Matlab Library for Convolutional Neural Networks
Son Lam Phung and Abdesselam Bouzerdoum

Technical Report

Address:  School of Electrical, Computer and Telecommunications Engineering, University of Wollongong, Northfields Av, Wollongong, NSW 2522, Australia.

Emails:  s.phung@ieee.org, a.bouzedoum@ieee.org.

Keywords: convolutional neural networks, Matlab implementation, image classification, training algorithms.

Date:  1 November 2009.
## Contents

1 Introduction

2 CNN network model
   2.1 Network architecture .................................................. 2
   2.2 Mathematical model ..................................................... 3
      2.2.1 Convolution layer ................................................... 3
      2.2.2 Sub-sampling layer ............................................... 4
      2.2.3 Output layer ....................................................... 5

3 Network training
   3.1 Derivation of CNN error gradient ...................................... 6
      3.1.1 Error sensitivity definition ...................................... 6
      3.1.2 Error sensitivity computation ................................... 7
      3.1.3 Error gradient computation ..................................... 7
   3.2 CNN training algorithms ................................................. 8

4 Library usage
   4.1 Main functions ............................................................ 10
      4.1.1 Function cnn_new .................................................. 10
      4.1.2 Function cnn_init ................................................ 10
      4.1.3 Function cnn_sim .................................................. 11
      4.1.4 Function cnn_train ............................................... 11
   4.2 Image classification application ...................................... 12
      4.2.1 Experiment data ................................................... 12
      4.2.2 Creating a CNN .................................................... 12
      4.2.3 Training the CNN ................................................ 14
      4.2.4 Testing the CNN .................................................. 15

5 Matlab implementation notes ................................................. 16

6 Conclusion ........................................................................ 17

7 Appendix ........................................................................... 18
1 Introduction

Convolutional neural network (CNN) was initially proposed by LeCun in the early 1980s. It was further revised by LeCun and his colleagues in the 1990s [1]. CNN is a neural network model that is based on three key architectural ideas: local receptive fields, weight sharing, and sub-sampling in the spatial domain. The network is designed mainly for the recognition of two-dimensional visual patterns. Convolutional neural network has many strengths: (i) feature extraction and classification are integrated into one structure and fully adaptive; (ii) the network extract 2-D image features at increasing dyadic scales; (iii) it is relatively invariant to geometric, local distortions in the image. CNN has been used for in several applications including hand-written digit recognition, face detection, and face recognition.

This report documents a Matlab library that can be used to create and train a convolutional neural network. The library originated from some Matlab code we wrote in 2006 for a paper on pyramidal neural network [2]. Since then, the library has gone through several revisions to optimize speed and improve code documentation. The library and this report are aimed at researchers who wish to experiment with convolutional neural networks.

The report is organized as follows. Section 2 describes architectural aspects of the convolutional neural networks. Section 3 presents algorithms for batch training of the network. Section 4 describes the main functions in this library and illustrate its usage with an example application. Section 5 discusses the Matlab implementation and some code optimization techniques. Finally, Section 6 gives some concluding remarks.

2 CNN network model

In this section, we first describe the architecture of CNN and then present a detailed mathematical model of the network.

2.1 Network architecture

Convolutional neural networks are designed to process two-dimensional (2-D) image. A CNN consists of three main types of layers: (i) convolution layers, (ii) sub-sampling layers, and (iii) an output layer. Network layers are arranged in a feed-forward structure: each convolution layer is followed by a sub-sampling layer, and the last convolution layer is followed by the output layer (see Fig. 1). The convolution and sub-sampling layers are considered as 2-D layers, whereas the output layer is considered as a 1-D layer. In CNN, each 2-D layer is made up from several planes. A plane consists of neurons that are arranged in a 2-D array. The output of a plane is called a feature map.

- In a convolutional layer, each plane is connected to one or more feature maps of the preceding layer. A connection is associated with a convolution mask, which is a 2-D matrix of adjustable entries called weights. Each plane first computes the convolution between its 2-D inputs and its convolution masks. The convolution outputs are summed together and then added with an adjustable scalar, known as a bias term. Finally, an activation function is applied on the result to obtain the plane’s output. The plane output is a 2-D matrix called a feature map; this name arises from the fact that the convolution output indicates the presence of a visual feature at a given pixel location. A convolution layer produces one or more feature maps. Each feature map is then connected to exactly one plane in the next sub-sampling layer.

- A sub-sampling layer has the same number of planes as the preceding convolution layer. A sub-sampling plane divides its 2-D input into non-overlapping blocks of size $2 \times 2$ pixels. For each block, the sum of four pixels is calculated; this sum is multiplied by an adjustable weight before being added to a bias term. The result is passed through an activation function to produce an output for the $2 \times 2$ block. Clearly, each sub-sampling plane reduces its input size by half, along each dimension. A feature map in a sub-sampling layer is connected to one or more planes in the next convolution layer.
Figure 1: Layers in a convolutional neural network.

- In the last convolution layer, each plane is connected to exactly one preceding feature map. This layer uses convolution masks that have the same size as the input feature maps. Therefore, each plane in the last convolution layer will produce one scalar output. The outputs from all planes in this layer are then connected to the output layer.

- The output layer, in general, can be constructed from sigmoidal neurons or radial-basis-function (RBF) neurons. Here, we will focus on using sigmoidal neurons for the output layer. The outputs of this layer are considered as the network outputs. In applications such as visual pattern classification, these outputs indicate the category of the input image.

2.2 Mathematical model

The notation used to describe the functional aspects of CNN is summarized in Table 1. The symbol $l$ denotes the index of a network layer. The layer index $l$ goes from 1 to $L$, where $L$ is the number of network layers. Here, we assume that $L = 2a + 2$ where $a$ is a positive integer. Let $N_l$ be the number of feature maps in layer $l$, and $f_i(.)$ be the activation function of layer $l$. Let $y_n^l$ be the $n$-th feature map or output of layer $l$.

2.2.1 Convolution layer

Consider a convolution layer $l$. In this structure, $l$ is an odd integer, $l = 1, 3, ..., 2a + 1$. Let $h_l \times w_l$ denote the size of convolution mask for layer $l$. For feature map $n$, let

- $w_{m,n}^l = \{w_{m,n}^l(i,j)\}$ be the convolution mask from feature map $m$ in layer $l - 1$ to feature map $n$ in layer $l$,
- $b_n^l$ be the bias term associated with feature map $n$,
- $V_n^l$ denote the list of all planes in layer $l - 1$ that are connected to feature map $n$. For example, $V_4^l = \{2, 3, 5\}$ means that feature maps 2, 3 and 5 of layer $l - 1$ are connected to feature map 4 of layer $l$. 

Table 1: Architectural notation for CNN

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input image size</td>
<td>( H_0 \times W_0 )</td>
</tr>
<tr>
<td>Input image pixel</td>
<td>( x(i,j) ) or ( y^l_l(i,j) )</td>
</tr>
<tr>
<td>Layer index</td>
<td>( l )</td>
</tr>
<tr>
<td>Number of layers</td>
<td>( L = 2a + 2 )</td>
</tr>
<tr>
<td>Convolution layers</td>
<td>( C^1, C^2, ..., C^{2a+1} )</td>
</tr>
<tr>
<td>Sub-sampling layers</td>
<td>( S^1, S^3, ..., S^{2a} )</td>
</tr>
<tr>
<td>Output layer</td>
<td>( F^{2a+2} )</td>
</tr>
<tr>
<td>Activation function of layer ( l )</td>
<td>( f^l )</td>
</tr>
<tr>
<td>Number of feature maps in layer ( l )</td>
<td>( N^l )</td>
</tr>
<tr>
<td>Size of convolution mask for layer ( C_l )</td>
<td>( r_l \times c_l )</td>
</tr>
<tr>
<td>Convolution mask from feature map ( m ) in layer ( S^{l-1} ) to feature map ( n ) in layer ( C^l )</td>
<td>( { w^l_{m,n}(i,j) } )</td>
</tr>
<tr>
<td>Bias for feature map ( n ) in convolution layer ( C^l )</td>
<td>( b^l_n )</td>
</tr>
<tr>
<td>Weight for feature map ( n ) in layer ( S^l )</td>
<td>( w^l_n )</td>
</tr>
<tr>
<td>Bias for feature map ( n ) in sub-sampling layer ( S^l )</td>
<td>( b^l_n )</td>
</tr>
<tr>
<td>Feature map ( n ) in layer ( l )</td>
<td>( y^l_l(i,j) )</td>
</tr>
<tr>
<td>Size of a feature map in layer ( l )</td>
<td>( H_l \times W_l )</td>
</tr>
</tbody>
</table>

Figure 2: Convolution layer in CNN.

Feature map \( n \) of convolution layer \( l \) is calculated as

\[
y^l_n = f_l(\sum_{m \in V^l_n} y^{l-1}_m \otimes w^l_{m,n} + b^l_n) \tag{1}
\]

In Equation (1), \( \otimes \) is the 2-D convolution operator. Suppose that the size of input feature maps \( y^{l-1}_m \) is \( H^{l-1} \times W^{l-1} \) pixels, and the size of convolution masks \( w^l_{m,n} \) is \( r^l \times c^l \) pixels. The size of output feature map \( y^l_n \) is

\[
(H^{l-1} - r^l + 1) \times (W^{l-1} - c^l + 1) \text{ pixels} \tag{2}
\]

2.2.2 Sub-sampling layer

Now, we consider a sub-sampling layer \( l \). In our structure, \( l \) is an even integer, \( l = 2, 4, ..., 2a \). For feature map \( n \), let \( w^l_n \) be the weight and \( b^l_n \) be the bias term. We divide feature map \( n \) of convolution
layer \( \{l-1\} \) into non-overlapping blocks of size \( 2 \times 2 \) pixels. Let \( z_{n}^{l-1} \) be a matrix obtained by summing the four pixels in each block. That is,

\[
z_{n}^{l-1}(i, j) = y_{n}^{l-1}(2i - 1, 2j - 1) + y_{n}^{l-1}(2i - 1, 2j) + y_{n}^{l-1}(2i, 2j - 1) + y_{n}^{l-1}(2i, 2j)
\]

(3)

Feature map \( n \) of sub-sampling layer \( l \) is now calculated as

\[
y_{n}^{l} = f_{l}(z_{n}^{l-1} \times w_{n}^{l} + b_{n}^{l})
\]

(4)

Clearly, a feature map \( y_{n}^{l} \) in sub-sampling layer \( l \) will have a size of \( H^{l} \times W^{l} \), where

\[
H^{l} = H^{l-1}/2 \quad \text{and} \quad W^{l} = W^{l-1}/2
\]

(5)

![Sub-sampling layer in CNN.](image)

2.2.3 Output layer

In this study, we consider output layer \( L \) that consists of sigmoidal neurons. Let \( N^{L} \) be the number of output sigmoidal neurons. Let \( w_{m,n}^{L} \) denote the weight from feature map \( m \) of the last convolutional layer, to neuron \( n \) of the output layer. Let \( b_{n}^{L} \) be the bias term associated with neuron \( n \) of layer \( L \). The output of sigmoidal neuron \( n \) is calculated as

\[
y_{n}^{L} = f^{L}(\sum_{m=1}^{N^{L-1}} y_{m}^{L-1} w_{m,n}^{L} + b_{n}^{L})
\]

(6)

The outputs of all sigmoidal neurons form the network outputs.

3 Network training

For CNNs to perform different visual recognition tasks, a training algorithm must be devised. In a given application, the objective of network training is to minimize an error function, which is defined in terms of the network actual outputs and the desired outputs.

The various definitions used in CNN training are summarized in Table 2. Suppose that the training set has \( K \) input images and \( K \) desired output vectors. Let \( x^{k} \) be the \( k \)-th image in the training set and \( d^{k} \) be the corresponding desired output vector. The error function is defined as

\[
E(w) = \frac{1}{K \times N^{L}} \sum_{k=1}^{K} \sum_{n=1}^{N^{L}} (y_{n}^{k} - d_{n}^{k})^2
\]

(7)

where \( y_{n}^{k} \) is the actual network output. This is a function of all network parameters (weights and biases).

\[
E(w) = \frac{1}{K \times N^{L}} \sum_{k=1}^{K} \sum_{n=1}^{N^{L}} (y_{n}^{k} - d_{n}^{k})^2
\]

(8)
There are two major approaches to CNN training. The first approach updates network parameters after each training sample is presented. This approach requires less memory and but it is less stable because each training sample can push the network parameters along a new direction. The second approach, known as batch training, updates network weights and biases after all training samples are presented. This approach requires large memory storage because it must accumulate the changes in network parameters. Since the first training approach can be considered as a special case of the second training approach, we will focus on batch training. Note that in batch training, an evaluation of network output for all training samples and an update of all network parameters are referred to collectively as a training epoch.

Table 2: Notation for CNN training algorithm

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training image index</td>
<td>$k$</td>
<td>$k = 1, 2, ..., K$</td>
</tr>
<tr>
<td>Training image $k$</td>
<td>$x^k$</td>
<td>$x^k = {x^k(i,j), i = 1, ..., H_0; j = 1, ..., W_0}$</td>
</tr>
<tr>
<td>Desired output sample $k$</td>
<td>$d^k$</td>
<td>$d^k = (d_{1}^k, d_{2}^k, ..., d_{N_0}^k)^T$</td>
</tr>
<tr>
<td>Network input or output of layer 0</td>
<td>$y_{l,k}^0(u,v)$</td>
<td>$(y_{l,k}^0(u,v), n = 1, ..., N_l; u = 1, ..., H_l; v = 1, ..., W_l)$</td>
</tr>
<tr>
<td>Weighted sum input to neuron $(u, v)$ in convolution layer $l$, feature map $n$</td>
<td>$s_{n}^{l,k}(u,v)$</td>
<td>$s_{n}^{l,k}(u,v)$</td>
</tr>
<tr>
<td>Output of neuron $(u, v)$ in convolution layer $l$, feature map $n$ for input image $k$</td>
<td>$y_{l,k}^n(u,v)$</td>
<td>$y_{l,k}^n(u,v) = f_l(s_{n}^{l,k}(u,v))$</td>
</tr>
<tr>
<td>Weighted sum input to neuron $(u, v)$ in in subsampling layer $l$, feature map $n$</td>
<td>$s_{n}^{l,k}(u,v)$</td>
<td>$s_{n}^{l,k}(u,v)$</td>
</tr>
<tr>
<td>Output of neuron $(u, v)$ in in subsampling layer $l$, feature map $n$</td>
<td>$y_{l,k}^n(u,v)$</td>
<td>$y_{l,k}^n(u,v) = f_l(s_{n}^{l,k}(u,v))$</td>
</tr>
<tr>
<td>The $n$th error for image $k$</td>
<td>$e_{n}^{k}$</td>
<td>$e_{n}^{k} = y_{n}^{k} - d_{n}^{k}$</td>
</tr>
<tr>
<td>Error function</td>
<td>$E(w)$</td>
<td>$E(w) = \frac{1}{K \times N_L} \sum_{k=1}^{K} \sum_{n=1}^{N_L} (y_{n}^{k} - d_{n}^{k})^2$</td>
</tr>
<tr>
<td>Error sensitivity of pixel $(u,v)$ in 2-D layer $l$</td>
<td>$\delta_{x,u,v}^{l,k}$</td>
<td>$\delta_{x,u,v}^{l,k} = \frac{\partial E}{\partial s_{x,u,v}^{l,k}}, l \leq L_p$</td>
</tr>
<tr>
<td>Error sensitivity of neuron $n$ in 1-D layer $l$</td>
<td>$\delta_{n}^{l,k}$</td>
<td>$\delta_{n}^{l,k} = \frac{\partial E}{\partial s_{n}^{l,k}}$, $l &gt; L_p$</td>
</tr>
</tbody>
</table>

3.1 Derivation of CNN error gradient

This section presents a method for calculating the gradient of the error function defined in (8). We discuss what is definition of error sensitivity, how to calculate error sensitivity, and how to compute the error gradient.

3.1.1 Error sensitivity definition

The gradient is computed through error sensitivities, which are defined as the partial derivatives of the error function with respect to (w.r.t.) the weighted sum input to a neuron.

For neuron $(i,j)$ in feature map $n$ of convolution layer $l$, its error sensitivity is defined as

$$\delta_{n}^{l,k}(i,j) = \frac{\partial E}{\partial s_{n}^{l,k}(i,j)}, l = 1,3, ..., 2a + 1$$

For neuron $(i,j)$ in feature map $n$ of sub-sampling layer $l$, its error sensitivity is given by

$$\delta_{n}^{l,k}(i,j) = \frac{\partial E}{\partial s_{n}^{l,k}(i,j)}, l = 2,4, ..., 2a$$
For neuron \( n \) in output layer \( L \), its error sensitivity is
\[
\delta_{n}^{L,k} = \frac{\partial E}{\partial s_{n}^{L,k}}
\]  

### 3.1.2 Error sensitivity computation

Using Table 2 and the chain rule of differentiation, we can express the error sensitivities as follows.

- **Output layer:** \( l = L \)
  \[
  \delta_{n}^{L,k} = \frac{2}{K \times N_{L}} e_{n}^{k} f_{L}'(s_{n}^{L,k}), \quad n = 1, 2, ..., N_{L}
  \]  
  
- **Last convolution layer:** \( l = L - 1 \)
  \[
  \delta_{n}^{l,k} = f_{l}'(s_{n}^{l,k}) \sum_{m=1}^{N_{l+1}} \delta_{n}^{l+1,k} w_{n,m}^{l+1}, \quad n = 1, 2, ..., N_{l}
  \]  
  Because feature maps in convolution layer \( L - 1 \) have a size of \( 1 \times 1 \) pixel, writing \( \delta_{n}^{L-1,k} \) or \( \delta_{n}^{L-1,k}(1,1) \) is equivalent.

- **Last sub-sampling layer:** \( l = L - 2 \)
  \[
  \delta_{n}^{l,k}(i,j) = f_{l}(s_{n}^{l,k}(i,j)) \times w_{n,n}^{l+1}(i,j) \times \delta_{n}^{l+1,k}(1,1)
  \]  
  where \( n = 1, 2, ..., N_{L-2}; \quad i = 1, 2, ..., H_{L-2}; \quad j = 1, 2, ..., W_{L-2}. \)

- **Other convolution layer:** \( l = 2a + 1 \)
  \[
  \delta_{n}^{l,k}(i,j) = f_{l}(s_{n}^{l,k}(i,j)) \times \delta_{n}^{l+1,k}(i',j') \times w_{n}^{l+1,k}
  \]  
  where \( i' = \lfloor i/2 \rfloor \) and \( j' = \lfloor j/2 \rfloor. \)

- **Other sub-sampling layer:** \( l = 2a \)
  \[
  \delta_{n}^{l,k}(i,j) = f_{l}(s_{n}^{l,k}(i,j)) \times \sum_{m \in U_{n}^{l}} \sum_{(i',j') \in R^{l}(i,j)} \delta_{n}^{l+1,k}(i',j') w_{n,m}^{l+1}(i - i', j - j')
  \]  

### 3.1.3 Error gradient computation

- **Output layer:** \( l = L \)-Weights \( w_{m,n}^{l} \):
  \[
  \frac{\partial E}{\partial w_{m,n}^{l}} = \sum_{k=1}^{K} \delta_{n}^{l,k} y_{m}^{l-1,k}
  \]  
  where \( m = 1, 2, ..., N_{l-1} \) and \( n = 1, 2, ..., N_{l}. \)

- **Biases \( b_{n}^{l} \):
  \[
  \frac{\partial E}{\partial b_{n}^{l}} = \sum_{k=1}^{K} \delta_{n}^{l,k}
  \]  
  where \( n = 1, 2, ..., N_{l}. \)
• Last convolution layer: \( l = L - 1 \)
- Weights \( w^l_{m,n}(i,j) \):
  \[
  \frac{\partial E}{\partial w^l_{m,n}(i,j)} = \sum_{k=1}^{K} \delta_{n}^{l,k}(i,j) \ y_{l-1,n}^{l-1,k}
  \]
  where \( n = 1, 2, ..., N_l \).
- Biases \( b^l_{n} \):
  \[
  \frac{\partial E}{\partial b^l_{n}} = \sum_{k=1}^{K} \sum_{(i,j)} \delta_{n}^{l,k}(i,j)
  \]
  where \( n = 1, 2, ..., N_l \).

• Sub-sampling layer: \( l = 2a \)
- Weights \( w^l_{n}(i,j) \):
  \[
  \frac{\partial E}{\partial w^l_{n}(i,j)} = \sum_{k=1}^{K} \sum_{(i', j')} \delta_{n}^{l,k}(i', j') \ y_{l-1,n}^{l-1,k}(i', j')
  \]
  where \( n = 1, 2, ..., N_l \).
- Biases \( b^l_{n} \):
  \[
  \frac{\partial E}{\partial b^l_{n}} = \sum_{k=1}^{K} \sum_{(i,j)} \delta_{n}^{l,k}(i,j)
  \]
  where \( n = 1, 2, ..., N_l \).

• Other convolution layer: \( l = 2a + 1 \)
- Weights \( w^l_{m,n}(i,j) \):
  \[
  \frac{\partial E}{\partial w^l_{m,n}(i,j)} = \sum_{k=1}^{K} \delta_{n}^{l,k}(i,j) \ y_{l-1,n}^{l-1,k}
  \]
  where \( n = 1, 2, ..., N_l \).
- Biases \( b^l_{n} \):
  \[
  \frac{\partial E}{\partial b^l_{n}} = \sum_{k=1}^{K} \sum_{(i,j)} \delta_{n}^{l,k}(i,j)
  \]
  where \( n = 1, 2, ..., N_l \).

3.2 CNN training algorithms

Once the error gradient \( \nabla E(t) \) is derived, numerous optimization algorithms for minimizing \( E \) can be applied to train the network. Here, we focus on five representative training algorithms:

• gradient descent (GD) [3],
• gradient descent with momentum and variable learning rate (GDMV) [4],
• resilient back-propagation (RPROP) [5],
• conjugate gradient (CG) [6]
Levenberg-Marquardt (LM) [7]

Three algorithms GD, GDMV and RPROP are first-order optimization methods. The conjugate gradient algorithm can be considered as an intermediate between first- and second-order methods, whereas the Levenberg-Marquardt algorithm is a trust-region method that uses the Gauss-Newton approximation of the Hessian matrix. Since details of these algorithms can be found in the given references, we only summarize here their main characteristics (see Table 3).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Descent (GD) [3]</td>
<td>Weights are updated along the negative gradient [\Delta w(t) = -\alpha \nabla E(t)] (\alpha) is scalar learning rate, (\alpha &gt; 0)</td>
</tr>
<tr>
<td>GD With Momentum and Variable Learning Rate (GDMV) [4]</td>
<td>Weight update is a linear combination of gradient and previous weight update [\Delta w(t) = \lambda \Delta w(t-1) - (1 - \lambda) \alpha(t) \nabla E(t)] (\lambda) is momentum parameter, (0 &lt; \lambda &lt; 1)</td>
</tr>
</tbody>
</table>
| Resilient Backpropagation (RPROP) [5] | Weight update depends only on the sign of gradient \[\Delta w_i(t) = -\text{sign}\{\frac{\partial E}{\partial w_i}(t)\} \times \Delta_i(t)\] \(\Delta_i(t)\) is adaptive step specific to weight \(w_i\), defined as \[
\alpha(t) = \begin{cases} 
\eta_{\text{inc}} \Delta_i(t-1), & \text{if } \frac{\partial E}{\partial w_i}(t) \frac{\partial E}{\partial w_i}(t-1) > 0 \\
\eta_{\text{dec}} \Delta_i(t-1), & \text{if } \frac{\partial E}{\partial w_i}(t) \frac{\partial E}{\partial w_i}(t-1) < 0 \\
\Delta_i(t-1), & \text{otherwise}
\end{cases}
\]
\(\eta_{\text{inc}} > 1, 0 < \eta_{\text{dec}} < 1\): scalar terms |
| Conjugate Gradient (CG) [6] | Weights are updated along directions mutually conjugated w.r.t. Hessian matrix \[\Delta w(t) = \alpha(t)s(t), \text{where search direction defined as} \]
\[s(t) = \begin{cases} 
-\nabla E(t) & \text{if } t \equiv 1 \mod P \\
-\nabla E(t) + \beta(t) s(t-1) & \text{otherwise}
\end{cases}
\]
learning step \(\alpha(t)\) is found through a line search [8]. \(\beta(t)\) is updated according to the following Polak-Ribiere formula \[
\beta(t) = \frac{\nabla E(t) - \nabla E(t-1)}{\|\nabla E(t-1)\|^2}
\]
| Levenberg-Marquardt (LM) [7] | 2nd-order Taylor expansion and Gauss-Newton approximation of Hessian matrix \[\Delta w(t) = -[J^T J + \mu I]^{-1} \nabla E\] \(J\) is the Jacobian matrix defined as \[J_{q+k,i} = \frac{\partial e_q^k}{\partial w_i}, q = 1, 2, ..., N_L; k = 1, 2, ..., K; i = 1, 2, ..., P\] Gradient \(\nabla E\) is computed through the Jacobian matrix \(J\). \(\mu\) is an adaptive parameter controlling the size of the trust region. |

Computation of the Jacobian matrix is similar to computation of the gradient \(\nabla E\). However, we need to modify the definitions of error sensitivities: they are now defined for each network error \(e_q^k\), where \(q = 1, 2, ..., N_L\), instead of the overall error function \(E\).
4 Library usage

In this section, we first present an overview of the main functions in the Matlab library. Then, we illustrate its usage through an example application, namely face versus non-face classification.

4.1 Main functions

Designing a CNN involves four main types of tasks: (i) creating a network; (ii) initializing the network weights and biases; (iii) computing the network output for a given input; (iv) training the network to produce the desired output for a given input. Correspondingly, four Matlab functions are created to perform these tasks: \texttt{cnn\_new}, \texttt{cnn\_init}, \texttt{cnn\_sim}, and \texttt{cnn\_train}.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{cnn_new}</td>
<td>Create a new network</td>
</tr>
<tr>
<td>\texttt{cnn_init}</td>
<td>Initialize a network</td>
</tr>
<tr>
<td>\texttt{cnn_cm}</td>
<td>Create a connection matrix for a network layer</td>
</tr>
<tr>
<td>\texttt{cnn_sim}</td>
<td>Compute network output</td>
</tr>
<tr>
<td>\texttt{cnn_train}</td>
<td>Train a network</td>
</tr>
<tr>
<td>\texttt{cnn_train_gd}</td>
<td>Train a CNN using gradient descent method</td>
</tr>
<tr>
<td>\texttt{cnn_train_rprop}</td>
<td>Train a CNN using resilient backpropagation method</td>
</tr>
<tr>
<td>\texttt{cnn_compute_gradient}</td>
<td>Compute gradient of the error function for CNN</td>
</tr>
</tbody>
</table>

4.1.1 Function \texttt{cnn\_new}

The syntax of this function is given as

\[
\text{net} = \text{cnn\_new}(\text{input\_size, c, rec\_size, tf\_fcn, train\_method})
\]

where

- \textit{input\_size:} size of input image to the network. For example, to process images with height of 36 pixels and width of 32 pixels, we set \text{input\_size} = [36 32].
- \textit{c:} a cell array of connection matrices. Cell \text{c}\{l\} stores the connection matrix for layer \text{l}. That is, \text{c}\{l\}(i,j) is set to \text{true} if there is a connection from feature map \text{i} of layer \text{l} − 1, to feature map \text{j} of layer \text{l}. A Matlab function \texttt{cnn\_cm} has been written to create standard connection matrices.
- \textit{rec\_size:} size of receptive fields. It is a two-column matrix where \text{rec\_size}(l,:) is the receptive size of layer \text{l}. Note that the receptive fields of the sub-sampling layer is always [2 2].
- \textit{tf\_fcn:} transfer functions of network layers. It is a cell array of strings. Possible Matlab transfer functions are ‘tansig’, ‘logsig’, ‘purelin’ and ‘ltanh’.
- \textit{train\_method:} training method for the network, as a string. Training methods that have been implemented are ‘gd’ and ‘rprop’.

4.1.2 Function \texttt{cnn\_init}

This function randomizes network weights and biases, according to a Gaussian distribution with mean 0 and standard deviation 1:

\[
\text{new\_net} = \text{cnn\_init}(\text{net})
\]

Note that when a network is created using \texttt{cnn\_new}, its weights and biases are automatically initialized, so calling \texttt{cnn\_init} is not needed.
4.1.3 Function cnn_sim

This function computes the network output \( y \) for given input \( x \).

\[
y = \text{cnn}_\text{sim}(\text{net}, x)
\]

Input \( x \) is a 3-D array with \( H \times W \times K \) elements, where \( K \) is the number of 2-D input samples. Output \( y \) is a matrix with \( K \) columns, each column is the output for one input sample.

4.1.4 Function cnn_train

This function trains the network to produce the desired output for a given input. Its syntax is given as

\[
[\text{new}\_\text{net}, \text{new}\_\text{tr}] = \text{cnn}\_\text{train}(\text{net}, x, d)
\]

where

- \( \text{net} \): existing network,
- \( x \): network input as 3-D array of size \( H \times W \times K \),
- \( d \): target output as matrix with \( K \) columns,
- \( \text{tr} \): existing training record (optional),
- \( \text{new}\_\text{net} \): trained network,
- \( \text{new}\_\text{tr} \): updated training record.

To support analysis of the training process, several performance indicators are recorded in a Matlab structure at defined training epochs. The indicators are stored in a Matlab structure \( \text{new}\_\text{tr} \), which includes the following fields

- \( \text{mse} \): mean-square-errors,
- \( \text{time} \): training time in seconds,
- \( \text{epoch} \): epoch count,
- \( \text{output}\_\text{eval} \): number of output evaluations,
- \( \text{gradient}\_\text{eval} \): number of gradient evaluations,
- \( \text{hessian}\_\text{eval} \): number of Hessian matrix evaluations,
- \( \text{jacobian}\_\text{eval} \): number of Jacobian matrix evaluations,

For example, to plot the mean-square-error versus training time, we can type

\[
\text{plot}(\text{new}\_\text{tr}\.\text{time}, \text{new}\_\text{tr}\.\text{mse});
\]

If an existing training record is passed to the function in \( \text{tr} \) parameter, \( \text{cnn}\_\text{train} \) will append new training information to it and return an updated record in \( \text{tr}\_\text{new} \).

To perform the actual training, \( \text{cnn}\_\text{train} \) will call another Matlab function; the name of this function is determined by the field \( \text{net}.\text{train}.\text{method} \). For example, if \( \text{net}.\text{train}.\text{method} \) is '"\( \text{gd} \)", training will be done by \( \text{cnn}\_\text{train}\_\text{gd} \):

\[
[\text{new}\_\text{net}, \text{new}\_\text{tr}] = \text{cnn}\_\text{train}\_\text{gd}(\text{net}, x, d)
\]

This scheme allows new training algorithms to be added easily.
4.2 Image classification application

In this section, we present an example application of convolutional neural network in image classification. We will use the Matlab library to design a CNN to differentiate between face and non-face image patterns. Face detection aims to determine the presence and the locations of human faces in a digital image. To tackle this vision problem, a common approach is to scan all rectangular regions of the digital image and decide if each region is a face or a nonface - a task that can be done using a CNN classifier. The complete code for this experiment can be found in section Appendix.

![Figure 4: Examples of training images.](image)

4.2.1 Experiment data

The data used in this experiment are taken from a face and skin detection database [9]. For training, we use 1000 face images and 1000 non-face images (see Fig. 4); each image has $32 \times 32$ pixels. To load the training data, type

```matlab
>> load('data\train_data.mat')
```

The data are

```matlab
>> whos
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1x2000</td>
<td>16000</td>
<td>double</td>
</tr>
<tr>
<td>x</td>
<td>32x32x2000</td>
<td>16384000</td>
<td>double</td>
</tr>
</tbody>
</table>

That is, $x$ is a 3-D array of 2000 input samples, each sample has $32 \times 32$ pixels. Variable $d$ is a row vector of 2000 entries where

- $d(k) = 1$ if sample $k$ is a face pattern,
- $d(k) = -1$ if sample $k$ is a non-face pattern.

4.2.2 Creating a CNN

We will create a CNN as shown in Fig. 5. This network accepts an input image of size $32 \times 32$ pixels. It has a total of six layers.
• Convolution layer C1 uses receptive field of size 5 × 5. It produces two feature maps, each consisting of 28 × 28 pixels.

• Sub-sampling layer S2 uses receptive field of size 2 × 2; this applies to all sub-sampling layers. This layer produces feature maps of size 14 × 14 pixels. Note that feature maps in a sub-sampling layer always has one-to-one connections to feature maps of the previous convolution layer.

• Convolution layer C3 uses receptive field of size 3 × 3. It produces five feature maps, each consisting of 12 × 12 pixels.

• Sub-sampling layer S4 uses a receptive field of size 2 × 2 and produces 5 feature maps of size 6 × 6 pixels.

• Convolution layer C5 uses receptive field of size 6 × 6; this size is the same as the size of the feature map produced by the last sub-sampling layer. As a result, convolution layer C5 produces 5 scalar features.

• Output layer F6 has only one neuron that is fully connected to the output of C5.

We enter the receptive sizes for network layers as

```matlab
>> rec_size = [5 5; 2 2; 3 3; 2 2; 6 6; 0 0]
```

![Diagram of convolutional neural network](image)

Figure 5: An example of convolutional neural network.

There are three types of connections between CNN layers: (i) one-to-one connection; (ii) full connection; and (iii) custom connection. By specifying the connections, we also state the number of feature maps in each layer.

• The connection from a convolution layer to the next sub-sampling layer is always one-to-one connection. That is, a feature map of a convolution layer is connected to exactly one feature map of the next sub-sampling layer. We see this in C1 to S2, and C3 to S4. A one-to-one connection from C1 to S2 can be created in Matlab as

```matlab
>> cm2 = cnn_cm('1-to-1', 2)
```
where 2 is the number of feature maps in C1.

- The connection from a sub-sampling layer to the next convolution layer is a custom connection. For example, consider the connection from S2 to C3 in Fig. 5. Feature map 1 of layer S2 is connected to feature maps 1, 2 and 5 of layer C3. Feature map 2 of layer S2 is connected to feature maps 3, 4 and 5 of layer C3. This can be described by a connection matrix cm3 as follows

\[
\begin{bmatrix}
1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1
\end{bmatrix}
\]

The entry \(cm3(i,j)\) at row \(i\) and column \(j\) indicates if there is a connection from feature map \(i\) of layer S2 to feature map \(j\) of layer C3.

- The connection from the last convolution layer to the output layer is always full connection. That is, each feature produced by the last convolution layer is connected to all neurons in the output layer. We see this in C5 to F6. A full connection for C5 to F6 can be created in Matlab as

\[
>> cm6 = \text{cnn}\_\text{cm}('\text{full}', 5, 1)
\]

where 5 is the number of feature maps in layer C5, and 1 is the number of neurons in layer F6.

After connection matrices \(cm1, cm2, cm3, cm4, cm5\) and \(cm6\) are created for all six layers, they are combined into one cell array for the network

\[
\begin{align*}
>> c &= \{cm1, cm2, cm3, cm4, cm5, cm6\}
\end{align*}
\]

In this example, the transfer functions for network layers are set as

- convolution layers C1, C3 and C5: 'tansig',
- sub-sampling layers S2 and S4: 'purelin',
- output layer F6: 'tansig'.

The training method for the network is chosen to be 'rprop', which is one of the fastest among the first-order training algorithms:

\[
\begin{align*}
>> \text{train}\_\text{method} &= '\text{rprop}'
\end{align*}
\]

The network structure can be created as

\[
\begin{align*}
>> \text{net} &= \text{cnn}\_\text{new}([H \ W], c, \text{rec}\_\text{size}, \text{tf}\_\text{fcn}, \text{train}\_\text{method});
\end{align*}
\]

### 4.2.3 Training the CNN

To train the network, we call library function \(\text{cnn}\_\text{train}\):

\[
\begin{align*}
>> [\text{new}\_\text{net}, \text{tr}] &= \text{cnn}\_\text{train}(\text{net}, x, d);
\end{align*}
\]

Parameters controlling the training process are stored in structure \(\text{net}\_\text{train}\). Important parameters include

- **epochs**: the maximum number of training epochs;
- **goal**: the target MSE;
- **show**: how often training progress is recorded/displayed. For example, if \(\text{show} = 20\), training error will be displayed every 20 epochs;
- **method**: the name of training method.
The parameters specific to a training algorithm is store in field `net.train.X`, where $X$ is the name of the algorithm. For example, all training parameters for RPROP are stored in structure `net.train.rprop`; the default values for these parameters are

- $\eta_t$: 1.01,
- $\eta_{am}$: 0.99,
- $\delta_{init}$: 0.01,
- $\delta_{max}$: 10.0,

To control the training process, the user can alter all of the default parameters in `net.train` before calling `cnn.train`. Figure 6 shows the training error at different training epoch, for the CNN network with 316 weights and biases and a training set of 2000 images.

![Figure 6: Training error versus epoch for a CNN. Training method: RPROP; network size: 316 weights and biases; training set: 2000 images; image size: 32 × 32 pixels.](image)

### 4.2.4 Testing the CNN

In the following code, we call function `cnn_sim` to compute the network output for a given set of images $x_{test}$.

```matlab
>> load('data\test_data.mat')
>> whos
>> y_test = cnn_sim(new_net, x_test);
>> cr_test = sum((y_test>0)==(d_test>=0))/length(d_test)*100
```

The trained CNN has a classification rate of 98.6% on the training set, and 95.8% on the test set.
5 Matlab implementation notes

All data about a CNN are stored in a Matlab structure. The main fields in this structure are listed in Table 5.

Table 5: Matlab structure net for convolutional neural network.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>number of layer in the network.</td>
</tr>
<tr>
<td>w</td>
<td>trainable weights. net.w{l} stores the weights for layer l.</td>
</tr>
<tr>
<td>b</td>
<td>trainable biases, net.b{l} stores the weights for layer l.</td>
</tr>
<tr>
<td>c</td>
<td>connection matrices. net.c{l} stores connection matrix to layer l.</td>
</tr>
<tr>
<td>rec_size</td>
<td>receptive sizes, net.rec_size(l,:) is a row-vector for the receptive size of layer l.</td>
</tr>
<tr>
<td>hrec_size</td>
<td>half receptive sizes, used internally for image filtering.</td>
</tr>
<tr>
<td>f</td>
<td>transfer functions, net.f{l} is the transfer function of layer l.</td>
</tr>
<tr>
<td>input_size</td>
<td>size of input image to network as [width, height].</td>
</tr>
<tr>
<td>no_fms</td>
<td>number of feature maps in each layer, net.no_fms(l) is the number of feature maps in layer l.</td>
</tr>
<tr>
<td>fm_size</td>
<td>the spatial size (with, height) of the feature maps in each layer, net.fm_size(l,:) is the size of feature maps in layer l.</td>
</tr>
<tr>
<td>layers</td>
<td>text information about each layer, net.layers{l} is for layer l .type: 'C', 'S' or 'F' .name: C1, 'S2'... .connection: '1-to-1', 'full', or 'custom'</td>
</tr>
<tr>
<td>P</td>
<td>total number of trainable parameters (weights and biases) for the network.</td>
</tr>
<tr>
<td>train</td>
<td>parameters that control the training process.</td>
</tr>
</tbody>
</table>

For the example network in Fig. 5, the Matlab structure net is given as follows.

```
net =
    L: 6
    w: {1x6 cell}
    b: {1x6 cell}
    c: {1x6 cell}
    rec_size: [6x2 double]
    hrec_size: [6x2 double]
    f: {6x1 cell}
    input_size: [32 32]
    no_fms: [2 2 5 5 5 1]
    fm_size: [6x2 double]
    layers: {1x6 cell}
    P: 316
    train: [1x1 struct]
```
6 Conclusion

Suggestions to improve the library are welcome. If you use the library in your published work, please advise us via e-mail, and provide references to


7 Appendix

% --------- Example Code: Using the Matlab Library for CNN ---------
% Load training data
load('data\train_data.mat'), whos

%% Create a CNN
H = 32; % height of 2-D input
W = 32; % width of 2-D input
% Create connection matrices
cm1 = cnn_cm('full', 1, 2); % input to layer C1, C1 has 2 planes
cm2 = cnn_cm('1-to-1', 2); % C1 to S2
cm3 = [1 1 0 0 1; 0 0 1 1 1]; % S2 to layer C3, C3 has 5 planes
cm4 = cnn_cm('1-to-1', 5); % C3 to S4
cm5 = cnn_cm('1-to-1', 5); % S4 to C5
cm6 = cnn_cm('full', 5, 1); % C5 to F6
% Receptive sizes for each layer
rec_size = [5 5; % C1
    2 2; % S2
    3 3; % C3
    2 2; % S4
    0 0; % C5 auto calculated
    0 0]; % F6 auto calculated
% Transfer function
tf_fcn = {'tansig', % layer C1
    'purelin', % layer S2
    'tansig', % layer C3
    'purelin', % layer S4
    'tansig', % layer C5
    'tansig'} % layer F6 output
% Training method
train_method = 'rprop'; % 'gd'

net = cnn_new([H W], c, rec_size, tf_fcn, train_method);

% Network training
net.train.epochs = 1100;
[new_net, tr] = cnn_train(net, x, d);
% new_net is trained network, tr is training record
save('data\trained_net.mat', 'new_net', 'net', 'tr');

% Plotting training performance
plot(tr.epoch, tr.mse, 'b-', 'LineWidth', 2); grid
h = xlabel('Epochs'), set(h, 'FontSize', 14);
h = ylabel('Training MSE'), set(h, 'FontSize', 14);
set(gca, 'FontSize', 14);
y = cnn_sim(new_net, x); % network output
cr = sum((y >0) == (d >=0))/length(d)*100;
fprintf('Classification rate (train): cr = %2.2f%%
',cr);

% Network testing
load('data\test_data.mat'), whos
y_test = cnn_sim(new_net, x_test); % network output
cr_test = sum((y_test >0) == (d_test >=0))/length(d_test)*100;
fprintf('Classification rate (test): cr = %2.2f%%
',cr_test);
References


