

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/299437550>

A Tutorial on Deep Neural Networks for Intelligent Systems

Technical Report · March 2016

DOI: 10.13140/RG.2.1.2420.6481

CITATION

1

READS

988

3 authors, including:



Juan Carlos Cuevas-Tello

Universidad Autónoma de San Luis Potosí

54 PUBLICATIONS 280 CITATIONS

[SEE PROFILE](#)



Manuel Valenzuela-Rendón

Tecnológico de Monterrey

64 PUBLICATIONS 968 CITATIONS

[SEE PROFILE](#)

A Tutorial on Deep Neural Networks for Intelligent Systems

Juan C. Cuevas-Tello^{1,2} Manuel Valenzuela-Rendón¹
 Juan A. Nolasco-Flores¹

¹Tecnológico de Monterrey, Campus Monterrey
 Av. Eugenio Garza Sada 2501 Sur, C.P. 64849,
 Monterrey, N.L., Mexico

{cuevastello,valenzuela,jnolasco}@itesm.mx

²Engineering Faculty, UASLP
 Dr. Manuel Nava No. 8, Zona Universitaria, C.P. 78290
 San Luis Potosi, SLP, Mexico
 cuevas@uaslp.mx

March 24, 2016

Abstract

Developing Intelligent Systems involves artificial intelligence approaches including artificial neural networks. Here, we present a tutorial of Deep Neural Networks (DNNs), and some insights about the origin of the term “deep”; references to deep learning are also given. Restricted Boltzmann Machines, which are the core of DNNs, are discussed in detail. An example of a simple two-layer network, performing unsupervised learning for unlabeled data, is shown. Deep Belief Networks (DBNs), which are used to build networks with more than two layers, are also described. Moreover, examples for supervised learning with DNNs performing simple prediction and classification tasks, are presented and explained. This tutorial includes two intelligent pattern recognition applications: handwritten digits (benchmark known as MNIST) and speech recognition.

1 Introduction

Intelligent systems involve artificial intelligence approaches including artificial neural networks. This paper focus mainly on Deep Neural Networks (DNNs).

The core of DNNs are the Restricted Boltzmann Machines (RBMs) proposed by Smolensky [23, 10], and widely studied by Hinton et al. [13, 12, 11], where the term *deep* comes from Deep Beliefs Networks (DBN) [12]. The next section describes the relationship among RBMs, DBN and DNNs.

Nowadays, the term *Deep Learning* (DL) is becoming popular in the machine learning literature [15, 3, 22]. However, DL mainly refers to Deep Neural Networks (DNNs) and in particular to DBNs and RBMs [15]. Some work related to DL is focusing on high performance computing to speed up the learning of DNNs, i.e. Graphics Processing Units (known as GPUs), Message Passing Interface (MPI) among other parallelization technologies [3].

A wide survey on artificial intelligence and in particular DL has been published recently, which covers DNNs, Convolutional Neural Networks, Recurrent Neural Networks, among many other learning strategies [22].

A Restricted Boltzmann Machine (RBM) is defined as

a single layer of hidden units which are not connected to each other and have undirected, symmetrical connections to a layer of visible units. The visible units and the hidden states are sampled from their conditional distribution using Gibbs sampling by running a Markov chain until it reaches its stationary distribution. The learning rule is the same as the maximum likelihood learning rule [contrastive divergence] for the infinite logistic belief net with tied weights [12].

Products of Experts (PoE) and Boltzmann machines are probabilistic generative models, and their intersection comes up with RBMs [10]. Learning by contrastive divergence of PoE is the basis of the learning algorithm of DBNs [10, 12].

We recommend [6] as a gentle introduction that explains the training of RBMs and their relationship to graphical models including Markov Random Fields (MRFs); it also presents Markov chains to explain how a RBM draws samples from probability distributions such as Gibbs distribution of a MRF.

The building blocks of a RBM are binary stochastic neurons [12]. Nevertheless, there are several ways to define real-valued visible neurons, where Gaussian-Binary-RBM are widely used [6].

We use a publicly available MATLAB[®]/Octave toolbox for RBMs developed by Tanaka and Okutomi [24]. This toolbox implements sparsity [16], dropout [4] and a novel inference for RBM [24].

The main contribution of this tutorial are the DNNs examples along the source code (Matlab/Octave) to build intelligent systems. Therefore, the spirit of this tutorial is that people can easily execute the examples and see what kind of results are obtained. There are examples with either unsupervised or supervised learning, and examples for prediction and classification tasks are also provided. Moreover, the parameter setting of DNNs with an example is shown.

This tutorial is organized as follows: The following section (§2) describes RBMs. Section §3 describes the toolbox developed by Tanaka and Okutomi [24] and the database MNIST. Section §4 presents a discussion about the parameter settings of DNNs. Section §5 explains some simple examples of DNNs. The last section presents speech processing with DNNs; §6. Finally, a summary and references are given.

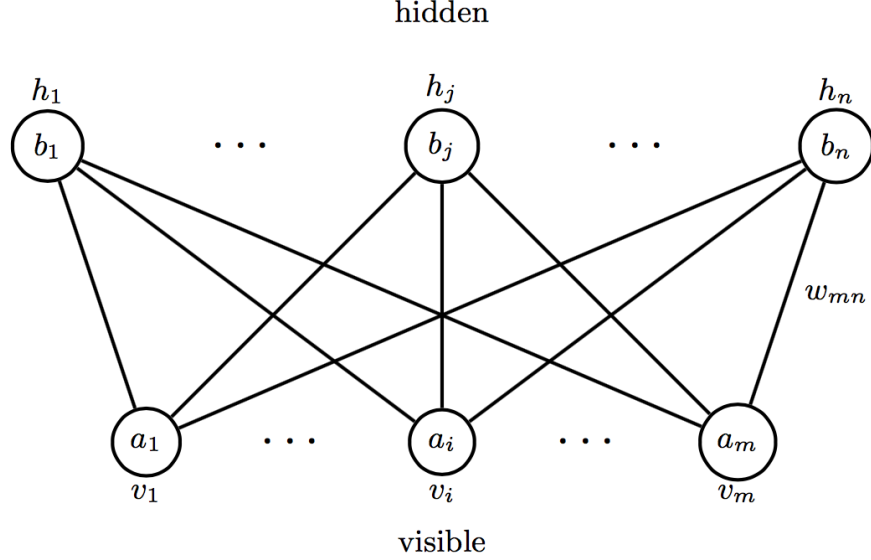


Figure 1: A RBM with a visible layer and a hidden layer.

2 Restricted Boltzmann Machines (RBMs)

A RBM is depicted in Fig. 1. The visible layer is the input, unlabeled data, to the neural network. The hidden layer grabs features from the input data, and each neuron captures a different feature [12]. By definition, a RBM is a bipartite undirected graph. A RBM has m visible units $\vec{V} = (V_1, V_2, \dots, V_m)$, the input data, and n hidden units $\vec{H} = (H_1, H_2, \dots, H_n)$, the features [6]. A joint configuration, (\vec{v}, \vec{h}) of the visible and hidden units has an energy given by [14]

$$E(\vec{v}, \vec{h}) = - \sum_{i=1}^m a_i v_i - \sum_{j=1}^n b_j h_j - \sum_{i=1}^m \sum_{j=1}^n v_i h_j w_{ij} , \quad (1)$$

where v_i and h_j are the binary states of the visible and hidden units, respectively; a_i, b_j are the biases, and w_{ij} is a real valued weight associated with each edge in the network [11], see Fig. 1.

The building block of a RBM is a binary stochastic neuron [12]. Fig. 2 shows how to obtain the state of a hidden neuron given a visible layer (data).

A RBM can be seen as a stochastic neural network. First, weights w_{ij} are randomly initialized. Then, the data to be learned is set at the visible layer; this data can be an image, a signal, etcetera. Now, the state of the neurons at

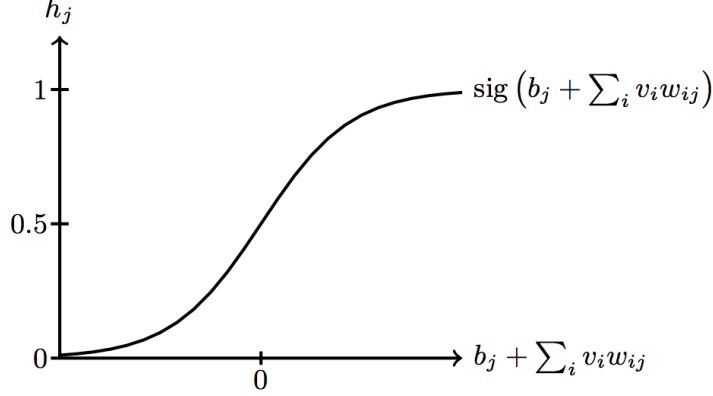


Figure 2: The building block of a RBM: a binary stochastic neuron. In this example, visible to hidden, h_j is the probability of producing a spike [11].

the hidden layer is obtained by

$$p(h_j = 1|\vec{v}) = \text{sig} \left(b_j + \sum_i v_i w_{ij} \right), \quad (2)$$

so the conditional probability of h_j being 1 is the firing rate of a stochastic neuron with a sigmoid activation function, $\text{sig}(x) = 1/(1 - e^{-x})$, see Fig. 2. This step, visible to hidden, is represented as $\langle v_i h_j \rangle^0$, at time $t = 0$ [12, 6, 24].

2.1 Contrastive Divergence algorithm

Learning in a RBM is achieved by the Contrastive Divergence (CD) algorithm, see Fig. 3 [12]. The first step of the CD algorithm is $\langle v_i h_j \rangle^0$, as shown above. The next step is the “reconstruction” of the visible layer by

$$p(v_i = 1|\vec{h}) = \text{sig} \left(a_i + \sum_j h_j w_{i,j} \right), \quad (3)$$

i.e., hidden to visible. This step is denoted as $\langle h_j v_i \rangle^0$. The new state of the hidden layer is obtained using the result of the reconstruction as the input data, and this step is denoted as $\langle v_i h_j \rangle^1$; at time $t = 1$. Finally, the weights and biases are adjusted in the following way [12]:

$$\Delta w_{ij} = \varepsilon (\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1); \quad (4)$$

$$\Delta a_i = \varepsilon (v_i^0 - v_i^1); \quad (5)$$

$$\Delta b_j = \varepsilon (h_j^0 - h_j^1); \quad (6)$$

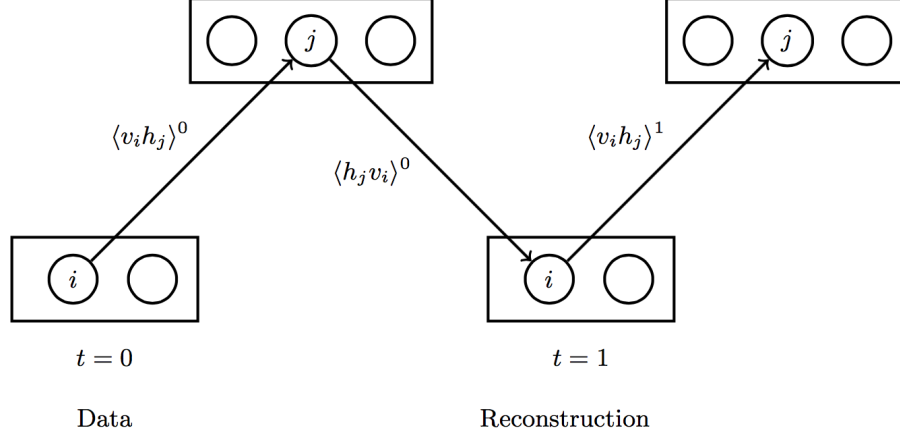


Figure 3: Contrastive Divergence (CD) algorithm.

ALGORITHM 1: Pseudocode of the Contrastive Divergence Algorithm, CD_k .

```

1 Set the visible units to a training vector ;
2 for  $k \leftarrow 1$  to maximum of iterations do
3   for  $s \leftarrow 1$  to size of training data do
4     Update all the hidden units in parallel with Eq. 2 ;
5     Update all visible units in parallel to get “reconstructions” with Eq. 3 ;
6     Update all hidden units again with Eq. 2 ;
7     Update weights and biases with Eqs. 4-6 ;
8     Select another training vector ;
9   end
10 end

```

where ε is the learning rate. RBMs find better models if more steps of the CD algorithm are performed; CD_k is used to denote the learning in k steps/iterations [11]. The CD algorithm is summarized in Algorithm 1, and it uses the complete training data, *batch learning* [6].

2.2 Deep Belief Network

A Deep Belief Network (DBN) [12] is depicted in Fig. 4. Comparing Fig. 1 with Fig. 4, we can see that a DBN is built by stacking RBMs. Thus, the more levels the DBN has, the deeper the DBN is. The hidden neurons in a RBM_1 capture the features from the visible neurons. Then, those features become the input to RBM_2 , and so on until the RBM_r is reached; see also Fig. 5. A DBN extracts features from features in an unsupervised manner (deep learning).

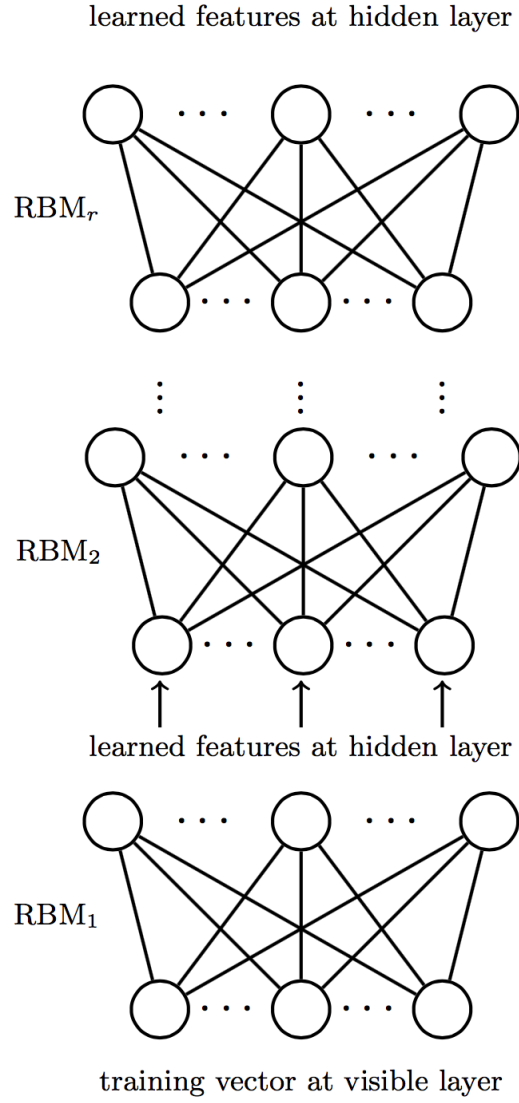


Figure 4: An infinite belief network. The more RBMs, the deeper the learning; i.e. Deep Belief Networks (DBN).

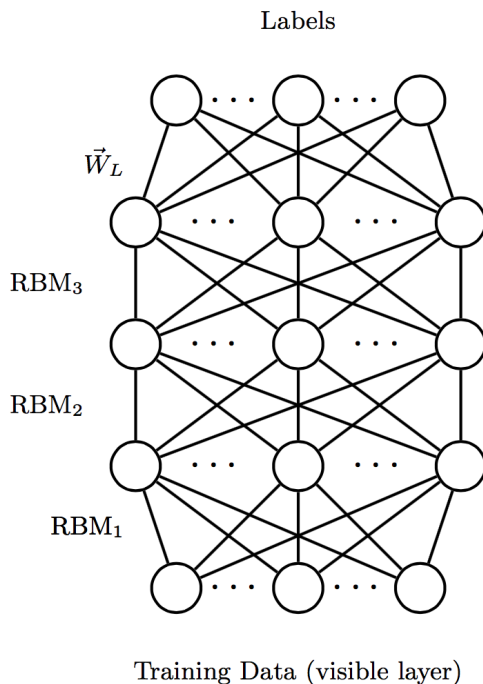


Figure 5: An hybrid DBN for supervised learning.

A hybrid DBN has been proposed for supervised learning, see Fig. 5. This network adds labels to the top layer. The weights \vec{W}_L between the top level and the last layer of hidden neurons, associative memory, are learned in a supervised manner. This process is called fine-tuning [12], and it can be achieved by many different algorithms including backpropagation [21, 1, 19, 8]. This hybrid DBN is referred as Deep Neural Networks [16, 4, 6, 24].

Hinton et al. applied a DNN to the MINST handwritten digits database¹ [12], see Fig. 6. At that time, the DNN produced the best performance with an error rate of 1.25% compared with other methods including Support Vector Machines (SVM) which had an error rate of 1.4% [12].

3 Toolbox for DNN

We use a publicly available toolbox for MATLAB[®] developed by Tanaka and Okutomi [24], and which can be downloaded online². This toolbox is based on [12]. This toolbox includes sparsity [16], dropout [4] and a novel inference

¹<http://yann.lecun.com/exdb/mnist/>

²<http://www.mathworks.com/matlabcentral/fileexchange/42853-deep-neural-network>

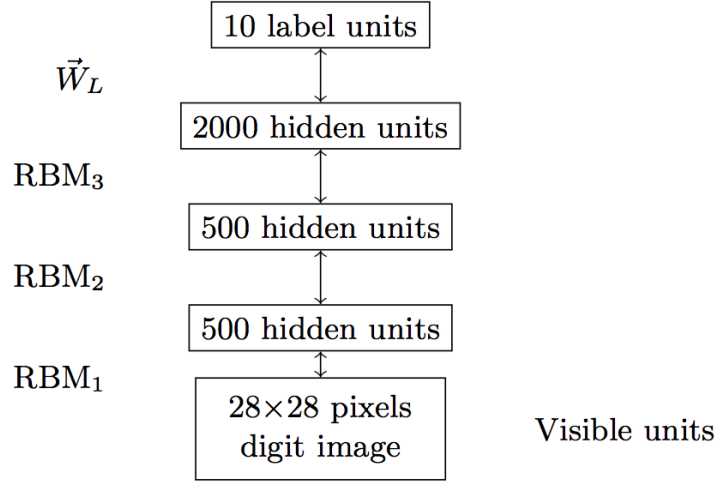


Figure 6: An hybrid DBN for supervised learning [12]; the MNIST database.

for RBM devised by Tanaka [24]. Once the toolbox has been downloaded and unzipped, it will generate the following directories:

- /DeepNeuralNetwork/
- /DeepNeuralNetwork/mnist

3.1 MNIST

The MNIST database³ of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. Once the MNIST has been downloaded and unzipped, we will come up with the following files:

- train-images-idx3-ubyte: training set images
- train-labels-idx1-ubyte: training set labels
- t10k-images-idx3-ubyte: test set images
- t10k-labels-idx1-ubyte: test set labels

Note that when you uncompress the *.gz files, then you will need to check the file names, and replace “.” by “-”. You must locate the files within the /DeepNeuralNetwork/mnist/ directory.

³<http://yann.lecun.com/exdb/mnist/>

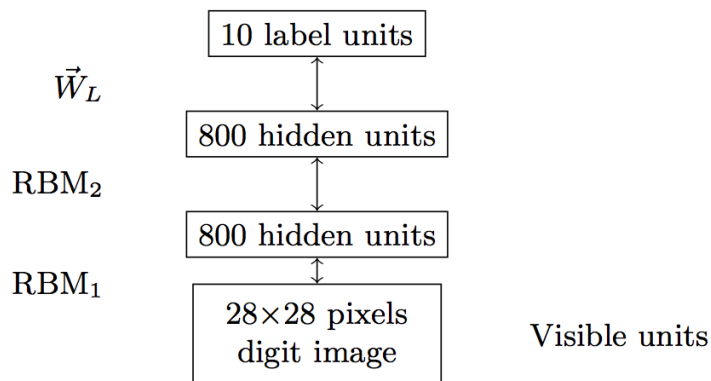


Figure 7: Another DNN architecture for the MNIST database [24].

3.2 Running the example: DNN-MNIST

The file `/mnist/testMNIST.m` is the main file of the example provided by the toolbox to train a DNN for the MNIST database. The example uses a hybrid network with only two hidden layers of 800 neurons each layer, see Fig. 7. We have tested the toolbox on Octave⁴ 3.2.4 and MATLAB[®] 7.11.0.584 (2010b), both in Linux Operating Systems.

The script `testMNIST.m` will generate the file `mnistbbdbn.mat` with the DNN already trained. Once `testMNIST.m` has finished it will appear something like:

- For training data: `rmse = 0.0155251`; `ErrorRate = 0.00196667` (0.196%); Tanaka et al. reported 0.158% [24].
- For test data: `rmse = 0.0552593`; `ErrorRate = 0.0161` (1.6%); Tanaka et al. reported 1.76% [24].

The computational time required to train 60,000 MNIST examples and 10,000 examples for testing is about 3 days on a computer with 384 GB memory, 4 CPUs 2.3GHz with 16 cores each (total of 64 cores).

3.3 Understanding the toolbox example for MNIST

Once the example script (`testMNIST.m`) has been successfully executed, we run the script in Fig. 8.

⁴<https://www.gnu.org/software/octave/>

```

% Matlab/Octave script
mnistfilenames = cell(4,1);
mnistfilenames{1} = 'train-images-idx3-ubyte';
mnistfilenames{2} = 'train-labels-idx1-ubyte';
mnistfilenames{3} = 't10k-images-idx3-ubyte';
mnistfilenames{4} = 't10k-labels-idx1-ubyte';
[TrainImages TrainLabels TestImages TestLabels]=mnistread(mnistfilenames);
% load data for training and testing from files
load mnistbbdbn; % load the trained DNN
dbn = bbdbn; % set dbn as the trained net
N = 10; %number of test data to analyze
IN = TestImages(1:N,:); %load only N records of testing data
OUT = TestLabels(1:N,:); %load only N records of testing data
for i=1:N,
    imshow(reshape(IN(i,:),28,28));
    name = ['print img-training-',num2str(i),'.jpg -djpeg'];
    eval(name); %save the plot, file in jpeg format
end
% v2h: get the output of the DNN
out = v2h( dbn, IN );
% get the maximum values (m) of out and indexes (ind)
[m ind] = max(out,[],2);
out = zeros(size(out)); % initialize the variable out
% Now, fill with ones where the maximum values where located (ind):
for i=1:size(out,1)
    out(i,ind(i)) = 1;
end
% Now compare out vs OUT. Let say the output of the DNN (out) vs
% the desired output (OUT)
ErrorRate = abs(OUT-out); % analytically compare OUT vs out
% sum(ErrorRate,2) performs the sum in two dimensions,
% first by row then the resulting column.
% It is divided by 2; if some output fails, the sum will count twice.
% mean gives us the percentage of error, known as error rate.
ErrorRate = mean(sum(ErrorRate,2)/2) % finally the Error rate is obtained

```

Figure 8: Matlab/Octave script: analysis of $N = 10$ test images from MNIST database.

This script generates $N = 10$ images via `imshow`, see Fig. 9. The images are part of the 10 first testing samples. Each image is stored in a vector of size 784, which corresponds to an image size of 28×28 pixels. And each pixel stores a number between 0 and 255, where 0 means background (white) and 255 means foreground (black); see Fig. 9.

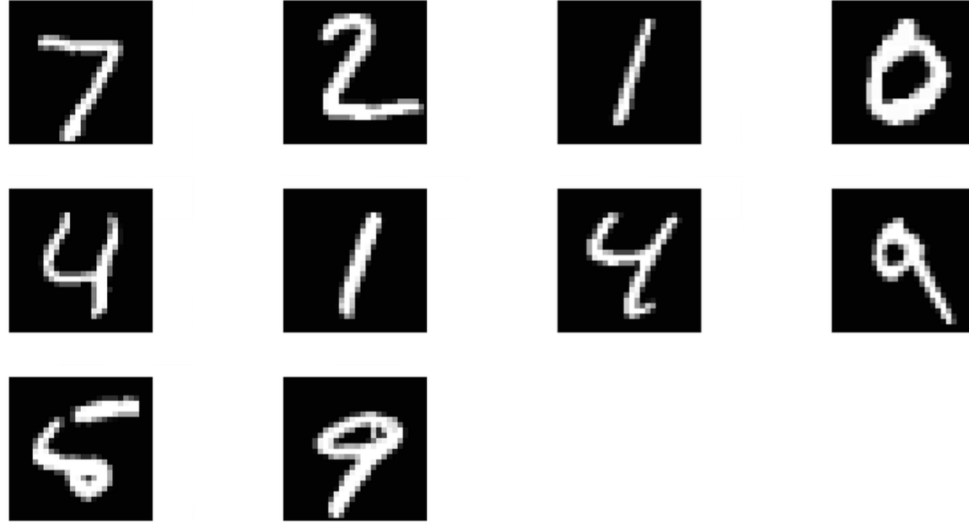


Figure 9: The first 10 testing examples of the MNIST database. The first one is the digit 7, then 2; the last one is the digit 9.

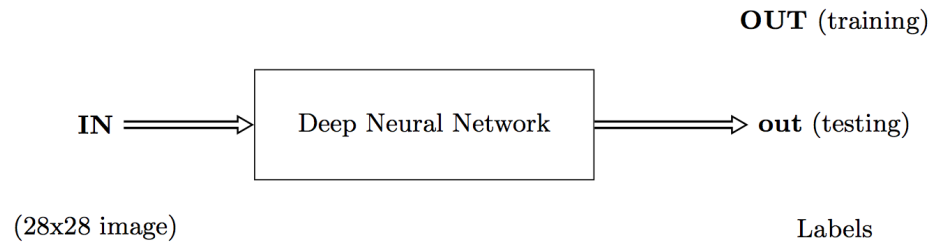


Figure 10: Inputs and outputs of a DNN for the MNIST database.

In Fig. 10, we depict the inputs and outputs of the DNN for the MNIST database. The variable **IN** represents the inputs for either training or testing samples. The variable **OUT** is the output for training, and the variable **out** the output for testing. Both output variables represent the labels of the digits to learn/recognize (digits from 0 to 9).

For example, the script in Fig. 8 generates the following result:

```

out =
0  0  0  0  0  0  0  1  0  0
0  0  1  0  0  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  0  0  0  0  0  0  0  0  1
0  0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  0  0  0  1

```

The variable **out** represents the labels. Each row correspond to each digit image⁵, and each column means the activation or not of the digits 0–9. That is, the first column represents the digit 0, and the last column the digit 9. For example, see top-left in Fig. 9, the image representing the digit 7 activates only column 8 (i.e. 0 0 0 0 0 0 0 1 0 0) of the first row of variable **out**. The image at the right side of digit 7 corresponds to digit 2, so the third column is activated on the second row of variable **out**, and so on. In this example the **ErrorRate** is zero, because the first ten samples of testing are all recognized successfully. Let us create an hypothetical scenario where the DNN fails to recognize the first image (digit 7), i.e. imagine that the output looks like the following:

```

out =
0  0  0  0  0  0  1  0  0  0
0  0  1  0  0  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0
0  0  0  0  0  0  0  0  0  1
0  0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  0  0  0  1

```

Look at the first row, now this row indicates that the first digit image in Fig. 9 corresponds to the digit 6, instead of digit 7. Therefore, **Errorrate** measures this error via the **abs** function along the difference between the desired output **OUT** and the output of the DNN, which is **out**. Then the error rate is obtained as follows:

⁵Note that there are 60,000 images for training, 10,000 images for testing and 10 images for illustration purposes (Fig. 9), i.e. only 10 rows.

```
ErrorRate = abs(OUT-out)
ErrorRate =
    0    0    0    0    0    0    1    1    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
```

Now, we sum out by row. This is to detect how many differences are found by test sample.

```
sum(ErrorRate,2)
ans =
    2
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
```

If an error exists, then the result is divided by 2. This is because if there is some difference per each sample we will have two 1's as show above.

```

sum(ErrorRate,2)/2
ans =
    1
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0

```

Finally, the error rate is given as the mean value:

```

mean(sum(ErrorRate,2)/2)
ans = 0.10000

```

Since the DNN fails to recognize one digit out of ten, the error rate is 10% (i.e. 0.10000).

4 Parameter Setting

There are several parameters to set up when working with DNNs, including statistics to monitor the contrastive divergence algorithm, batch sizes, monitoring overfitting (iterations), learning rate ϵ , initial weights, number of hidden unit and hidden layers, types of units (e.g. binary or Gaussian), dropout, among others [16, 11, 4, 6]. In practice, we can only focus on the following:

- Maximum of iterations (**MaxIter**), which is also know as k for the contrastive divergence algorithm.
- The learning rate ϵ (**StepRatio**).
- Type units (e.g. Bernoulli or Gaussian distribution).

We analyze the impact of these DNN parameters through the XOR example; see §5.2.4 below.

5 Further Examples

Besides the MNIST database example, described above, this section presents examples for unsupervised and supervised learning; including prediction and

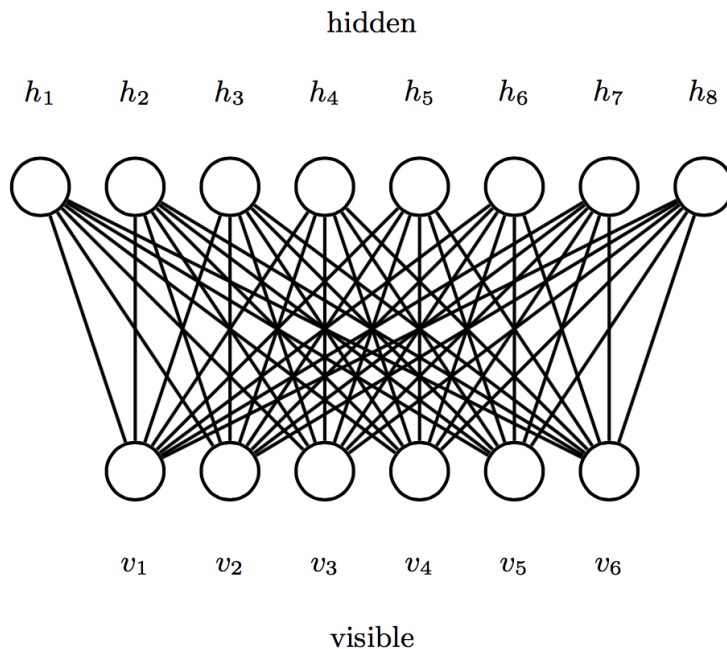


Figure 11: Example of unsupervised learning with a single RBM.

classification tasks.

5.1 Unsupervised learning

We now show an example for unsupervised learning. The network architecture is a single RBM with six visible units and eight hidden units; see Fig. 11. The goal is to learn a simple pattern (**Pattern**) as shown below within the script in Fig. 12. This **Pattern** is a very simple example of unlabeled data.

All the computational times reported throughout this section were obtained running on a personal computer with the following characteristics: 2.3 GHz Intel Core i7 and 4GB memory; Linux-Ubuntu 12.04 and GNU Octave 3.2.4.

5.1.1 Script

Fig. 12 shows the Matlab/Octave script for our example of unsupervised learning. We used the same toolbox than for the MNIST database [24].

5.1.2 Results

As the script in Fig. 12 is executed, several data is displayed. First, the training data (our pattern) is shown:


```

% Matlab/Octave script
clear all;
addpath('..');
ini = clock;
Pattern = [1  1  0  0  0  0 ;
           0  0  1  1  0  0 ;
           0  0  0  0  1  1 ];
Inputs = 6;          % #no variables as input
TrainData = Pattern(:,1:Inputs)
TestData = TrainData;
nodes = [Inputs 8]; % [#inputs #hidden]
bbdbn = randDBN( nodes, 'BBDBN' ); % Bernoulli-Bernoulli RBMs
nrbm = numel(bbdbn.rbm);
% meta-paramters or hyper-parameters
opts.MaxIter = 50;
opts.BatchSize = 1;
opts.Verbose = false;
opts.StepRatio = 2.5;
opts.object = 'CrossEntorpy';
%Learning stage
fprintf( 'Training...\n' );
opts.Layer = nrbm-1;
bbdbn = pretrainDBN(bbdbn, TrainData, opts);
%Testing stage
fprintf( 'Testing...\n' );
H = v2h( bbdbn, TestData); % visible layer to hidden layer
out = h2v(bbdbn,H); % hidden to visible layers (reconstruction)
fprintf( 'Results...\n' );
out
round(out)
theend = clock;
fprintf('\nElapsed time: %f\n',etime(theend,ini));

```

Figure 12: Matlab/Octave script: unsupervised learning example.

```
TrainData =
    1    1    0    0    0    0
    0    0    1    1    0    0
    0    0    0    0    1    1
```

then the output after `pretrainDBN` is:

```
out =
    9.9e-01    9.9e-01    1.0e-04    1.2e-04    2.5e-05    2.6e-05
    5.0e-04    4.3e-04    9.9e-01    9.9e-01    6.1e-04    6.0e-04
    4.2e-06    3.6e-06    5.9e-06    5.4e-06    9.9e-01    9.9e-01
```

This output are the probabilities $[0,1]$, known as reconstructions, so we apply the function `round`, and then we obtain:

```
out =
    1    1    0    0    0    0
    0    0    1    1    0    0
    0    0    0    0    1    1
```

The total elapsed time is 0.254730 seconds.

5.1.3 Discussion

We found that the DNN in Fig. 11 is able to learn the given pattern in an unsupervised manner. We use a `stepRatio` of 2.5 because this allow us to have fewer iterations, i.e. `MaxIter` = 50. Some authors recommend a learning rate (`stepRatio`) of 0.01 [11, 24], but with this setting, we need at least 1,000 iterations to learn the pattern; see §5.2.4 for parameter setting issues.

5.2 Predicting Patterns

The following example simulates a time series prediction scenario. We test two different patterns (`Pattern1` and `Pattern2`). Our training data is a matrix with eight columns, which is the number of variables. We use only six variables as input, and the last two column variables as output. The main idea is that we feed the network only with six variables, then the network must “predict” the next two variables; see Fig. 13. Compared with the previous example, we use supervised learning as shown above in the MNIST example, see §3. Therefore, the labels are our two last columns of the pattern (outputs), i.e. `TrainLabels`.

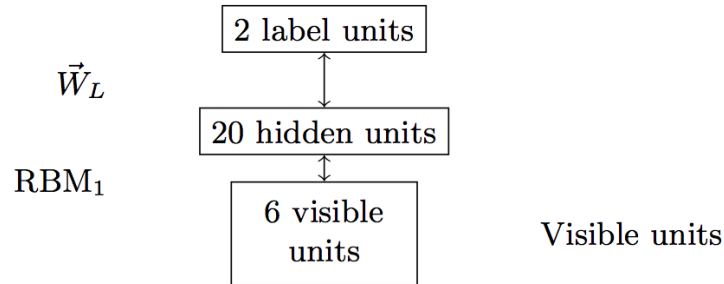


Figure 13: A DNN architecture for the example of supervised learning: predicting patterns.

5.2.1 Script

The script for this example is shown in Fig. 13.

5.2.2 Results

The script in Fig. 14 generates the following output. First, it prints out the `TrainData` and `TrainLabels` together with the instruction `[TrainData TrainLabels]:`

```
Training...
ans =
    0    0    0    0    0    1    0    1
    0    0    0    0    1    0    1    0
    0    0    0    1    0    1    0    0
    0    0    1    0    1    0    0    0
    0    1    0    1    0    0    0    0
    1    0    1    0    0    0    0    0
```

The reconstructions (probabilities) are given in the last two columns:

```
Testing...
ans =
0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0031 0.9881
0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 0.9911 0.0011
0.0000 0.0000 0.0000 1.0000 0.0000 1.0000 0.0044 0.0112
0.0000 0.0000 1.0000 0.0000 1.0000 0.0000 0.0092 0.0073
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0065 0.0002
1.0000 0.0000 1.0000 0.0000 0.0000 0.0000 0.0003 0.0041
```

```

% Matlab/Octave script
clear all;
addpath('..');
ini = clock();
Pattern1 = [ 0 0 0 0 0 1 0 1;
             0 0 0 0 1 0 1 0;
             0 0 0 1 0 1 0 0;
             0 0 1 0 1 0 0 0;
             0 1 0 1 0 0 0 0;
             1 0 1 0 0 0 0 0];

Pattern2 = [ 1 1 0 0 0 0 0 0;
             0 1 1 0 0 0 0 0;
             0 0 1 1 0 0 0 0;
             0 0 0 1 1 0 0 0;
             0 0 0 0 1 1 0 0;
             0 0 0 0 0 1 1 0;
             0 0 0 0 0 0 1 1;
             1 0 0 0 0 0 0 1];

Pattern=Pattern1; % set the pattern to simulate
Inputs = 6;      % #no variables as input
Outputs = 2;     % #no. variables as outputs
TrainData = Pattern(:,1:Inputs); % get the training data, inputs
TrainLabels = Pattern(:,Inputs+1:Inputs+Outputs); % show the labels
TestData = TrainData; % we test with the same training data
TestLabels = TrainLabels;
nodes = [Inputs 20 Outputs]; % [#inputs #hidden #outputs]
bbdbn = randDBN( nodes, 'BBDBN' ); % Bernoulli-Bernoulli RBMs
nrbm = numel(bbdbn.rbm);
% meta-parameters or hyper-parameters
opts.MaxIter = 1000;
opts.BatchSize = 6;
opts.Verbose = false;
opts.StepRatio = 2.5;
opts.object = 'CrossEntorpy';
%Learning stage
fprintf( 'Training...\n' );
[TrainData TrainLabels]
opts.Layer = nrbm-1;
bbdbn = pretrainDBN(bbdbn, TrainData, opts);
bbdbn= SetLinearMapping(bbdbn, TrainData, TrainLabels);
opts.Layer = 0;
bbdbn = trainDBN(bbdbn, TrainData, TrainLabels, opts);
fprintf( 'Testing...\n' );
out = v2h( bbdbn, TestData );
[TestData out]
[TestData round(out)]
theend = clock();
fprintf('\nElapsed time: %f\n',etime(theend,ini));

```

Figure 14: Matlab/Octave script: predicting patterns example.

As in the previous example, we apply the function `round`, so we obtain:

```
[TestData round(out)]
ans =
    0    0    0    0    0    1    0    1
    0    0    0    0    1    0    1    0
    0    0    0    1    0    1    0    0
    0    0    1    0    1    0    0    0
    0    1    0    1    0    0    0    0
    1    0    1    0    0    0    0    0
```

5.2.3 Discussion

In this example, we set a DNN to predict two variables given six input variables. We found that the DNN is able to successfully predict the given patterns. Here, we show only results for `Pattern1`, but the results for `Pattern2` are similar. We started to test the DNN with a different pattern, and we came across accidentally with a pattern that the DNN cannot predict (4 inputs, 2 outputs):

```
Training...
ans =
    0    0    0    0    0    1
    0    0    0    0    1    0
    0    0    0    1    0    0
    0    0    1    0    0    0
    0    1    0    0    0    0
    1    0    0    0    0    0

Testing...
ans =
    0.00000    0.00000    0.00000    0.00000    0.49922    0.49922
    0.00000    0.00000    0.00000    0.00000    0.49922    0.49922
    0.00000    0.00000    0.00000    1.00000    0.00993    0.00993
    0.00000    0.00000    1.00000    0.00000    0.01070    0.01070
    0.00000    1.00000    0.00000    0.00000    0.01142    0.01142
    1.00000    0.00000    0.00000    0.00000    0.01109    0.01109
```

The two first rows of the training data have the same input values, but they have different output. Therefore, the DNN “intelligently” suggest an output of 0.499 (probability).

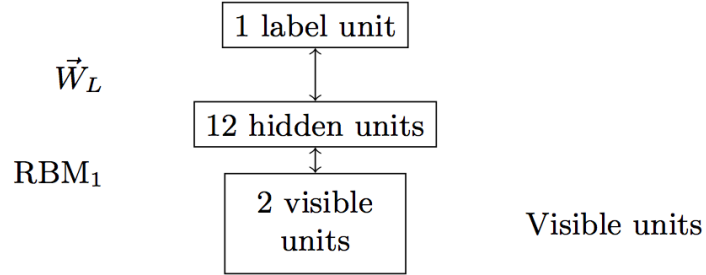


Figure 15: A DNN architecture for the XOR example.

5.2.4 XOR problem

The XOR problem is a non-linear problem that is typical test for a classifier because it is a problem that a simple linear classifier cannot learn. In the neural networks literature, an example of a linear classifier is the *perceptron* introduced by Frank Rosenblatt in 1957 [20]. A decade later, Marvin Minsky and Seymour Paper wrote their famous book *Perceptrons*, and they showed that perceptrons cannot solve the XOR problem [18]. Perhaps partly due to the publication of *Perceptrons*, there was a decline of research in neural networks until the backpropagation algorithm appeared about twenty year after Minsky and Paper's publication.

Here, we analyze the XOR problem with a DNN; see Fig. 15.

5.2.5 Script

The script for the XOR problem is shown in Fig. 16.

```

%Matlab/Octave script
TrainData    = [0 0; 0 1; 1 0; 1 1];
TrainLabels  = [0; 1; 1; 0];
TestData     = TrainData;
TestLabels   = TrainLabels;
nodes        = [2 12 1]; % [#inputs #hidden #outputs]
bbdbn        = randDBN( nodes, 'BBDBN' ); % Bernoulli-Bernoulli RBMs
nrbm         = numel(bbdbn.rbm);
% meta-paramters or hyper-parameters
opts.MaxIter = 100;
opts.BatchSize = 4;
opts.Verbose = false;
opts.StepRatio = 2.5;
opts.object = 'CrossEntorpy';
%Learning stage
fprintf( 'Training...\n' );
opts.Layer = nrbm-1;
bbdbn = pretrainDBN(bbdbn, TrainData, opts);
bbdbn= SetLinearMapping(bbdbn, TrainData, TrainLabels);
opts.Layer = 0;
bbdbn = trainDBN(bbdbn, TrainData, TrainLabels, opts);
%Testing stage
fprintf( 'Testing...\n' );
TestData = [0.1 0.1; 0 0.9; 1 0.2; 0.8 1];
TestData
out = v2h( bbdbn, TestData )
%Printing results
fprintf( '\nResults:\n\n' );
rmse= CalcRmse(bbdbn, TestData, TestLabels);
ErrorRate= CalcErrorRate(bbdbn, TestData, TestLabels);
fprintf( 'For test data:\n' );
fprintf( 'rmse: %g\n', rmse );
fprintf( 'ErrorRate: %g\n', ErrorRate );

```

Figure 16: Matlab/Octave script: XOR example.

5.2.6 Results

The script in Fig. 16 involves only two input variables and a single output, so the `TrainData` and `TrainLabels` are as follows:

```

TrainData =
    0    0
    0    1
    1    0
    1    1

```

```

TrainLabels =
    0
    1
    1
    0

```

Before coming up with the script of Fig. 16, we tested different configurations for the DNN. We started with the following hyperparameter setting:

```

nodes = [2 3 3 1]; % [#inputs #hidden #hidden #outputs]
pts.MaxIter = 10;
opts.BatchSize = 4;
opts.Verbose = true;
opts.StepRatio = 0.1;
opts.object = 'CrossEntropy';
TestData = TrainData;

```

The `TrainLabels` is a column vector, and `out'` is a row vector, where `out'` is the transpose of `out`. Therefore, the desired output is `out' = 0 1 1 0`. The output of the DNN for this setting is:

```
out' = 0.49994    0.49991    0.50009    0.50006
```

This setting does not work with the above parameters. If we add more iterations `opts.MaxIter = 100`, it still does not work properly, and we obtain the output:

```
out' = 0.47035    0.51976    0.49161    0.50654
```

If we add more hidden neurons `nodes = [2 12 12 1]` with the same number of iterations, then the performance improves. The output now is:

```
out' = 0.118510    0.906046    0.878771    0.096262
```

The performance is still better if we add more iterations `opts.MaxIter = 1000`. The output now is:

```
out' = 0.014325    0.982409    0.990972    0.012630
Elapsed time: 32.607048 seconds
```


The previous experiment takes about 33 seconds. In order to decrease the complexity of the network, we reduce the number of hidden neurons. Now we have a single hidden layer `nodes = [2 12 1]`, and with the same number of iterations `opts.MaxIter = 1000`. In about 24 seconds, the output is:

```
out' = 0.043396    0.950205    0.947391    0.059305
Elapsed time: 23.640984 seconds
```

Now, if we reduce the the number of iterations with less neurons as the previous experiments, i.e. `opts.MaxIter = 100`, it is faster but the performance decays. So the output now is:

```
out' = 0.16363    0.80535    0.82647    0.20440
Elapsed time: 2.617439 seconds
```

Other hyperparameter is `opts.StepRatio`, the learning rate, so we tunned this parameter to `opts.StepRatio = 0.01`, and the number of iterations is `opts.MaxIter = 1000`. We found similar results to the previous experiment, i.e. we do not reach the desired output.

Nevertheless, if we use `opts.StepRatio = 2.5` and `opts.MaxIter = 100`, then we obtain a good performance in about one second. The output is:

```
out' = 0.022711    0.955236    0.955606    0.065202
Elapsed time: 0.806343
```

Another important experiment is to test the performance with real values. So we change the test data, and we obtain the following results:

```
TestData =
    0.10000    0.10000
    0.00000    0.90000
    1.00000    0.20000
    0.80000    1.00000
```

```
out =
    0.213813
    0.956192
    0.889679
    0.053432
```

```
Elapsed time: 0.686760
```

Finally, we ran some experiments by tuning the hyperparameter `opts.object` to 'Square' or 'CrossEntorpy', and we found no difference in performance. The flag `opts.verbose` is only for showing or not the training performance.

5.2.7 Discussion

For the XOR problem, we found that the best performance is with the following combination: `opts.StepRatio = 2.5` and `opts.MaxIter = 100`. A large step ratio with few iterations allow us to obtain results faster than other settings. The performance is good enough to solve the XOR problem. Moreover, this setting classifies correctly when the inputs are real data. For this reason, this setting is used in the previous examples; see §5.1 and §5.2.

6 Speech Processing

Speech Processing has several applications including Speech Recognition, Language Identification and Speaker Recognition; see Fig. 17. Sometimes, additional information is stored and associated to speech. Therefore, Speaker Recognition can be either text dependent or text independent. Moreover, Speaker Recognition involves different tasks such as [2]:

- Speaker Identification
- Speaker Detection
- Speaker Verification.

6.1 Speech Features

The first step for most speech recognition systems is the feature extraction from the time-domain sampled acoustic waveform (audio); see Figure 18a. The time-domain waveform is represented by overlapping *frames*. Each frame is generated every 10ms with a duration of 25ms. Then, a feature is extracted for every frame. Several methods have been investigated for feature extraction (acoustic representation) including Linear Prediction Coefficients (LPCs), Perceptual Linear Prediction (PLP) coefficients and Mel-Frequency spaced Cepstral Coefficients(MFCCs) [5, 25].

6.2 DNN and Speech Processing

As we shown above, DNNs have the flexibility to be used as either unsupervised or supervised learning. Therefore, DNNs can be used for regression or classification problems in speech recognition; see Fig. 19.

Nowadays, DNNs have been applied successfully on speech processing including speaker recognition [9, 26], language identification [7] and speech generation [17].

VOICEBOX⁶ is a Speech Processing Toolbox for MATLAB[®], which is also publicly available.

⁶<http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html>

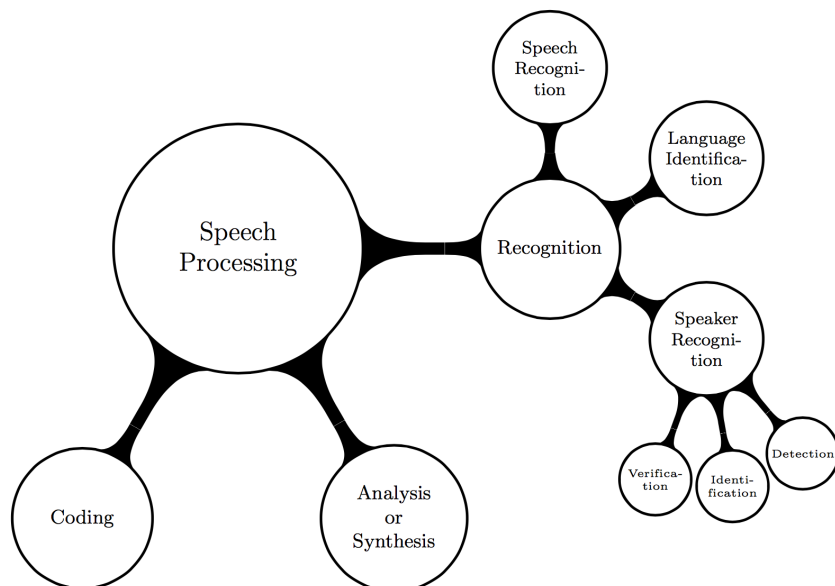


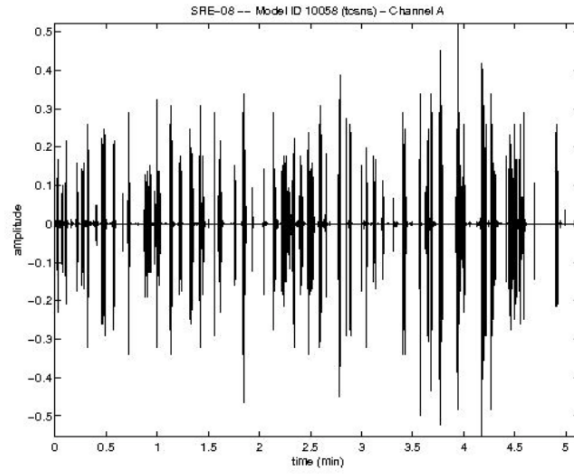
Figure 17: Speech Processing

7 Summary

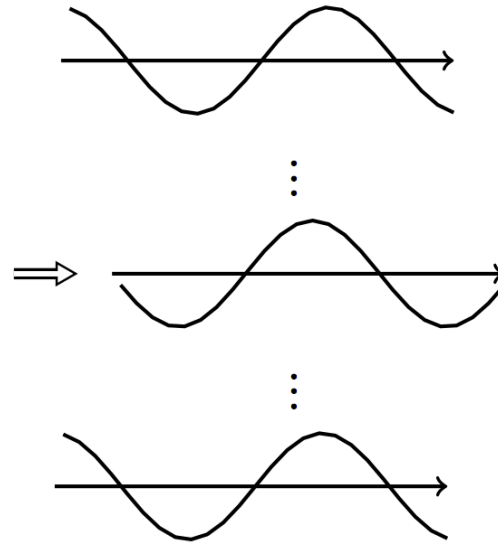
Neural networks approaches have been used widely to build Intelligent Systems. We introduced Deep Neural Networks (DNNs) and Restricted Boltzmann Machines (RBMs), and their relationship to Deep Learning (DL) and Deep Belief Nets (DBNs). Across the literature, there are some introductory papers for RBMs [11, 6]. One of the contributions of this tutorial are the simple examples for a better understanding of RBMs and DNNs. The examples cover unsupervised and supervised learning, therefore, we cover both unlabeled and labeled data, for prediction and classification. Moreover, we introduce a publicly available MATLAB[®] toolbox to show the performance of DNNs and RBMs [24]. The toolbox and the examples have been tested on Octave, the open source version of MATLAB[®]. The last example, XOR problem, presents some results by different setting of some hyperparameters of DNNs. Finally, two applications for intelligent pattern recognition are also covered on this tutorial: the MNIST benchmarking and speech recognition.

References

- [1] C. M. Bishop and G. E. Hinton. *Neural Networks for Pattern Recognition*. Clarendon Press, 1995.



(a) Audio



(b) Features

Figure 18: Audio sample. (a) Audio represented as a time-domain waveform; (b) Features obtained from the time-domain waveform.

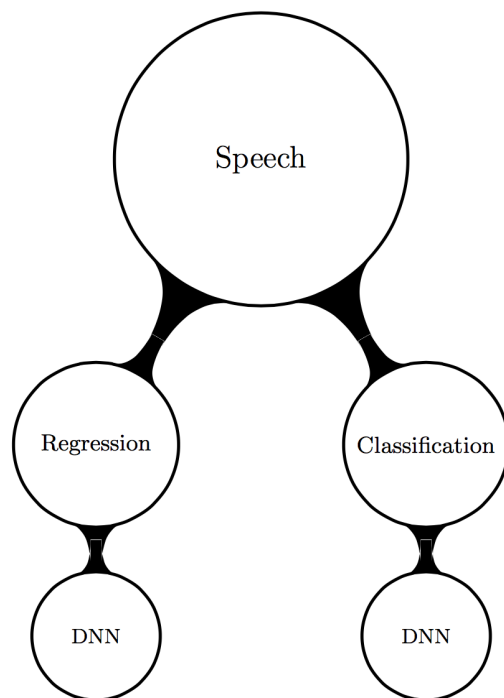


Figure 19: DNN and Speech

- [2] J.P. Jr. Campbell. Speaker recognition: A tutorial. *Proceedings of IEEE*, 85(6):1437–1462, 1997.
- [3] A. Coates, B. Huval, T. Wang, and A. Y. Wu, D. J. and Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-2013)*, 2013.
- [4] G. E. Dahl, T. N. Sainath, and G. E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-2013)*, pages 8609–8613, 2013.
- [5] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, volume 28, 1980.
- [6] A. Fischer and C. Igel. Training restricted Boltzmann machines: An introduction. *Pattern Recognition*, 14:25–39, 2014.
- [7] J. Gonzalez-Dominguez, I. Lopez-Mreno, P.J. Moreno, and J. Gonzalez-Rodriguez. Frame-by-frame language identification in short utterances using deep neural networks. *Neural Networks*, 64:49–58, 2015.
- [8] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.
- [9] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
- [10] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1711–1800, 2002.
- [11] G. E. Hinton. A practical guide to training restricted Boltzmann machines version 1. Department of Computer Science, University of Toronto, 2010.
- [12] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.
- [13] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(1):504–507, 2006.
- [14] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences*, volume 79, pages 2554–2558, 1982.

- [15] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, , and A. Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of International Conference on Machine Learning (ICML2012)*, 2012.
- [16] H. Lee, C. Ekanadham, and A. Y. Ng. Sparse deep belief net model for visual area V2. In *Advances in Neural Information Processing Systems (NIPS-2008)*, volume 20, 2008.
- [17] Z.H. Ling, S.Y. Kang, H. Zen, A. Senior, M. Schuster, X.J. Qian, H. Meng, and L. Deng. Deep learning for acoustic modeling in parametric speech generation. *IEEE Signal Processing Magazine*, 32(3):35–52, 2015.
- [18] M. L. Minsky and S. A. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969.
- [19] R. Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, 1996.
- [20] F. Rosenblatt. The perceptron—A perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.
- [21] D. E. Rumelhart, G. E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(1):533–536, 1986.
- [22] J Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.
- [23] P. Smolensky. *Parallel Distributed Processing*, volume 1, chapter Information Processing in Dynamical Systems: Foundations of Harmony Theory, pages 194–281. MIT Press, Cambridge, 1986.
- [24] M. Tanaka and M. Okutomi. A novel inference of a restricted Boltzmann machine. In *International Conference on Pattern Recognition (ICPR2014)*, 2014.
- [25] R. Togneri and D. Pullella. An overview of speaker identification: Accuracy and robustness issues. *IEEE Circuits and Systems Magazine*, 2(1):23–61, 2011.
- [26] C. Weng, D. Yu, M.L. Seltzer, and J. Droppo. Single-channel mixed speech recognition using deep neural networks. In *IEEE International Acoustics, Speech and Signal Processing (ICASSP)*, pages 5632–5636, 2014.