layer connected in pairs to each first-layer feature map. Then, the first-layer pooled feature maps become inputs and are applied to the second convolutional layer. There are 64 feature maps of size 8×8 at the second convolutional layer, hence in total of $32 \times 64 = 2,048$ filters of size 5×5, followed by 64 pooled feature maps of size 4×4 at the second pooling layer using an average pooling mask of size 2×2. Finally, the neurons at the second pooling layer were flattened into a long layer with

1024 units, followed by a fully connected output layer with 10 units. The activation function in each neuron was set to the sigmold function.

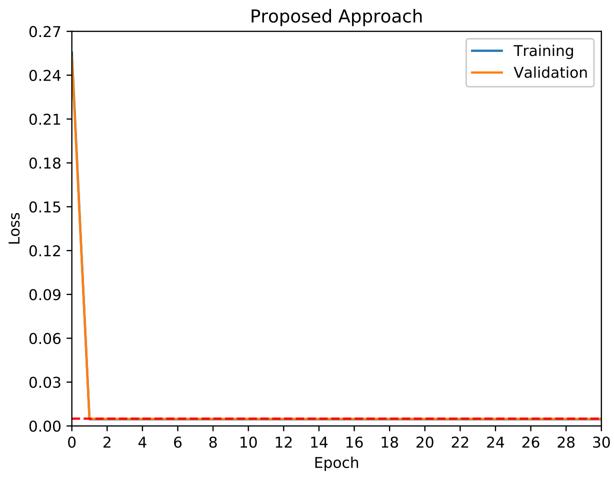
In this implementation, the unknown parameters in CNN were randomly initialized using a uniform distribution between -0.1 and 0.1. Matrices $P^{(1)}(0)$, $P_a^{(2)}(0)$, $a \in [1, 32]$, and $P^{(3)}(0)$ in (3.26) and (3.41) were initialized as identity matrices. The large number specified for the inverse of the sigmoid function in (3.8) was set to 5. The forgetting factor and the batch size both were set to 1.

To make the comparison fair all the training methods including SGD with the momentum term, RMSProp, Adam, and our proposed RLS algorithm used exactly the same CNN structure, training, validation and testing data subsets, average sum squared error cost function, weight initialization, and learning rate 0.002. For the SGD the momentum factor was set to 0.9. For Adam, the exponential decay rates for the first and second moment estimates β_1 and β_2 were 0.9 and 0.999, respectively. For RMSProp, the discounting factor β was 0.9. The presented learning curves and validation curves on the training data and accuracy on the testing data were obtained by averaging over ten simulation results.

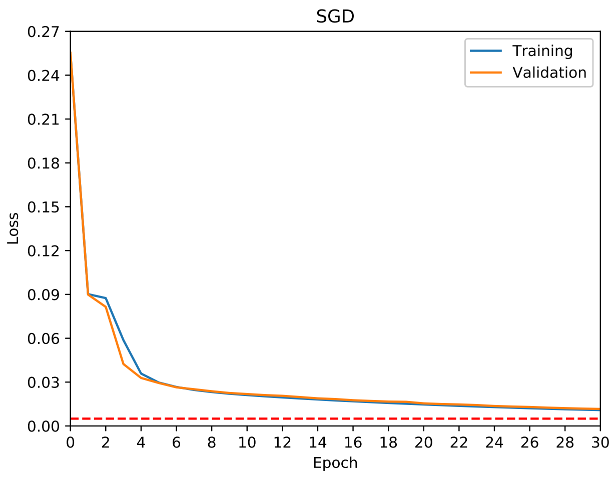
4.4 Results and Observations

4.4.1 MNIST

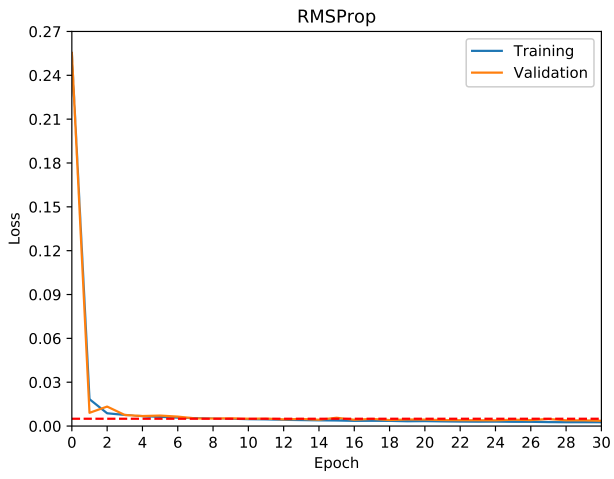
The constructed CNNs were trained and validated over 20,000 training samples and 20,000 validation samples on MNIST dataset for 30 epochs. At each epoch, the average MSEs over all output nodes on training and validation samples were computed. The learning curves were formed by plotting these average MSEs of training and validation samples at each epoch, which are shown in Figures 4.4 (a), (b), (c), and (d) for the proposed RLS method, SGD with momentum, RMSProp, and Adam, respectively. The convergent error goal was 0.005 as the red dashed lines show in Figure 4.4, and the convergent epoch was determined for each method once the training average MSE was below the error goal. The performance of each method was compared in terms of the convergent epochs on the training set and the average correct classification rates on the testing set, which are summarized in Table 4.1.



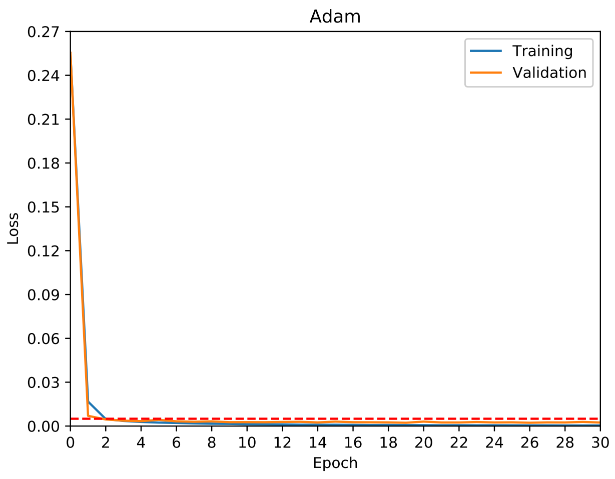
(a) Proposed RLS Approach.



(b) SGD with Momentum.



(c) RMSProp.



(d) Adam.

Figure 4.4: Learning curves from different approaches on MNIST. The proposed RLS-based approach converged after only one epoch.

As can be seen from Figure 4.4 and Table 4.1, all methods did not cause the overfitting issue according to the validation curves, while SGD did not reach the error goal within 30 epochs. As far as the testing accuracy's concerned, the proposed RLS-based method still reached a notable accuracy, which was approximately 1% lower than the best accuracy from Adam and only slightly

Table 4.1: Performance comparison of the proposed method and other methods on the MNIST dataset.

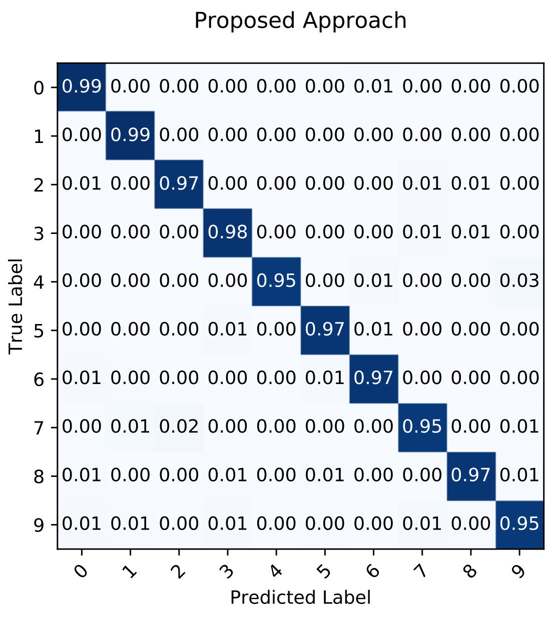
	Proposed Method	SGD with Momentum	RMSProp	Adam
Convergent Epoch	1	>30	9	2
Testing Accuracy	97.1	94.8	97.3	98.6

lower than the accuracy from RMSProp method. Additionally, although the proposed RLS-based training method did not achieve the classification higher than those of RMSProp or Adam, it converged after just one epoch, versus at least 2 epochs for other methods.

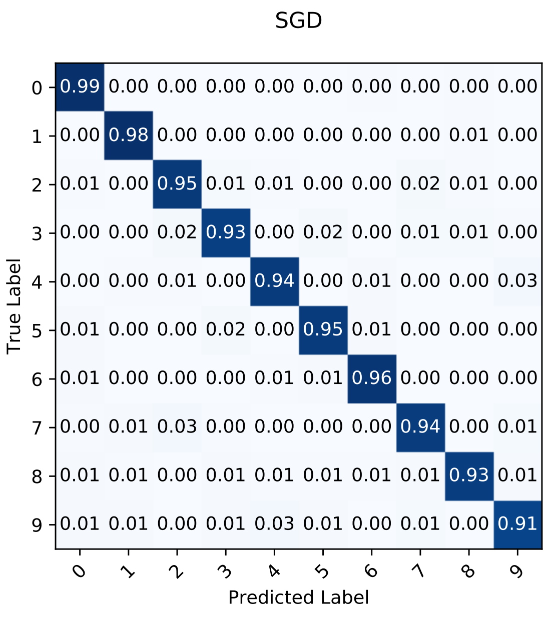
Finally, the average confusion matrix of each approach is displayed in Figures 4.5 (a), (b), (c), and (d), respectively. The diagonal elements indicate the correct accuracy of each class, and off-diagonal elements exhibit the mis-classification rate. All the classification rates were rounded to preserve 2 decimals. It can be found that the proposed RLS-based approach struggled in classifying digit 4 against 9 as the correct classification rate for digit 4 was 0.95 and mis-classification rate for digit 4 from digit 9 was 0.03.

In addition, for the proposed RLS-based method, let $P^{(2)}$ be the sum matrix of $P_a^{(2)}$, $a \in [1, 32]$, over all 32 matrices. Then, the Frobenius norms of $P^{(1)}$, $P^{(2)}$, and $P^{(3)}$ versus the number of iterations in the first epoch were plotted in Figures 4.6 (a), (b), and (c), respectively. These plots indicate that $P^{(1)}$ and $P^{(2)}$ at the first and the second convolutional layers converged within the first 1,000 iterations and became sparse after converging since their elements were close to zero. However, $P^{(3)}$ at the fully connected layer required more than 15,000 samples to converge.

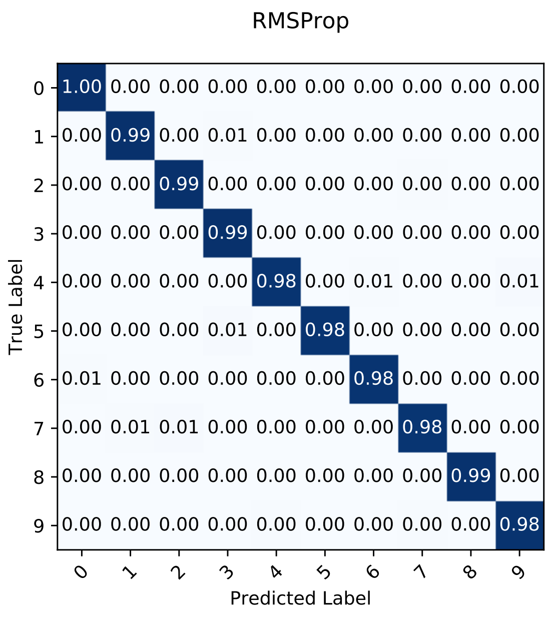
These plots suggested potential improvements, i.e., a two-phase training scheme for the parameters at the convolutional layer using the proposed RLS-based method, and fast matrix-vector multiplication for symmetric and sparse matrices [41–43]. Namely, in the first phase of an epoch, we train the filter vectors at the l^{th} convolutional layer, $l \in [1, L]$, until $\hat{t}^{(l)}$ iterations, $1 < \hat{t}^{(l)} < T$. After $\hat{t}^{(l)}$ iterations, we stop updating matrix $P_a^{(l)}$ since it has been convergent, and we only need to update the weight vectors. That is, let $\hat{P}_a^{(l)} = P_a^{(l)}(\hat{t}^{(l)})$, then for $t > \hat{t}$, $P_a^{(l)}(t) = \hat{P}^{(l)}$. For



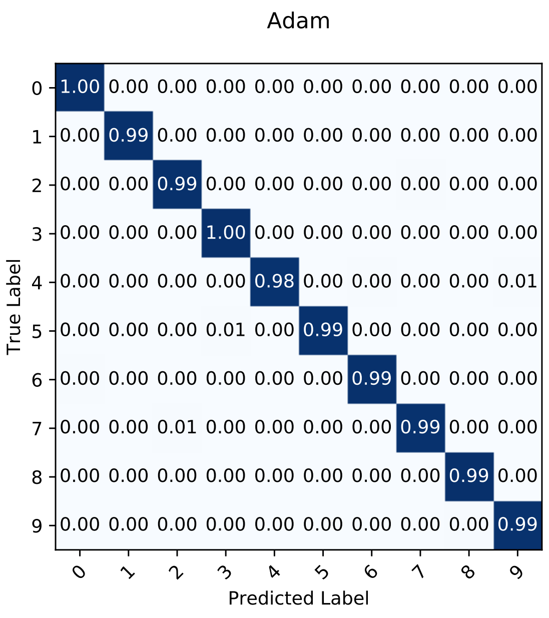
(a) Proposed RLS Approach.



(b) SGD with Momentum.



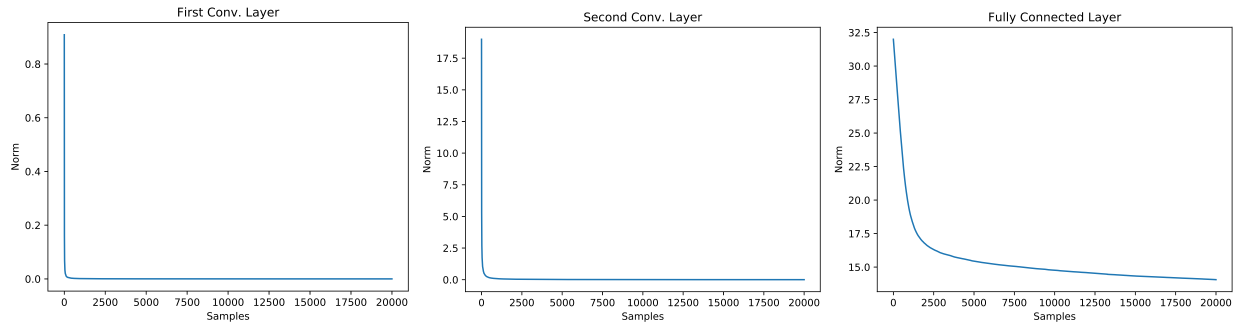
(c) RMSProp.



(d) Adam.

Figure 4.5: Confusion matrices from different approaches on the MNIST testing set. The classification rates are rounded to preserve 2 decimals. The proposed RLS-based approach struggled in classifying digit 4 against 9.

example, $\hat{t}^{(l)}$ can be set to 1,000 for both convolutional layers on the MNIST dataset. Meanwhile, $\hat{P}_a^{(l)}$ is a symmetric and sparse matrix. The standard matrix-vector multiplication can be replaced



(a) First Convolutional Layer. **(b)** Second Convolutional Layer **(c)** Fully Connected Layer

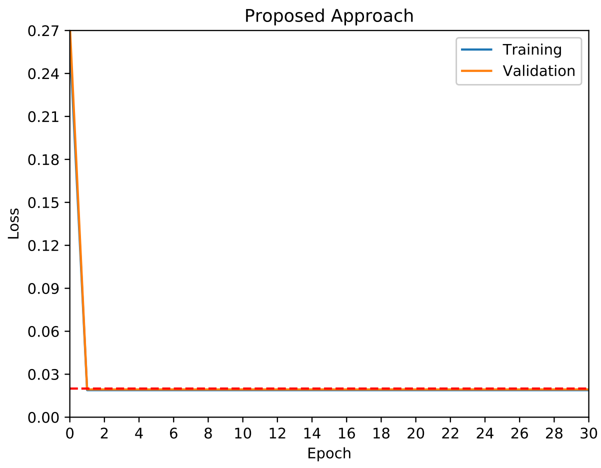
Figure 4.6: The Frobenius norms of $P^{(1)}$, $P^{(2)}$ and $P^{(3)}$ at the first and the second convolutional layers, and the fully convolutional layer, respectively.

with the fast matrix-vector multiplication for symmetric and sparse matrices presented in [41–43]. This two-phase training scheme with fast matrix-vector multiplication for symmetric and sparse matrices can reduce the computational cost of the RLS-based training approach since the heaviest computational costs are from recursively computing $P_a^{(l)}$ in (3.41). However, $\hat{t}^{(l)}$ requires to be determined by plotting the Frobenius norm of $P_a^{(l)}$ and assigned an proper number for which the norm is convergent.

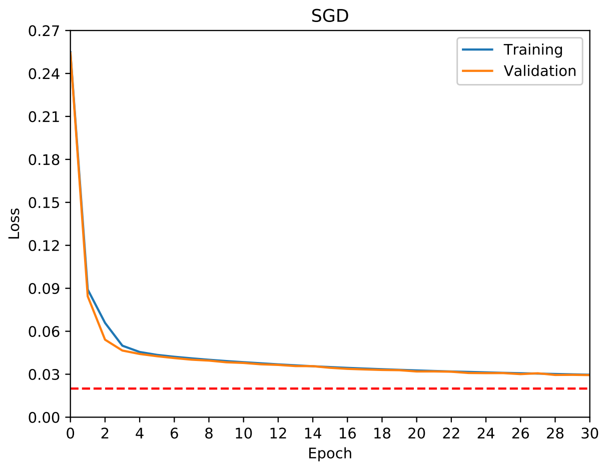
It should be noticed that as $t \to T$, matrices $P^{(1)}$, $P^{(2)}$ and $P^{(3)}$ decreased because $R^{(1)}$, $R_a^{(2)}$ and $R^{(3)}$ were added up as the new training sample came. Taking the inverse of them would result in $P^{(1)}$, $P_a^{(2)}$ and $P^{(3)}$ decreasing. For the next epoch, if $P^{(1)}$, $P_a^{(2)}$ and $P^{(3)}$ start with the values from the previous epoch, then the unknown parameters will not change too much. This was redeemed by re-setting $P^{(1)}$, $P_a^{(2)}$ and $P^{(3)}$ as identity matrices at the beginning of each epoch. Otherwise, the weight vectors will not be updated too much due to $P_a^{(l)}$ is close to zero.

4.4.2 Fashion-MNIST

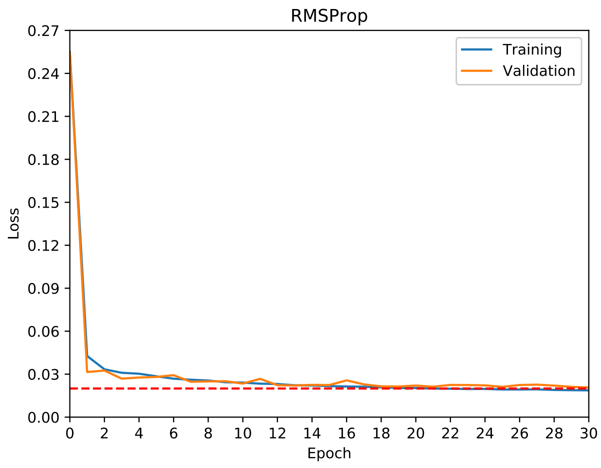
A CNN with the same structure for the MNIST dataset were trained on 20,000 Fashion-MNIST training samples and validated on 20,000 validation samples. The average MSEs of each method were calculated on training and validation samples at each epoch. The learning curves are shown in Figures 4.7 (a), (b), (c), and (d) for the proposed RLS method, SGD with momentum, RMSProp, and Adam, respectively. The training was run 30 epochs and the convergent error goal was set to



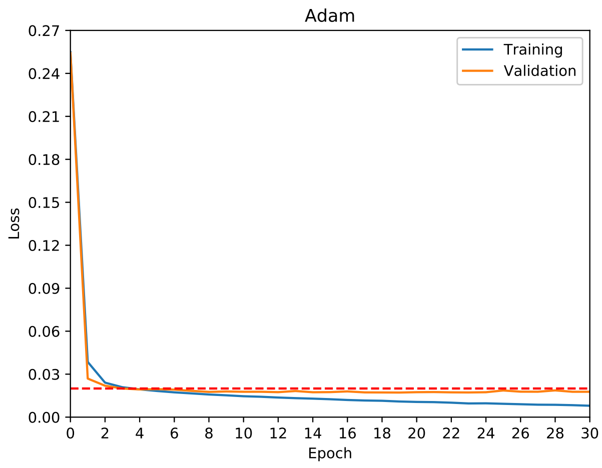
(a) Proposed RLS Approach.



(b) SGD with Momentum.



(c) RMSProp.



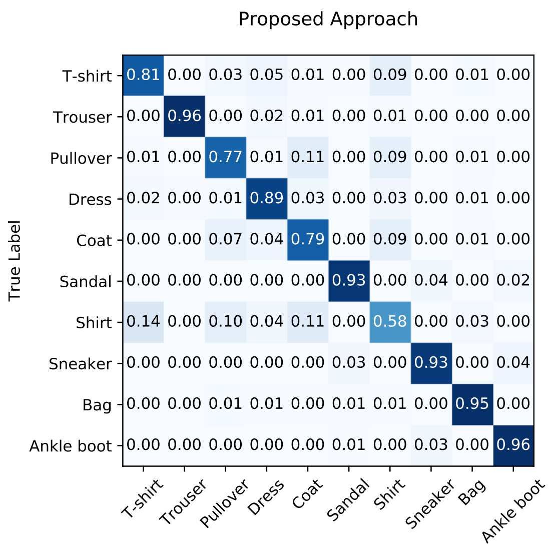
(d) Adam.

Figure 4.7: Learning curves from different approaches on Fashion-MNIST dataset.

0.02 as the red dashed lines show in Figure 4.4. The convergent epoch the training set and the correct classification rate on the testing set of each method were summarized in Table 4.2.

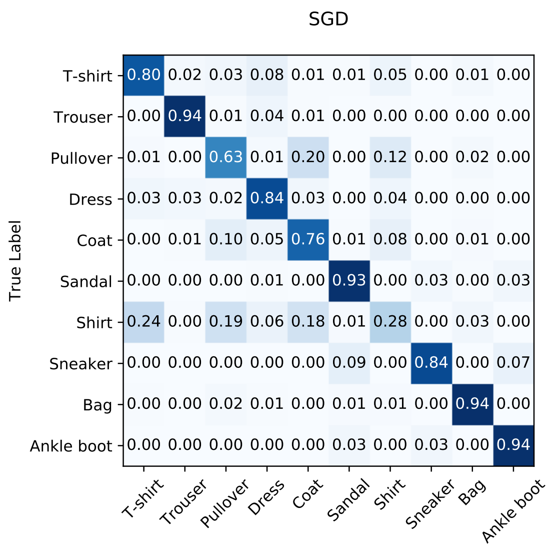
Table 4.2: Performance comparison of the proposed method and other methods on Fashion-MNIST dataset.

	Proposed Method	SGD with Momentum	RMSProp	Adam
Convergent Epoch	1	>30	21	4
Testing Accuracy	85.6	79.1	86.7	88.5



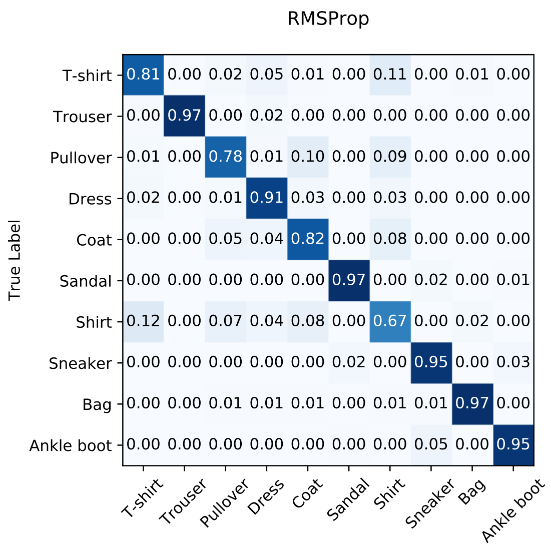
	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt	0.81	0.00	0.03	0.05	0.01	0.00	0.09	0.00	0.01	0.00
Trouser	0.00	0.96	0.00	0.02	0.01	0.00	0.01	0.00	0.00	0.00
Pullover	0.01	0.00	0.77	0.01	0.11	0.00	0.09	0.00	0.01	0.00
Dress	0.02	0.00	0.01	0.89	0.03	0.00	0.03	0.00	0.01	0.00
Coat	0.00	0.00	0.07	0.04	0.79	0.00	0.09	0.00	0.01	0.00
Sandal	0.00	0.00	0.00	0.00	0.00	0.93	0.00	0.04	0.00	0.02
Shirt	0.14	0.00	0.10	0.04	0.11	0.00	0.58	0.00	0.03	0.00
Sneaker	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.93	0.00	0.04
Bag	0.00	0.00	0.01	0.01	0.00	0.01	0.01	0.00	0.95	0.00
Ankle boot	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.03	0.00	0.96

(a) Proposed RLS Approach.



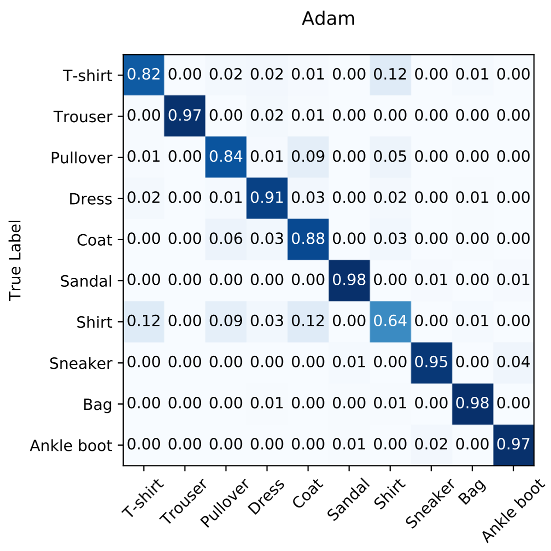
	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt	0.80	0.02	0.03	0.08	0.01	0.01	0.05	0.00	0.01	0.00
Trouser	0.00	0.94	0.01	0.04	0.01	0.00	0.00	0.00	0.00	0.00
Pullover	0.01	0.00	0.63	0.01	0.20	0.00	0.12	0.00	0.02	0.00
Dress	0.03	0.03	0.02	0.84	0.03	0.00	0.04	0.00	0.00	0.00
Coat	0.00	0.01	0.10	0.05	0.76	0.01	0.08	0.00	0.01	0.00
Sandal	0.00	0.00	0.00	0.01	0.00	0.93	0.00	0.03	0.00	0.03
Shirt	0.24	0.00	0.19	0.06	0.18	0.01	0.28	0.00	0.03	0.00
Sneaker	0.00	0.00	0.00	0.00	0.00	0.09	0.00	0.84	0.00	0.07
Bag	0.00	0.00	0.02	0.01	0.00	0.01	0.01	0.00	0.94	0.00
Ankle boot	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.03	0.00	0.94

(b) SGD with Momentum.



	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt	0.81	0.00	0.02	0.05	0.01	0.00	0.11	0.00	0.01	0.00
Trouser	0.00	0.97	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00
Pullover	0.01	0.00	0.78	0.01	0.10	0.00	0.09	0.00	0.00	0.00
Dress	0.02	0.00	0.01	0.91	0.03	0.00	0.03	0.00	0.00	0.00
Coat	0.00	0.00	0.05	0.04	0.82	0.00	0.08	0.00	0.00	0.00
Sandal	0.00	0.00	0.00	0.00	0.00	0.97	0.00	0.02	0.00	0.01
Shirt	0.12	0.00	0.07	0.04	0.08	0.00	0.67	0.00	0.02	0.00
Sneaker	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.95	0.00	0.03
Bag	0.00	0.00	0.01	0.01	0.01	0.00	0.01	0.01	0.97	0.00
Ankle boot	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.00	0.95

(c) RMSProp.



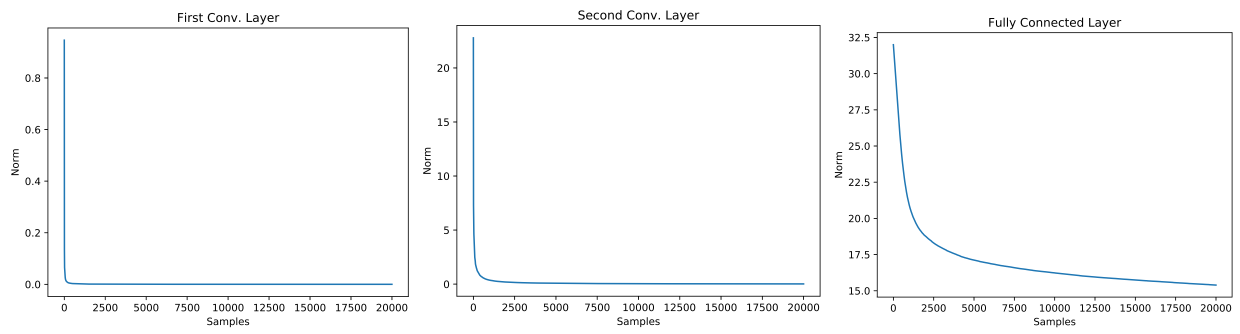
	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt	0.82	0.00	0.02	0.02	0.01	0.00	0.12	0.00	0.01	0.00
Trouser	0.00	0.97	0.00	0.02	0.01	0.00	0.00	0.00	0.00	0.00
Pullover	0.01	0.00	0.84	0.01	0.09	0.00	0.05	0.00	0.00	0.00
Dress	0.02	0.00	0.01	0.91	0.03	0.00	0.02	0.00	0.01	0.00
Coat	0.00	0.00	0.06	0.03	0.88	0.00	0.03	0.00	0.00	0.00
Sandal	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.01	0.00	0.01
Shirt	0.12	0.00	0.09	0.03	0.12	0.00	0.64	0.00	0.01	0.00
Sneaker	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.95	0.00	0.04
Bag	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.98	0.00
Ankle boot	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.02	0.00	0.97

(d) Adam.

Figure 4.8: Confusion matrices from different approaches on Fashion-MNIST testing set. The classification rates are rounded to preserve 2 decimals.

It can be observed that, the proposed RLS-based training method did not outperform RMSProp or Adam as far as the testing accuracy's concerned, but it retained the fast convergent property, which converged after one epoch. Finally, the average confusion matrix of each approach is presented in Figures 4.8 (a), (b), (c), and (d), respectively. The elements in them were normalized and 2-decimal rounded to indicate the classification rates. From Figure 4.8, one can conclude that the most difficult class in Fashion-MNIST is the class "shirt", the reason being that shirts are naturally similar to other clothes, e.g., classes "T-shirt", "pullover", and "coat".

Furthermore, for the proposed RLS-based method, we found $P^{(2)}$ by taking the sum of $P_a^{(2)}$, $a \in [1, 32]$, over all 32 matrices. The Frobenius norms of $P^{(1)}$, $P^{(2)}$, and $P^{(3)}$ versus the number of iterations in the first epoch were plotted in Figures 4.9 (a), (b), and (c), respectively. These plots indicate that $P^{(1)}$ and $P_a^{(2)}$ at the first and the second convolutional layers retained their fast convergent property by converging significantly fast within 1,000 samples and became sparse after converging, whereas $P^{(3)}$ needed more than 15,000 sample to converge. Again, a two-phase training scheme of the proposed method with fast matrix-vector multiplication for symmetric and sparse matrices on the Fashion-MNIST dataset can help improve the computational efficiency. Also, the issue of elements in $P^{(1)}$ and $P^{(2)}$ decreasing rapidly to zero was redeemed by re-setting them as identity matrices at each beginning of epoch.



(a) First Convolutional Layer. **(b)** Second Convolutional Layer **(c)** Fully Connected Layer.

Figure 4.9: The Frobenius norms of $P^{(1)}$, $P^{(2)}$ and $P^{(3)}$ at the first and the second convolutional layers, and the fully convolutional layer, respectively.

4.5 Conclusion

This chapter presents the implementation results of the proposed RLS-based approach as well as other benchmark approaches on the MNIST and the Fashion-MNIST datasets. First, the fast convergent property of the proposed RLS-based approach has been shown since it converged within a single epoch on both datasets. Then, the proposed approach outperformed SGD with momentum dominantly, and achieved comparable testing accuracy with these of Adam and RMSProp. However, the proposed method struggled with the computational complexity as Section 3.4 motioned. Although it only needed one epoch to converge, it took much more computational efforts going through for one epoch. Other methods, Adam for example, needed multiple epoch to converge but took less computation cost to go through one epoch. A trade-off needs to be considered here for practical implementation. In addition, matrix $P_a^{(l)}(t)$ in (3.41) at the convolutional layers converged significantly fast. An advanced two-phase training scheme can improve the training efficiency. In the first phase, we train the weight vectors at the convolutional layers recursively as usual. When $P_a^{(l)}(t)$ converges, we fix $P_a^{(l)}(t)$ and enter the second phase of training, where we only update the weight vectors based on the fixed $P_a^{(l)}(t)$. Moreover, for the proposed RLS-based method, all the matrices in the recursive equations for the fully connected layer and the convolutional layers are symmetric matrices. Once $P_a^{(l)}(t)$ converges and becomes sparse, a fast matrix-vector multiplication for symmetric and sparse matrices algorithm can be implemented to improve the efficiency [41–43]. This scheme prevents the computational waste in a sense that it saves massive computational resources to update $P_a^{(l)}(t)$ at the convolutional layers after it converges.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, a recursive least squares (RLS) training algorithm is proposed for convolutional neural network (CNNs) in the hope that the RLS algorithm can speed up the training of the unknown parameters in CNNs. This is done by first computing the inverse of the accumulating weighted correlation matrices with the newly coming training samples, where the inverse computation is performed by applying the matrix inversion lemma [30] recursively. Then, the unknown parameters of CNNs are updated in terms of the inverse of the weighted sample correlation matrix and the newly coming samples. One benefit of the proposed approach is, unlike other conventional approaches that rely on the gradient descent algorithm, e.g., stochastic gradient descent (SGD) [2, 3, 5, 13], root mean square propagation (RMSProp) [15], and adaptive moment estimation (Adam) [18], the RLS-based learning does not need any learning rate to be specified, which avoids an improper learning rate resulting in mis-adjustment around the minimum.

In summary, the implementation results reveal that the proposed RLS-based training approach outperformed SGD with momentum in terms of convergent speed and overall testing accuracy on the MNIST handwritten digit and the Fashion-MNIST datasets. Also, with achieving comparable testing accuracy with those of RMSProp and Adam, the proposed RLS-based training method converged after one epoch, which was faster than any other traditional gradient-based updating method. Finally, the confusion matrices of these four methods on the testing set of Fashion-MNIST indicate that, overall they classified fashion items successfully but struggled with distinguishing ”shirt” from ”T-shirt”, ”pullover”, and ”coat”.

Regardless of the effectiveness and the fast convergent property of the proposed RLS-based approach, some improvements can still be done to enhance the performance of the proposed approach for training CNNs, which will be introduced in the next section.

5.2 Future Work

Although the trained CNNs using RLS-based method have achieved good results, some extensions can still be taken into consideration to improve the performance. The potential extensions are

- A two-phase training scheme.
- Efficient matrix multiplication for symmetric and sparse matrix.
- Advance version of RLS.
- Dynamic node creation.
- Implementation on other types of neural networks, e.g., recurrent neural networks (RNNs).

Two-Phase Training Scheme: The implementation on the MNIST and the Fashion-MNIST datasets has shown that, for the proposed RLS-based training approach, matrix $P_a^{(l)}(t)$ in (3.41) converged significantly fast. Thus, a two-phase training scheme for the convolutional layers can improve the computational efficiency. Specifically, in the first phase, we train the weight vectors at the convolutional layers recursively as usual. After a certain number of iterations and when $P_a^{(l)}(t)$ converges, we fix $P_a^{(l)}(t)$ and enter the second phase of training, where we stop updating $P_a^{(l)}(t)$ and only update the weight vectors based on the fixed $P_a^{(l)}(t)$. Since most computational complexity for the proposed RLS-based approach is from recursively calculating $P_a^{(l)}$, this two-phase training scheme can reduce the computational cost of the proposed approach.

Efficient Matrix Multiplication: All the matrices in the recursive equations for the weight vectors are symmetric matrices. Also, the implementation on the MNIST and the Fashion-MNIST showed that matrices $P_a^{(l)}$'s became sparse once they converged. Thus, in the second phase of the proposed two-phase training scheme above, we can implement a fast matrix-vector multiplication for symmetric and sparse matrices algorithm presented in [41–43], instead of using the standard matrix multiplication. It can reduce the computational complexity so to speed up the training process.

Advance Version of RLS: This thesis uses the standard RLS algorithm. However, it can be replaced by some fast RLS algorithms to save the computational resource, e.g., fast transversal filter (FTF) algorithm [44]. The FTF algorithm can be regarded as the combination of four transversal filters for forward and backward prediction errors, gain-vector computation, and joint-process estimation. It has shown effectiveness in reducing computational cost and retaining fast convergent property. However, the FTF algorithm has instability issue when implemented in finite-precision arithmetic [8].

Dynamic node creation: A certain structure of CNN was implemented in this thesis, while the setup structure may not be the most optimal, although the change of network structure may not improve the performance necessarily. One strategy is to use dynamic node creation technique [45–47]. Namely, the CNNs will learn from the data and setup the optimal structure for themselves. The approaches in [45–47] update weight vectors and create nodes simultaneous, and provide an optimal or near optimal new topology in the sense that the mean squared error is minimized for new topology.

Other Neural Networks: Although the RLS-based training approach has been presented and shows its effectiveness in this thesis, it can also be applied to other types of neural networks, e.g., RNNs, in the hope that the RLS-based training approach will retain the fast convergent property. However, the derivation of the RLS algorithm starts from the normal equations of the unknown parameters in neural networks. Therefore, the exploitation of the RLS algorithm for neural network requires the a detailed forward and back-propagation derivations.

Bibliography

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, D. E. Rumelhart and J. L. Mcclelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[2] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[3] Y. L. Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson, *Handwritten Digit Recognition with a Back-Propagation Network*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, p. 396–404.

[4] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, 2009.

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[6] E. Kauderer-Abrams, "Quantifying translation-invariance in convolutional neural networks," *arXiv preprint arXiv:1801.01450*, 2017.

[7] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[8] S. S. Haykin, *Adaptive filter theory*. Pearson Education India, 2008.

[9] G. C. Goodwin and K. S. Sin, *Adaptive filtering prediction and control*. Courier Corporation, 2014.

[10] A. H. Sayed, *Adaptive filters*. John Wiley & Sons, 2011.

[11] M. R. Azimi-Sadjadi and R.-J. Liou, "Fast learning process of multilayer neural networks using recursive least squares," *IEEE Transactions on Signal Processing*, vol. 40, no. 2, pp. 446–450, 1992.

[12] Q. Lu, R. Yang, M. Zhong, and Y. Wang, "An improved fault diagnosis method of rotating machinery using sensitive features and RLS-BP neural network," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 4, pp. 1585–1593, 2020.

[13] L. Bottou, "Stochastic gradient learning in neural networks," *Proceedings of Neuro-Nımes*, vol. 91, no. 8, p. 12, 1991.

[14] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[15] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[16] T. Schaul, I. Antonoglou, and D. Silver, "Unit tests for stochastic optimization," *arXiv preprint arXiv:1312.6055*, 2013.

[17] J. Zhang, H. Lin, S. Das, S. Sra, and A. Jadbabaie, "Stochastic optimization with non-stationary noise," *arXiv preprint arXiv:2006.04429*, 2020.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[20] N. S. Keskar and R. Socher, "Improving generalization performance by switching from Adam to SGD," *arXiv preprint arXiv:1712.07628*, 2017.

[21] F. Zou, L. Shen, Z. Jie, W. Zhang, and W. Liu, “A sufficient condition for convergences of adam and rmsprop,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 127–11 135.

[22] S. J. Reddi, S. Kale, and S. Kumar, “On the convergence of adam and beyond,” *arXiv preprint arXiv:1904.09237*, 2019.

[23] M. J. Shensa *et al.*, “The discrete wavelet transform: wedding the a trous and mallat algorithms,” *IEEE Transactions on signal processing*, vol. 40, no. 10, pp. 2464–2482, 1992.

[24] P. Liu, H. Zhang, K. Zhang, L. Lin, and W. Zuo, “Multi-level wavelet-cnn for image restoration,” in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2018, pp. 773–782.

[25] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient BackProp*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. [Online]. Available: https://doi.org/10.1007/978-3-642-35289-8_3

[26] J. Shore and R. Johnson, “Properties of cross-entropy minimization,” *IEEE Transactions on Information Theory*, vol. 27, no. 4, pp. 472–482, 1981.

[27] J. Shore, “Minimum cross-entropy spectral analysis,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 2, pp. 230–237, 1981.

[28] P. Golik, P. Doetsch, and H. Ney, “Cross-entropy vs. squared error training: a theoretical and experimental comparison.” in *Interspeech*, vol. 13, 2013, pp. 1756–1760.

[29] D. M. Kline and V. L. Berardi, “Revisiting squared-error and cross-entropy functions for training neural network classifiers,” *Neural Computing & Applications*, vol. 14, no. 4, pp. 310–318, 2005.

[30] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[31] K. S. Miller, "On the inverse of the sum of matrices," *Mathematics Magazine*, vol. 54, no. 2, pp. 67–72, 1981. [Online]. Available: https://doi.org/10.1080/0025570X.1981.11976898

[32] S. Khirirat, H. R. Feyzmahdavian, and M. Johansson, "Mini-batch gradient descent: Faster convergence under data sparsity," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017, pp. 2880–2887.

[33] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[34] D. S. Maitra, U. Bhattacharya, and S. K. Parui, "Cnn based common approach to handwritten character recognition of multiple scripts," in *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 2015, pp. 1021–1025.

[35] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[36] Y. Liu, E. Racah, J. Correa, A. Khosrowshahi, D. Lavers, K. Kunkel, M. Wehner, W. Collins *et al.*, "Application of deep convolutional neural networks for detecting extreme weather in climate datasets," *arXiv preprint arXiv:1605.01156*, 2016.

[37] A. Chattopadhyay, P. Hassanzadeh, and S. Pasha, "Predicting clustered weather patterns: A test case for applications of convolutional neural networks to spatio-temporal climate data," *Scientific reports*, vol. 10, no. 1, pp. 1–13, 2020.

[38] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[39] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012, pp. 4277–4280.

[40] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.

[41] R. Geus and S. Röllin, “Towards a fast parallel sparse symmetric matrix–vector multiplication,” *Parallel Computing*, vol. 27, no. 7, pp. 883–896, 2001.

[42] C. De Sa, A. Cu, R. Puttagunta, C. Ré, and A. Rudra, “A two-pronged progress in structured dense matrix vector multiplication,” in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018, pp. 1060–1079.

[43] K. A. Y. M. de Lorimjer and L. Zhong, “Performance optimizations and bounds for sparse symmetric matrix–multiple vector multiply,” *Computer Science*, 2003.

[44] J. Cioffi and T. Kailath, “Fast, recursive-least-squares transversal filters for adaptive filtering,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 2, pp. 304–337, 1984.

[45] M. R. Azimi-Sadjadi, S. Sheedvash, and F. O. Trujillo, “Recursive dynamic node creation in multilayer neural networks,” *IEEE Transactions on Neural Networks*, vol. 4, no. 2, pp. 242–256, 1993.

[46] M. Azimi-Sadjadi, S. Sheedvash, and F. Trujillo, “A new approach for dynamic node creation in multilayer neural networks,” in *Proceedings 1991 IEEE International Joint Conference on Neural Networks*. IEEE, 1991, pp. 2631–2638.

[47] F. O. Trujillo, S. Sheedvash, and M. R. Azimi-Sadjadi, “Recursive dynamic node creation in multilayer neural networks,” Ph.D. dissertation, Colorado State University. Libraries, 1993.