Visual and Audio Signal Processing Lab
University of Wollongong

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

Convolutional neural network (CNN), originally proposed by LeCun [1], is a neural network model with three key architectural ideas: local receptive fields, weight sharing, and sub-sampling in the spatial domain. The network is designed for the recognition of two-dimensional visual patterns. Convolutional neural network has many strengths. First, feature extraction and classification are integrated into one structure and are fully adaptive. Second, the network extracts 2-D image features at increasing dyadic scales. Third, 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 MATLAB code we wrote in 2006 for a paper on pyramidal neural network [2]. Since then, the library has been revised to optimize speed and improve documentation. The library and this report are aimed at researchers who wish to experiment with convolutional neural networks.

The report is structured as follows. Section 2 describes architectural aspects of the convolutional neural networks. Section 3 presents algorithms for batch training of the networks. Section 4 describes the main functions in this library, and illustrates its usage with an example application. Section 5 discusses the MATLAB implementation. 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 has 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 because each 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 2×2 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 2×2 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.

- In the *last convolution layer*, each plane is connected to exactly one preceding feature map. This layer uses convolution masks that have exactly the same size as its input feature maps.

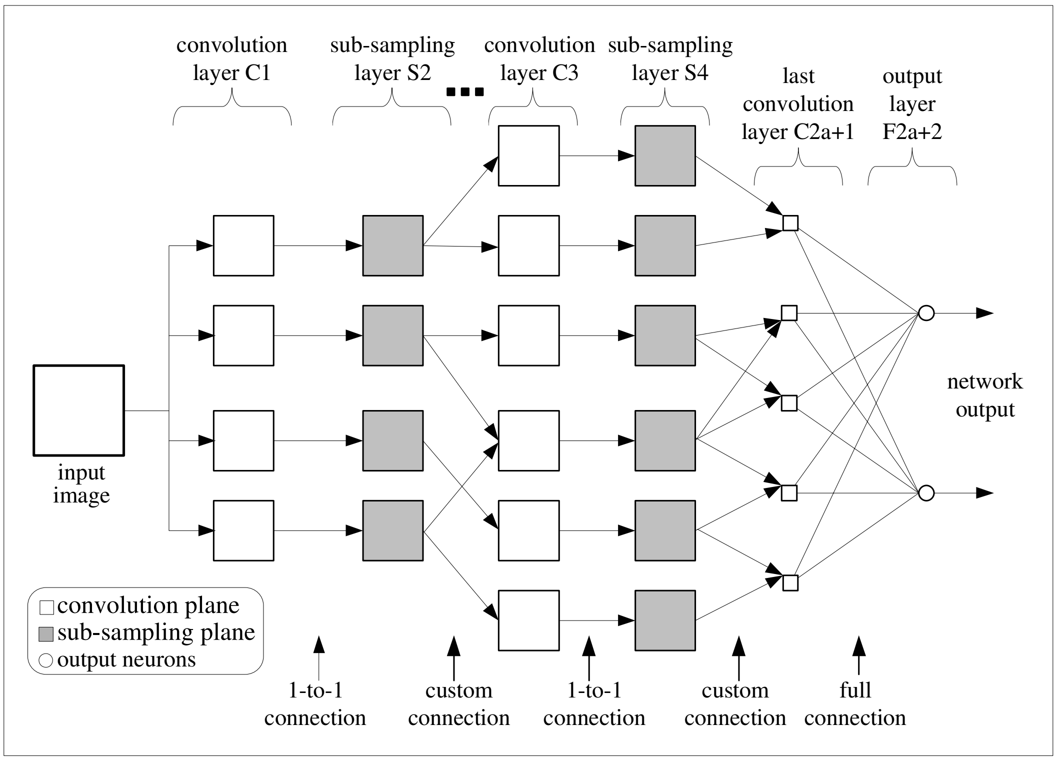


Figure 1: Layers in a convolutional neural network.

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

Table 1 summarizes the notation used to describe the functional aspects of CNN. 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 $f_l(.)$ be the activation function of layer l. Let $\mathbf{y}_n^l$ be the n-th feature map (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

- $\mathbf{w}_{m,n}^l = \{w_{m,n}^l(i, j)\}$ 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,

- $\mathbf{V}_n^l$ denote the list of all planes in layer $(l-1)$ that are connected to feature map n. For example, $\mathbf{V}_4^l = \{2, 3, 5\}$ 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

Description	**Symbol**
Input image size	$H_0 \times W_0$
Input image pixel	$x(i,j)$ or $y_1^0(i,j)$
Layer index	l
Number of layers	$L = 2a + 2$
Convolution layers	C^1, C^2,..., C^{2a+1}
Sub-sampling layers	S^1, S^3,..., S^{2a}
Output layer	F^{2a+2}
Activation function of layer l	f^l
Number of feature maps in layer l	N^l
Size of convolution mask for layer C_l	$r_l \times c_l$
Convolution mask from feature map m in layer S^{l-1} to feature map n in layer C^l	$\{w_{m,n}^l(i,j)\}$
Bias for feature map n in convolution layer C^l	b_n^l
Weight for feature map n in layer S^l	w_n^l
Bias for feature map n in sub-sampling layer S^l	b_n^l
Feature map n in layer l	$y_n^l(i,j)$
Size of a feature map in layer l	$H_l \times W_l$

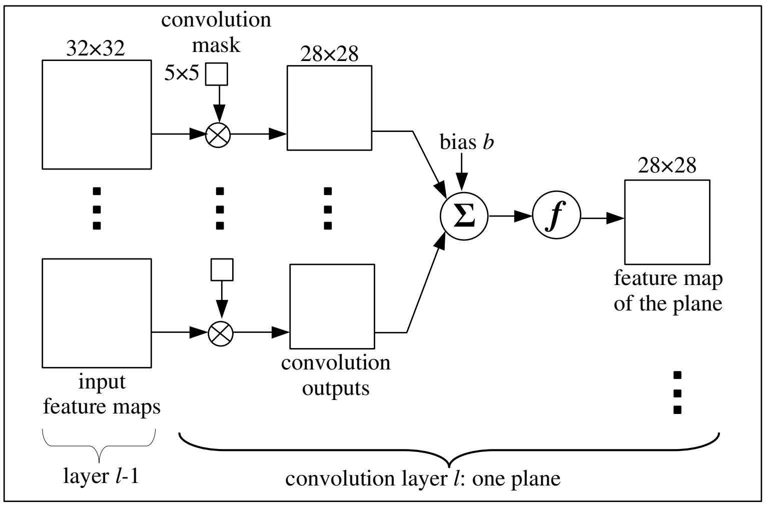


Figure 2: A convolution layer in CNN.

Let $\otimes$ be the 2-D convolution operator. Feature map n of convolution layer l is calculated as

$$\mathbf{y}_n^l = f_l\Big(\sum_{m \in \mathbf{V}_n^l} \mathbf{y}_m^{l-1} \otimes \mathbf{w}_{m,n}^l + b_n^l\Big) \tag{1}$$

Suppose that the size of input feature maps $\mathbf{y}_m^{l-1}$ is $H^{l-1} \times W^{l-1}$ pixels, and the size of convolution masks $\mathbf{w}_{m,n}^l$ is $r^l \times c^l$ pixels. The size of output feature map $\mathbf{y}_n^l$ 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 this structure, l is an even integer, $l = 2, 4, ..., 2a$. For feature map n, let w_n^l be the weight and b_n^l be the bias term. We divide feature map n of convolution layer $(l-1)$ into non-overlapping blocks of size 2×2 pixels. Let $\mathbf{z}_n^{l-1}$ be a matrix obtained by summing the four pixels in each block. That is,

$$z_n^{l-1}(i,j) = y_n^{l-1}(2i-1,2j-1) + y_n^{l-1}(2i-1,2j) + y_n^{l-1}(2i,2j-1) + y_n^{l-1}(2i,2j). \quad (3)$$

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

$$\mathbf{y}_n^l = f_l(\mathbf{z}_n^{l-1} \times w_n^l + b_n^l) \quad (4)$$

Clearly, a feature map $\mathbf{y}_n^l$ in sub-sampling layer l will have a size of $H^l \times W^l$, where

$$H^l = H^{l-1}/2 \quad \text{and} \quad W^l = W^{l-1}/2. \quad (5)$$

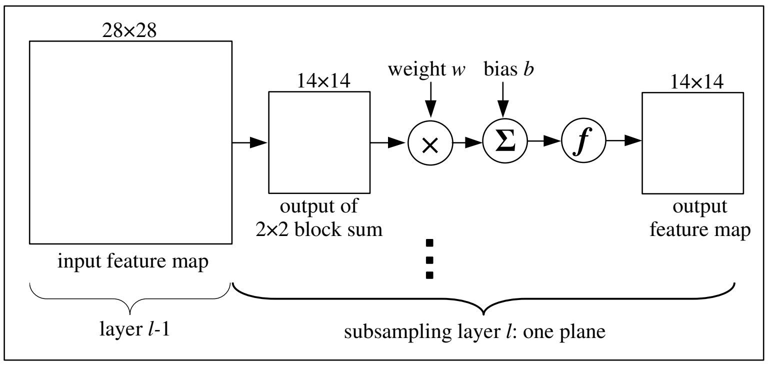


Figure 3: A sub-sampling layer in CNN.

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

$$y_n^L = f^L\left(\sum_{m=1}^{N^{L-1}} y_m^{L-1} w_{m,n}^L + b_n^L\right). \quad (6)$$

The outputs of all sigmoidal neurons form the network outputs:

$$\mathbf{y} = [y_1^L, y_2^L, \ldots, y_{N_L}^L]. \quad (7)$$

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.

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

$$E(\mathbf{w}) = \frac{1}{K \times N_L} \sum_{k=1}^{K} \sum_{n=1}^{N_L} (y_n^k - d_n^k)^2, \quad (8)$$

where y_n^k is the actual network output. This is a function of all network parameters (weights and biases).

There are two major approaches to CNN training. The first approach, known as *online training*, 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 online training is a special case of batch training (when $K = 1$), we will focus on batch training. Note that in batch training, an evaluation of network outputs 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.

Description	**Symbol**	**Formula**
Training image index	k	$k = 1, 2, ..., K$
Training image k	$\mathbf{x}^k$	$\mathbf{x}^k = \{x^k(i,j), i = 1, ..., H_0; j = 1, ..., W_0\}$
Desired output sample k	$\mathbf{d}^k$	$\mathbf{d}^k = (d_1^k, d_2^k, ..., d_{N_L}^k)^T$
Network input or output of layer 0	$y^{0,k}(i,j)$	$(y^{0,k}(i,j) = (x^k(i,j)), i = 1, ..., H_0; j = 1, ..., W_0$
Weighted sum input to neuron (u,v) in convolution layer l, feature map n	$s_n^{l,k}(u,v)$	$n = 1, ..., N_l$ $u = 1, ..., H_l; v = 1, ..., W_l$
Output of neuron (u,v) in convolution layer l, feature map n for input image k	$y_n^{l,k}(u,v)$	$y_n^{l,k}(u,v) = f_l(s_n^{l,k}(u,v))$
Weighted sum input to neuron (u,v) in in subsamplinglayer l, feature map n	$s_n^{l,k}(u,v)$	
Output of neuron (u,v) in in subsamplinglayer l, feature map n	$y_n^{l,k}(u,v)$	$y_n^{l,k}(u,v) = f^l(s_n^{l,k}(u,v))$
The nth error for image k	e_n^k	$e_n^k = y_n^{L,k} - d_n^k$
Error function	$E(\mathbf{w})$	$E(\mathbf{w}) = \frac{1}{K \times N_L} \sum_{k=1}^{K} \sum_{n=1}^{N_L} (y_n^k - d_n^k)^2$
Error sensitivity of pixel (u,v) in 2-D layer l	$\delta_{u,v}^{l,k}$	$\delta_{u,v}^{l,k} = \partial E / \partial s_{u,v}^{l,k}, \quad l \leq L_p$
Error sensitivity of neuron n in 1-D layer l	$\delta_n^{l,k}$	$\delta_n^{l,k} = \partial E / \partial s_n^{l,k}, \quad l > L_p$

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 the definition of error sensitivity, and methods to calculate error sensitivity, and the error gradient.

3.1.1 Error sensitivity definition

The error 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

$$\delta_n^{l,k}(i,j) = \frac{\partial E}{\partial s_n^{l,k}(i,j)}, \text{ for } l = 1, 3, ..., 2a+1. \tag{9}$$

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

$$\delta_n^{l,k}(i,j) = \frac{\partial E}{\partial s_n^{l,k}(i,j)}, \text{ for } l = 2, 4, ..., 2a. \tag{10}$$

For neuron n in output layer L, its error sensitivity is

$$\delta_n^{L,k} = \frac{\partial E}{\partial s_n^{L,k}}. \tag{11}$$

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: $l = L$

$$\delta_n^{L,k} = \frac{2}{K \times N_L}\; e_n^k \; f_L'(s_n^{L,k}), \text{ for } n = 1, 2, ..., N_L. \tag{12}$$

- Last convolution layer: $l = L - 1$

$$\delta_n^{l,k} = f_l'(s_n^{l,k}) \sum_{m=1}^{N_{l+1}} \delta_n^{l+1,k}\; w_{n,m}^{l+1}, \text{ for } n = 1, 2, ..., N_l. \tag{13}$$

  Because feature maps in convolution layer $(L-1)$ have a size of 1×1 pixel, writing $\delta_n^{L-1,k}$ or $\delta_n^{L-1,k}(1,1)$ is equivalent.

- Last sub-sampling layer: $l = L - 2$

$$\delta_n^{l,k}(i,j) = f_l[s_n^{l,k}(i,j)] \times w_{n,n}^{l+1}(i,j) \times \delta_n^{l+1,k}(1,1), \tag{14}$$

  where $n = 1, 2, ..., N_{L-2}$; $i = 1, 2, ..., H_{L-2}$; $j = 1, 2, ..., W_{L-2}$.

- Other convolution layer: $l = 2a + 1$

$$\delta_n^{l,k}(i,j) = f_l[s_n^{l,k}(i,j)] \times \delta_n^{l+1}(i',j') \times w_n^{l+1,k}, \tag{15}$$

  where $i' = \lfloor i/2 \rfloor$ and $j' = \lfloor j/2 \rfloor$.

- Other sub-sampling layer: $l = 2a$

$$\delta_n^{l,k}(i,j) = f_l[s_n^{l,k}(i,j)] \times \sum_{m \in U_n^l} \sum_{(i',j') \in R^l(i,j)} \delta_m^{l+1,k}(i',j')\; w_{n,m}^{l+1}(i-i', j-j'). \tag{16}$$

3.1.3 Error gradient computation

- Output layer: $l = L$
  -Weights $w_{m,n}^l$:

$$\frac{\partial E}{\partial w_{m,n}^l} = \sum_{k=1}^{K} \delta_n^{l,k}\; y_m^{l-1,k}, \tag{17}$$

  where $m = 1, 2, ..., N_{l-1}$ and $n = 1, 2, ..., N_l$.
  -Biases b_n^l:

$$\frac{\partial E}{\partial b_n^l} = \sum_{k=1}^{K} \delta_n^{l,k}, \tag{18}$$

  where $n = 1, 2, ..., N_l$.

- Last convolution layer: $l = L - 1$

  -Weights $w^l_{m,n}(i,j)$:

$$\frac{\partial E}{\partial w^l_{m,n}(i,j)} = \sum_{k=1}^{K} \delta^{l,k}_n(i,j)\ y^{l-1,k}_{n,n} \tag{19}$$

  where $n = 1, 2, ..., N_l$.

  -Biases b^l_n:

$$\frac{\partial E}{\partial b^l_n} = \sum_{k=1}^{K} \sum_{(i,j)} \delta^{l,k}_n(i,j) \tag{20}$$

  where $n = 1, 2, ..., N_l$.

- Sub-sampling layer: $l = 2a$

  -Weights w^l_n:

$$\frac{\partial E}{\partial w^l_n} = \sum_{k=1}^{K} \sum_{(i',j')} \delta^{l,k}_n(i',j')\ y^{l-1,k}_n(i',j') \tag{21}$$

  where $n = 1, 2, ..., N_l$.

  -Biases b^l_n:

$$\frac{\partial E}{\partial w^l_n} = \sum_{k=1}^{K} \sum_{(i',j')} \delta^{l,k}_n(i',j') \tag{22}$$

  where $n = 1, 2, ..., N_l$.

- Other convolution layer: $l = 2a + 1$

  -Weights $w^l_{m,n}(i,j)$:

$$\frac{\partial E}{\partial w^l_{n,n}(i,j)} = \sum_{k=1}^{K} \delta^{l,k}_n(i,j)\ y^{l-1,k}_{n,n} \tag{23}$$

  where $n = 1, 2, ..., N_l$.

  -Biases b^l_n:

$$\frac{\partial E}{\partial b^l_n} = \sum_{k=1}^{K} \sum_{(i,j)} \delta^{l,k}_n(i,j) \tag{24}$$

  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 3: CNN training algorithms.

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

Computation of the Jacobian matrix is similar to computation of the gradient ∇E. However, we need to modify the definitions of error sensitivities. For the Jacobian matrix, error sensitivities are 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 in 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: *cnn_new*, *cnn_init*, *cnn_sim*, and *cnn_train*.

Table 4: Main MATLAB functions in the CNN library.

Name	**Description**
cnn_new	Create a new network
cnn_init	Initialize a network
cnn_cm	Create a connection matrix for a network layer
cnn_sim	Compute network ouput
cnn_train	Train a network
cnn_train_gd	Train a CNN using gradient descent method
cnn_train_rprop	Train a CNN using resilient backpropagation method
cnn_compute_gradient	Compute gradient of the error function for CNN

4.1.1 Function cnn_new

The syntax of this function is given as

```
net = cnn_new(input_size, c, rec_size, tf_fcn, train_method)
```

where

- *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 input_size = [36 32].
- *c*: a cell array of connection matrices. Cell $c\{l\}$ stores the connection matrix for layer l. That is, $c\{l\}(i, j)$ is set to *true* if there is a connection from feature map i of layer $l-1$, to feature map j of layer l. A MATLAB function *cnn_cm* has been written to create standard connection matrices.
- *rec_size*: size of receptive fields. It is a two-column matrix where rec_size(l, :) is the receptive size of layer l. Note that the receptive fields of the sub-sampling layer is always [2 2].
- *tf_fcn*: transfer functions of network layers. It is a cell array of strings. Possible MATLAB transfer functions are 'tansig', 'logsig', 'purelin' and 'ltanh'.
- *train_method*: training method for the network, as a string. Training methods that have been implemented are 'gd' and 'rprop'.

4.1.2 Function cnn_init

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

```
new_net = cnn_init(net)
```

Note that when a network is created using *cnn_new*, its weights and biases are automatically initialized, so calling *cnn_init* is not needed.

4.1.3 Function cnn_sim

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

```
y = cnn_sim(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

```
[new_net, new_tr] = cnn_train(net, x, d)
```

where

- *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,
- *tr*: existing training record (optional),
- *new_net*: trained network,
- *new_tr*: updated training record.

To support analysis of the training process, several performance indicators are recorded in a MATLAB structure at defined training epoches. The indicates are stored in a MATLAB structure *new_tr*, which includes the following fields

- *mse*: mean-square-errors,
- *time*: training time in seconds,
- *epoch*: epoch count,
- *output_eval*: number of output evaluations,
- *gradient_eval*: number of gradient evaluations,
- *hessian_eval*: number of Hessian matrix evaluations,
- *jacobian_eval*: number of Jacobian matrix evaluations,

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

```
plot(new_tr.time, new_tr.mse);
```

If an existing training record is passed to the function in *tr* parameter, *cnn_train* will append new training information to it and return an updated record in *tr_new*.

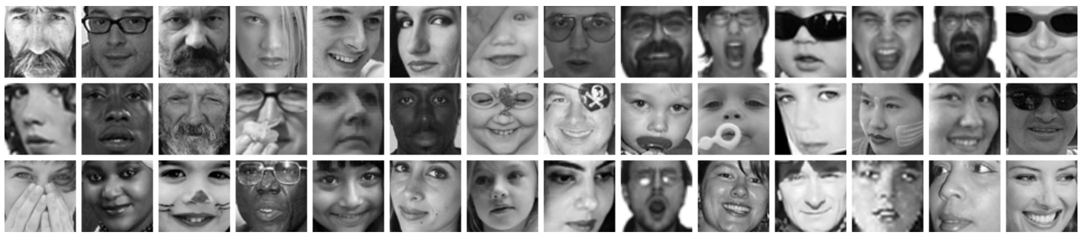
To perform the actual training, *cnn_train* will call another MATLAB function; the name of this function is determined by the field *net.train.method*. For example, if *net.train.method* is '*gd*', training will be done by *cnn_train_gd*:

```
[new_net, new_tr] = cnn_train_gd(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.



(a) face patterns

(b) nonface patterns

Figure 4: Examples of training images.

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 32×32 pixels. To load the training data, type

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

The data are

```
>> whos
  Name        Size                        Bytes  Class
  d           1x2000                      16000  double
  x           32x32x2000               16384000  double
```

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

$$d(k) = \begin{cases} 1, & \text{if sample } k \text{ is a face pattern,} \\ -1, & \text{if sample } k \text{ is a non} - \text{face pattern.} \end{cases} \tag{25}$$

4.2.2 Creating a CNN

We will create a CNN as shown in Fig. 5. This network accepts an input image of size 32×32 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

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

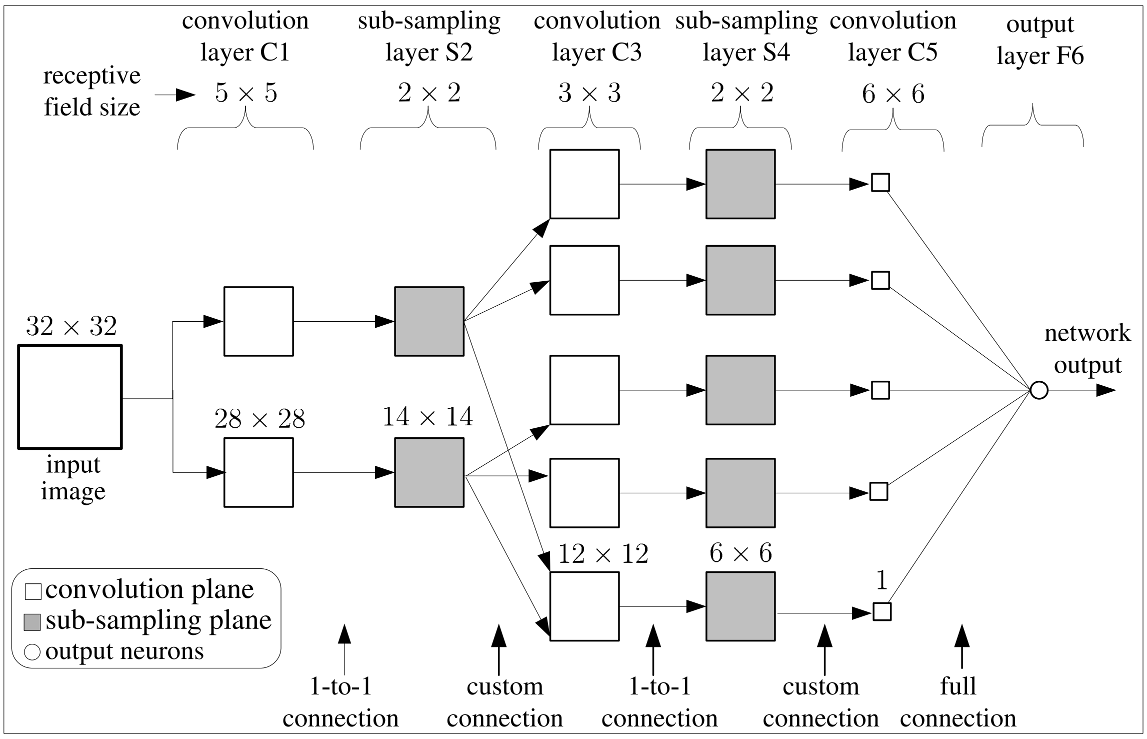


Figure 5: An example of convolutional neural network.

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

```
    >> 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

  ```
  >> cm3 = [1 1 0 0 1;
            0 0 1 1 1]
  ```

  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 = cnn_cm('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

  ```
  >> c = {cm1, cm2, cm3, cm4, cm5, cm6}
  ```

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:

```
>> train_method = 'rprop';
```

The network structure can be created as

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

4.2.3 Training the CNN

To train the network, we call library function *cnn_train*:

```
>> [new_net, tr] = cnn_train(net, x, d);
```

Parameters controlling the training process are stored in structure *net.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 *show* = 20, training error will be displayed every 20 epochs;
- *method*: the name of training method.

The parameters specific to a training algorithm is stored in the 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

- *etap*: 1.01,
- *etam*: 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.

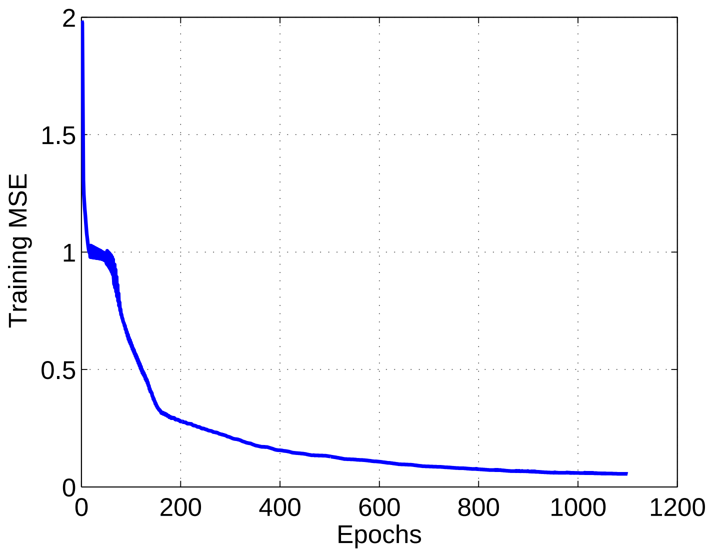


Figure 6: Training error versus epoch for a CNN. Training method: RPROP; network size: 316 weights and biases; training set: 2000 images; image size: 32 × 32 pixels.

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*.

```
>> 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.

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

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:

1. S. L. Phung and A. Bouzerdoum, "*MATLAB library for convolutional neural network*," Technical Report, ICT Research Institute, Visual and Audio Signal Processing Laboratory, University of Wollongong. Available at: http://www.uow.edu.au/˜phung.
2. S. L. Phung and A. Bouzerdoum, "*A pyramidal neural network for visual pattern recognition*," IEEE Transactions on Neural Networks, vol. 27, no. 1, pp. 329343, 2007.

This library has been used by:

University of Missouri (USA), Chinese Academy of Sciences (China), Seoul National University (Korea), National University of Sciences and Technology (Pakistan), University of Tennessee (USA), Beijing Normal University (China), University of New Brunswick (Canada), University of Bologna (Italy), National University of Singapore (Singapore), University of Western Australia (Australia), The Johns Hopkins University (USA), Jawaharlal Nehru University (India), University of Manchester (UK), Sharif University of Technology (Iran), South China University of Technology (China), Shanghai Jiao Tong University (China), Indian Statistical Institute (India), Yildiz Technical University (Turkey), University of Belgrade (Serbia), University of Naples (Italy), K. N. Toosi University of Technology (Iran), Beijing University of Technology (China), University of Innsbruck (Austria), Politehnica University in Bucharest (Romania).

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
c = {cm1, cm2, cm3, cm4, cm5, cm6};
% 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'

% Create CNN
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%%\n',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%%\n',cr_test);
```

References

[1] 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.

[2] S. L. Phung and A. Bouzerdoum, "A pyramidal neural network for visual pattern recognition," *IEEE Transactions on Neural Networks*, vol. 27, no. 1, pp. 329–343, 2007.

[3] 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.* Cambridge, MA: Bradford Books, 1986, vol. I, p. 318 362.

[4] M. T. Hagan, H. B. Demuth, and M. H. Beale, *Neural network design.* Boston, MA: PWS Publishing, 1996.

[5] M. Riedmiller and H. Braun, "A direct adaptive method of faster backpropagation learning: The rprop algorithm," in *IEEE International Conference on Neural Networks*, San Francisco, 1993, pp. 586–591.

[6] E. K. P. Chong and S. H. Zak, *An introduction to optimization.* New York: John Wiley and Sons, Inc., 1996.

[7] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[8] C. Charalambous, "A conjugate gradient algorithm for the efficient training of artificial neural networks," *IEE Proceedings Part G*, vol. 139, no. 3, pp. 301–310, 1992.

[9] S. L. Phung, A. Bouzerdoum, and D. Chai, "Skin segmentation using color pixel classification: analysis and comparison," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 1, pp. 148–154, 2005.