Chapter Four

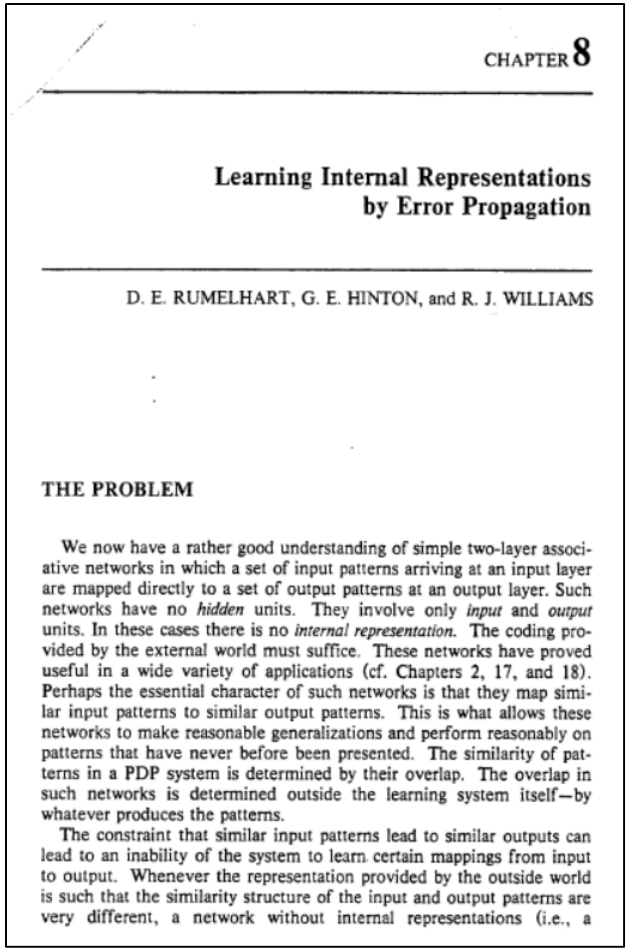
BP: An Illustrating Example

Zhang Yi, *IEEE Fellow*
Autumn 2019

Outline

- Brief Review of Backpropagation Algorithm
- An Illustrating Example
- Experiments
- Assignment

Brief History of BP



CHAPTER 8

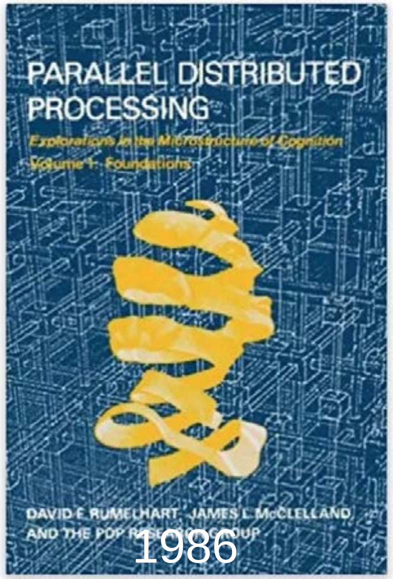
Learning Internal Representations by Error Propagation

D. E. RUMELHART, G. E. HINTON, and R. J. WILLIAMS

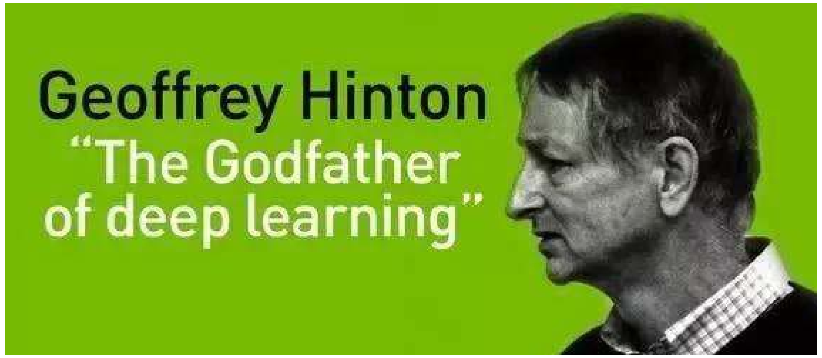
THE PROBLEM

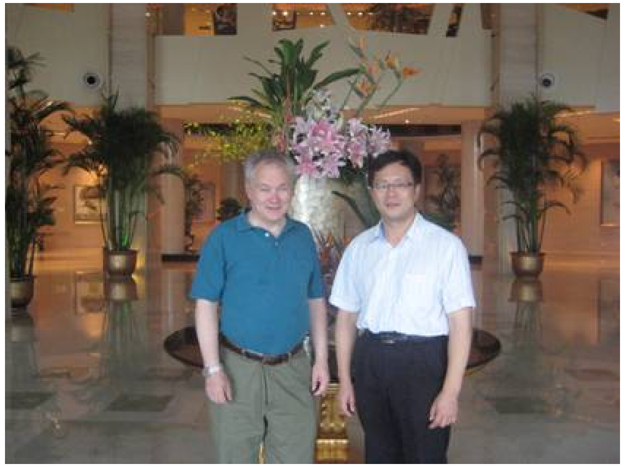
We now have a rather good understanding of simple two-layer associative networks in which a set of input patterns arriving at an input layer are mapped directly to a set of output patterns at an output layer. Such networks have no *hidden* units. They involve only *input* and *output* units. In these cases there is no *internal representation*. The coding provided by the external world must suffice. These networks have proved useful in a wide variety of applications (cf. Chapters 2, 17, and 18). Perhaps the essential character of such networks is that they map similar input patterns to similar output patterns. This is what allows these networks to make reasonable generalizations and perform reasonably on patterns that have never before been presented. The similarity of patterns in a PDP system is determined by their overlap. The overlap in such networks is determined outside the learning system itself—by whatever produces the patterns.

The constraint that similar input patterns lead to similar outputs can lead to an inability of the system to learn certain mappings from input to output. Whenever the representation provided by the outside world is such that the similarity structure of the input and output patterns are very different, a network without internal representations (i.e., a



1986



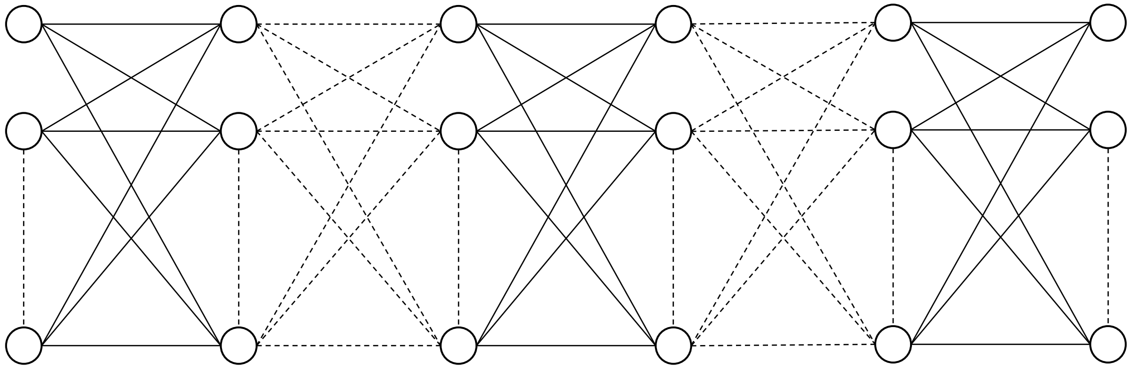


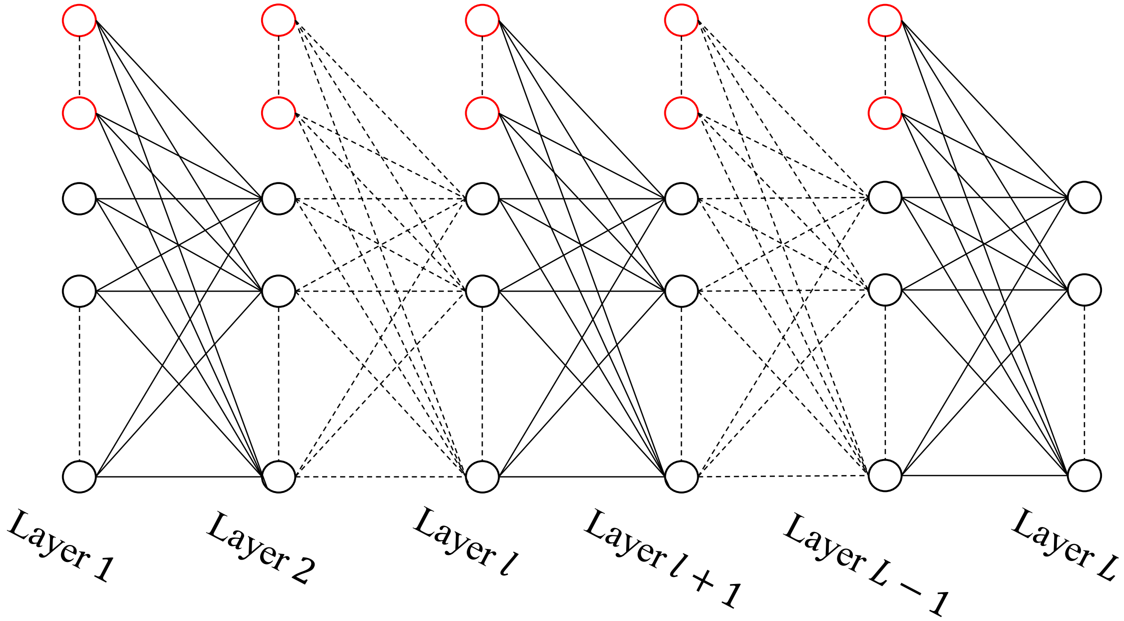
Professor P. Werbos

Computational Model of Neural Networks

Two important characters:

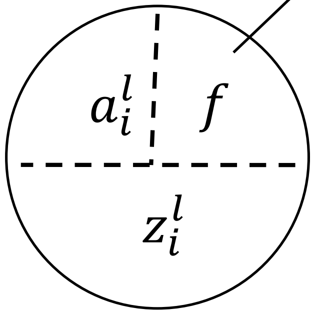
- No any connection in any layer
- No any connection across any layer





External inputs

Local function defined on neuron



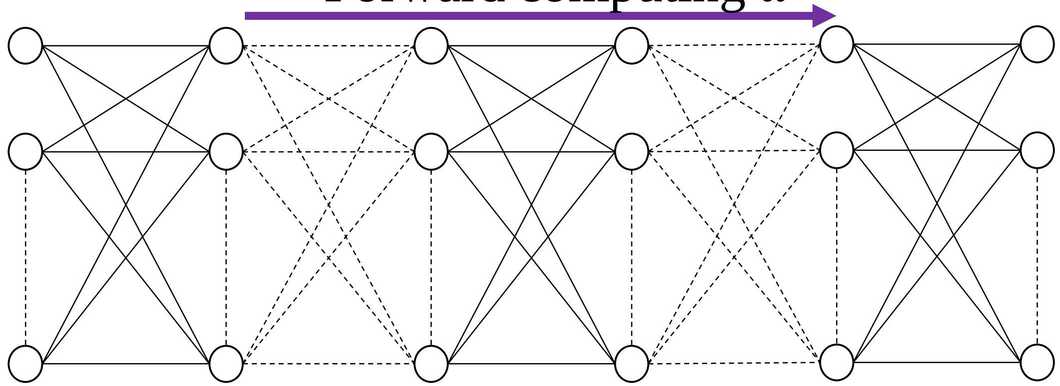
l layer i^{th} neuron

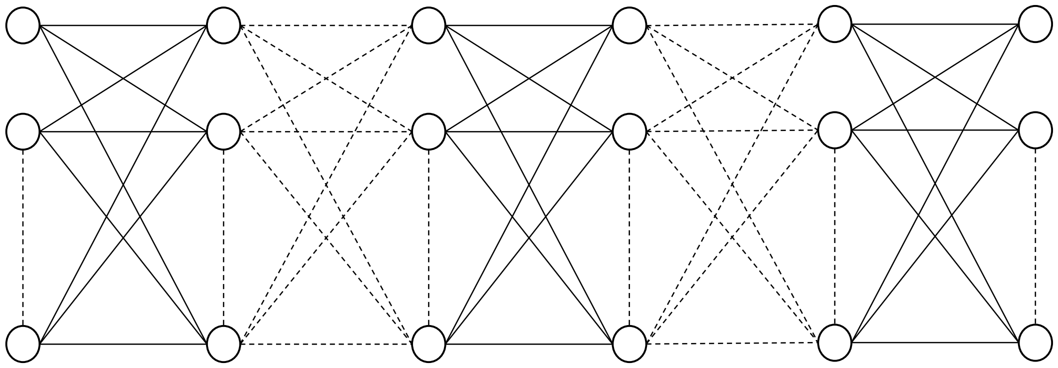
$$a_i^l = f(z_i^l)$$

$$a^l = \begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_{n_l}^l \end{bmatrix}$$

Local activation function f

Forward computing a^l



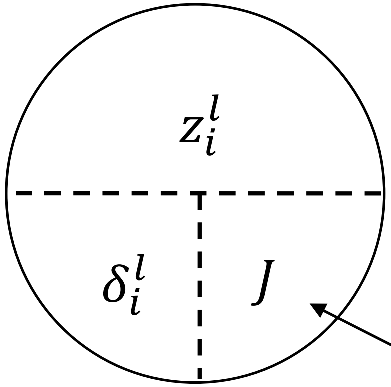


Backpropagation δ^l

Global cost function J

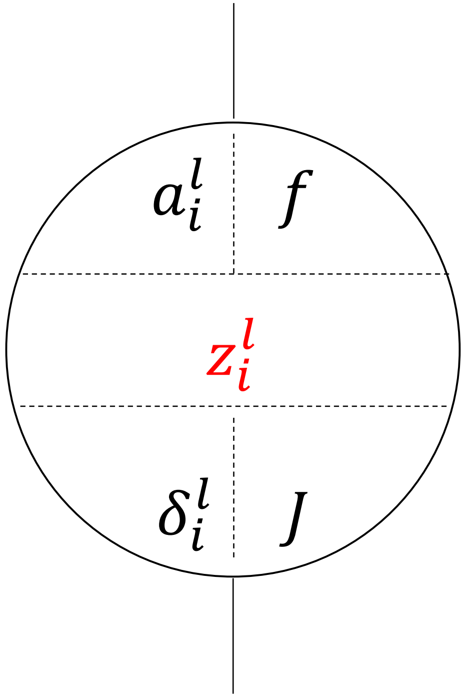
l layer i^{th} neuron

$$\delta_i^l = \frac{\partial J}{\partial z_i^l}$$

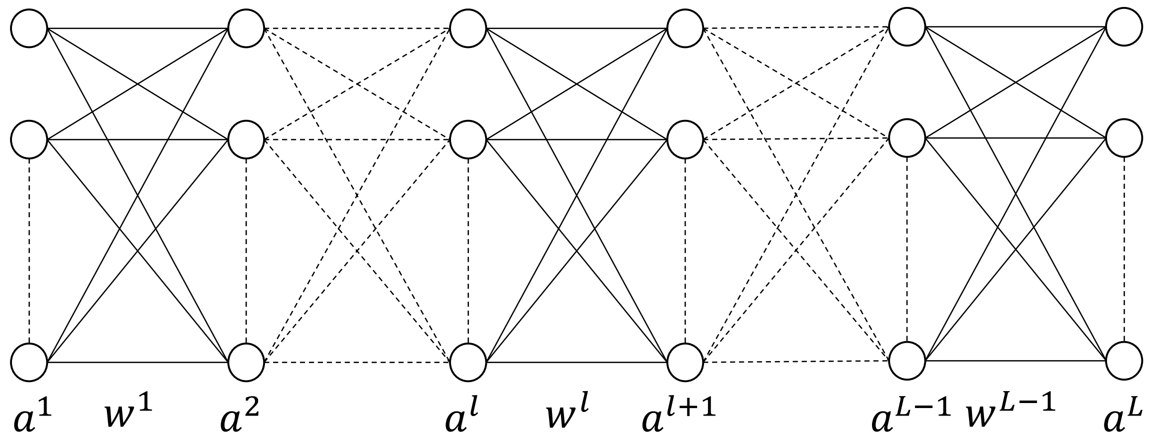


$$\delta^l = \begin{bmatrix} \delta_1^l \\ \delta_2^l \\ \vdots \\ \delta_{n_l}^l \end{bmatrix}$$

Global function defined on network



Network Performance: Cost Function



A cost function J describes the performance of the network. If the J is small, it implies that the network prediction a^L close to the target y^L, the network is called in good performance. Since J is a function with variables $(w^1, \cdots, w^L)$, good performance means to find suitable $(w^1, \cdots, w^L)$ such that J is small. The process of looking for suitable $(w^1, \cdots, w^L)$ is called network learning.

Target

Network prediction

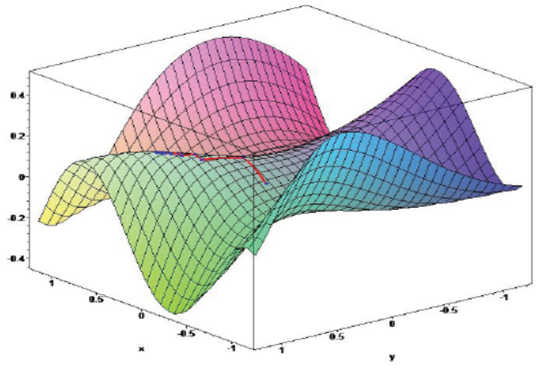
$$y^L = \begin{bmatrix} y_1^L \\ \vdots \\ y_{n_L}^L \end{bmatrix} \qquad a^L = \begin{bmatrix} a_1^L \\ \vdots \\ a_{n_L}^L \end{bmatrix}$$

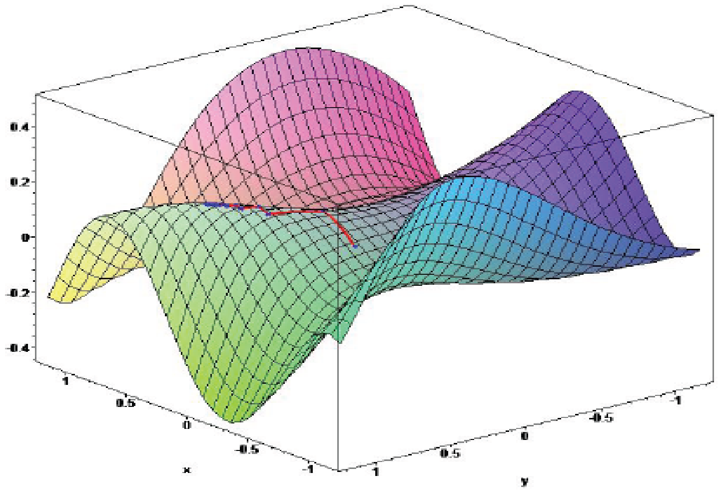
There are many ways to construct cost functions. A frequently used cost is as follows:

$$e_j = a_j^L - y_j^L$$

$$J = \frac{1}{2}\sum_{j=1}^{n_L} e_j^2 = J(w^1, \cdots, w^L)$$

Clearly, J is a function of $w^1, \cdots, w^L$.

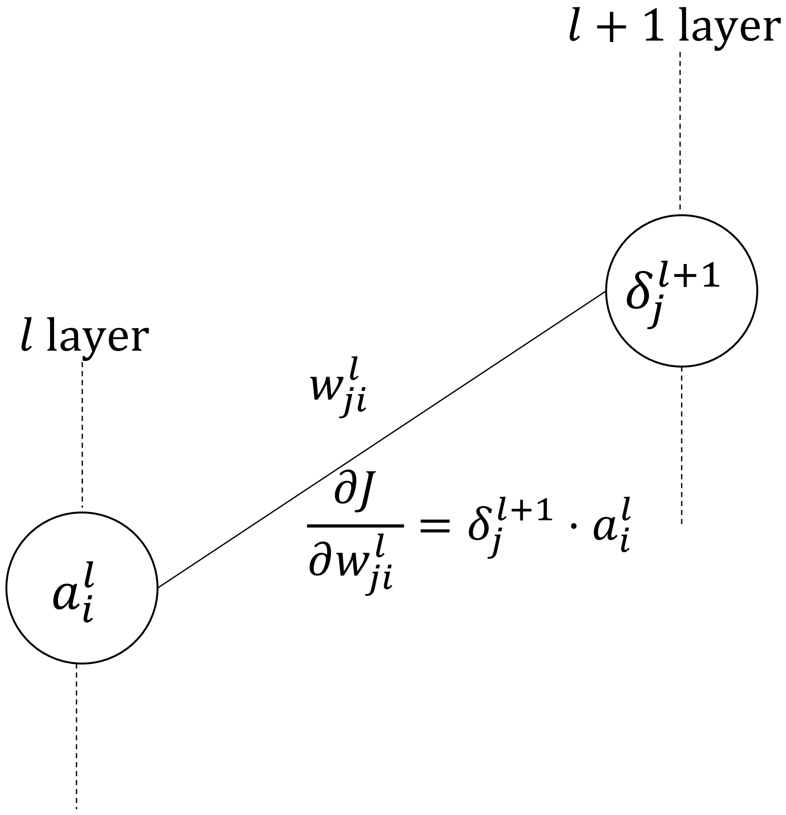




$$J = J(\cdots, w_{ij}^l, \cdots)$$

Steepest Descent Method

$$w_{ji}^l \leftarrow w_{ji}^l - \alpha \cdot \frac{\partial J}{\partial w_{ji}^l}$$

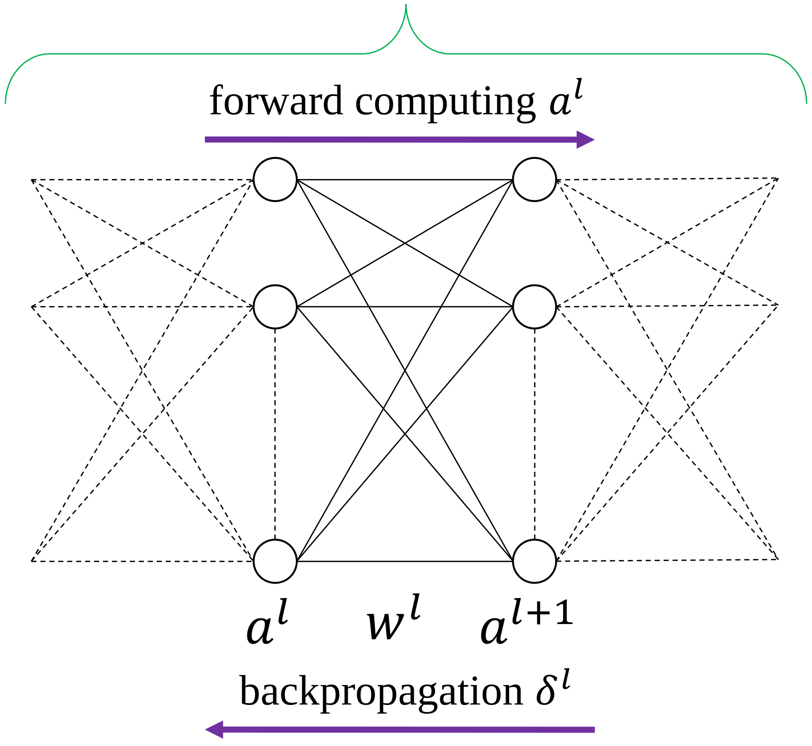


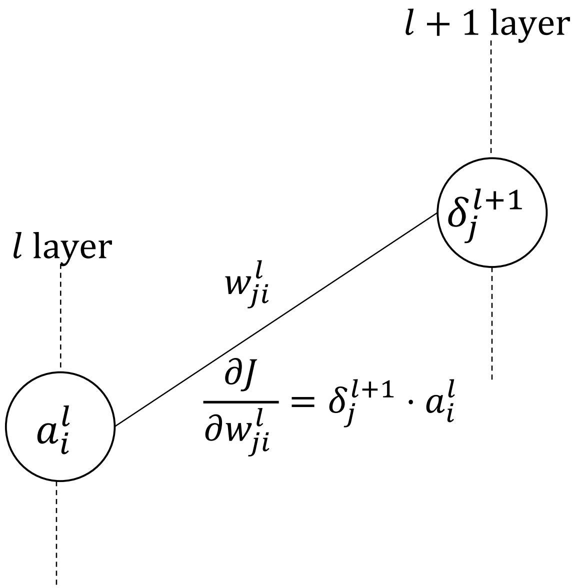
One Page to Understand BP

Cost function: $J(w^1, \cdots, w^L)$

Relationship: $\frac{\partial J}{\partial w_{ji}^l} = \delta_j^{l+1} \cdot a_i^l$

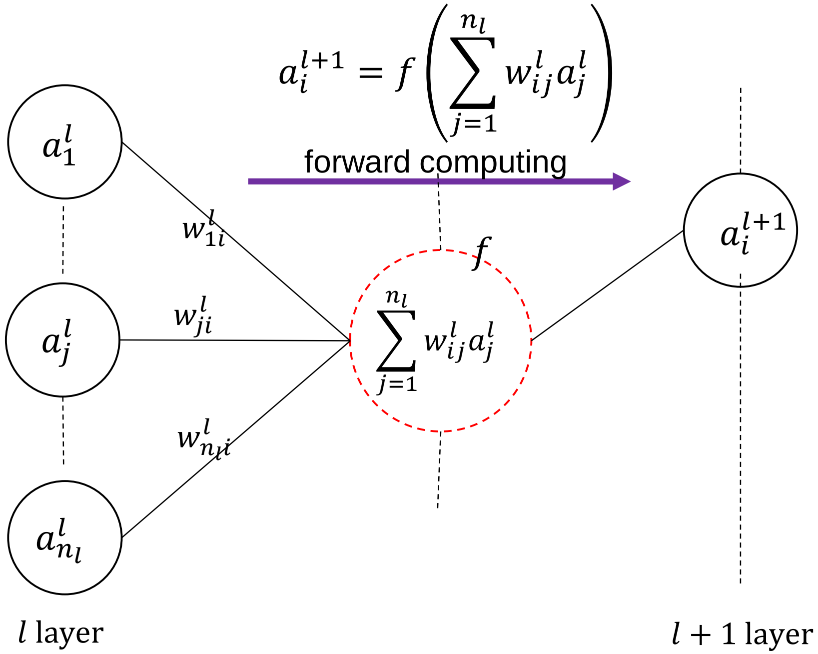
Updating rule: $w_{ji}^l \leftarrow w_{ji}^l - \alpha \cdot \frac{\partial J}{\partial w_{ji}^l}$

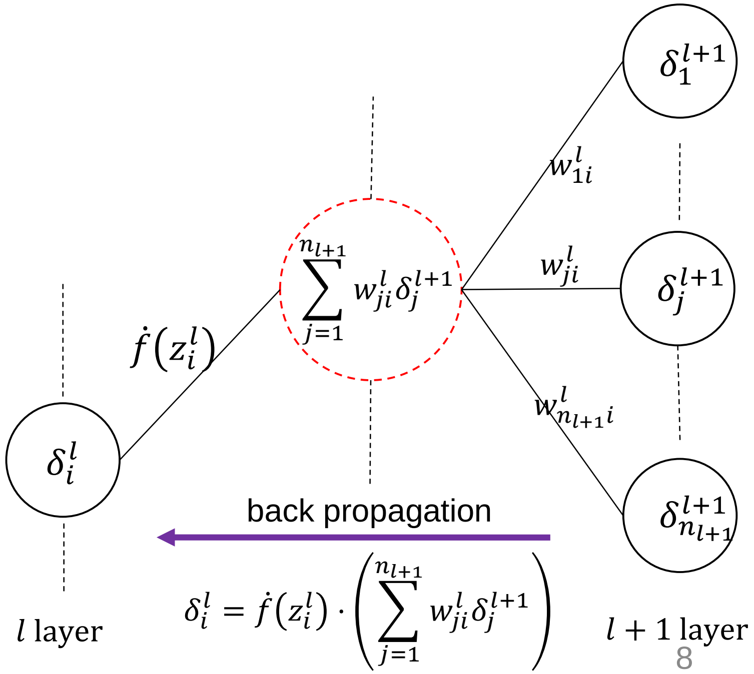




$$a_i^l = f(z_i^l)$$

$$\delta_i^l = \frac{\partial J}{\partial z_i^l}$$



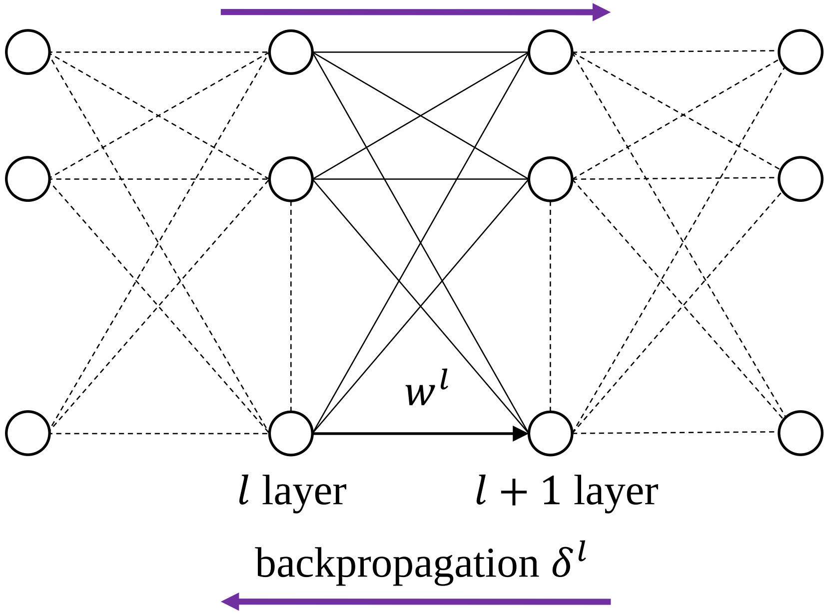


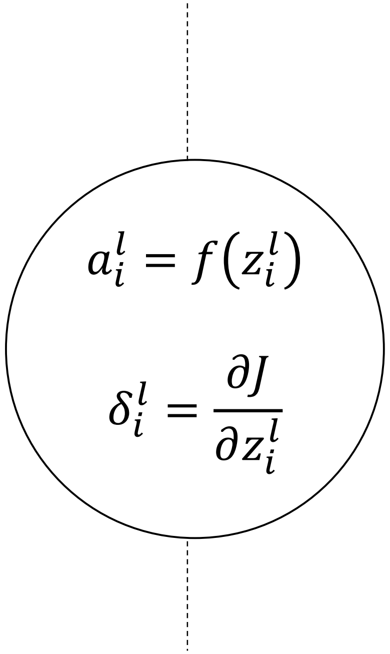
BP Functions

Cost function: $J(w^1, \cdots, w^L)$

Updating rule: $w_{ji}^l \leftarrow w_{ji}^l - \alpha \cdot \frac{\partial J}{\partial w_{ji}^l}$

Relationship: $\frac{\partial J}{\partial w_{ji}^l} = \delta_j^{l+1} \cdot a_i^l$





%forward computing

function $fc(w^l, a^l)$

$for\ i = 1{:}\,n_{l+1}$

$$z_i^{l+1} = \sum_{j=1}^{n_l} w_{ij}^l a_j^l$$

$$a_i^{l+1} = f\left(z_i^{l+1}\right)$$

end

%backpropagation

function $bc(w^l, \delta^{l+1})$

$for\ i = 1{:}\,n_l$

$$\delta_i^l = \dot{f}(z_i^l) \cdot \left(\sum_{j=1}^{n_{l+1}} w_{ji}^l \delta_j^{l+1}\right)$$

end

The BP Algorithm

Step 1. Input the training data set $D = \{(x, y)\}$

Step 2. Initialize each w_{ij}^l, and choose a learning rate α.

Step 3. for each mini-batch sample $D_m \subseteq D$

for each $x \in D_m$

$a^1 \leftarrow x \in D_m$;

for $l = 1 : L\text{-}1$

$a^{l+1} \leftarrow fc(w^l, a^l)$;

end

$$\delta^L = \frac{\partial J(x)}{\partial z^L};$$

for $l = L - 1 : 2$

$\delta^l \leftarrow bc(w^l, \delta^{l+1})$;

end

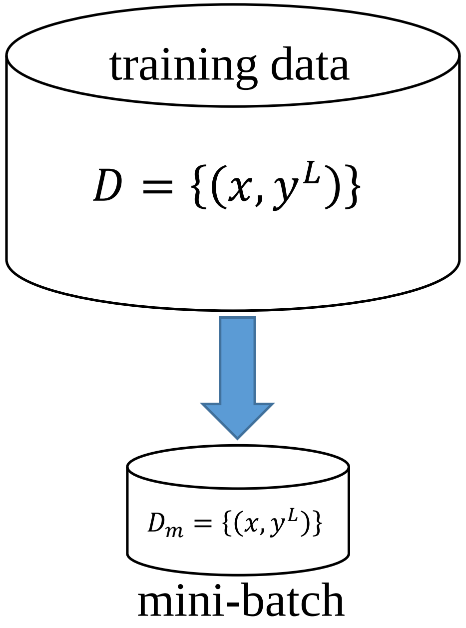
$$\frac{\partial J}{\partial w_{ji}^l} \leftarrow \frac{\partial J}{\partial w_{ji}^l} + \delta_j^{l+1} \cdot a_i^l;$$

end

$$w_{ji}^l \leftarrow w_{ji}^l - \alpha \cdot \frac{\partial J}{\partial w_{ji}^l};$$

end

Step 4. Return to Step 3 until each w^l converge.



function $fc(w^l, a^l)$

$for\ i = 1 : n_{l+1}$

$$z_i^{l+1} = \sum_{j=1}^{n_l} w_{ij}^l a_j^l$$

$$a_i^{l+1} = f(z_i^{l+1})$$

end

Relationship:

$$\frac{\partial J}{\partial w_{ji}^l} = \delta_j^{l+1} \cdot a_i^l$$

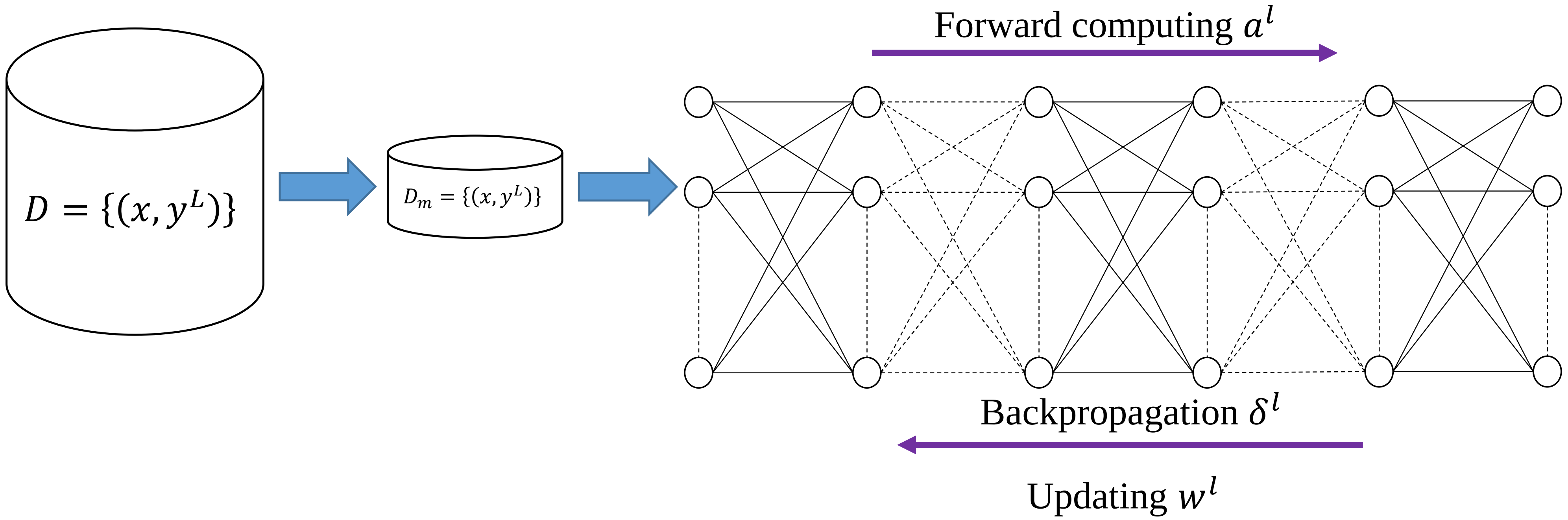
function $bc(w^l, \delta^{l+1})$

$for\ i = 1 : n_l$

$$\delta_i^l = \dot{f}(z_i^l) \cdot \left(\sum_{j=1}^{n_{l+1}} w_{ji}^l \delta_j^{l+1} \right)$$

end

Network Training



Network Training Steps

Step 1: Data Preparation → Step 2: Design Network Architecture → Step 3: Initialize Parameters → Step 4: Define Cost Function → Step 5: Define Evaluation Index

Step 1: Data Preparation

$D_{train} = \{(x, y^L)\}$

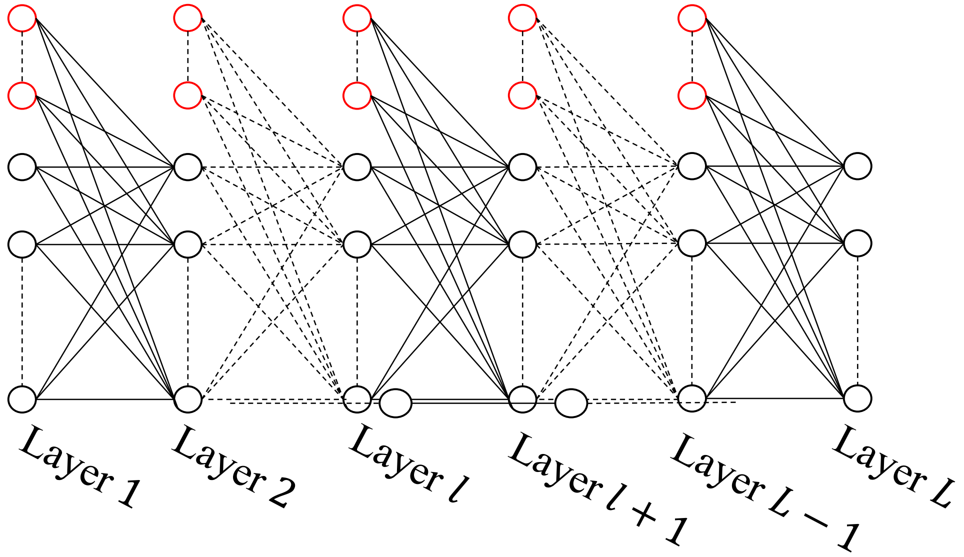
Training set

$D_{test} = \{(x, y^L)\}$

Testing set

Step 2: Design Network Architecture

- Number of layers
- Number of neurons in each layer
- Activation function



Step 3: Initialize Parameters

- Weights
- Learning rate

Step 4: Define Cost Function

$$J = \frac{1}{2}\sum_{j=1}^{n_L} e_j^2$$
$$= J(w^1, \cdots, w^L)$$

Step 5: Define Evaluation Index

Accuracy

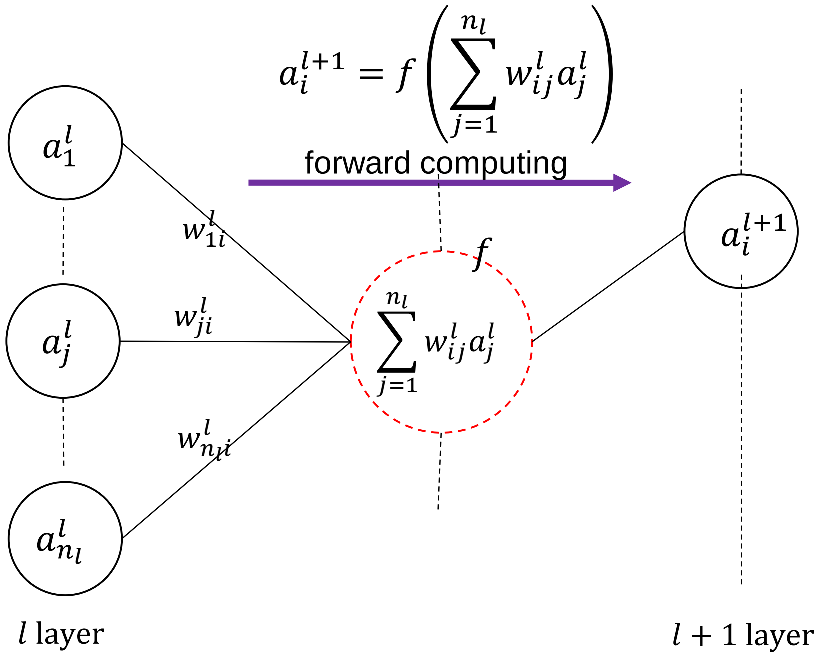
Network Training Steps

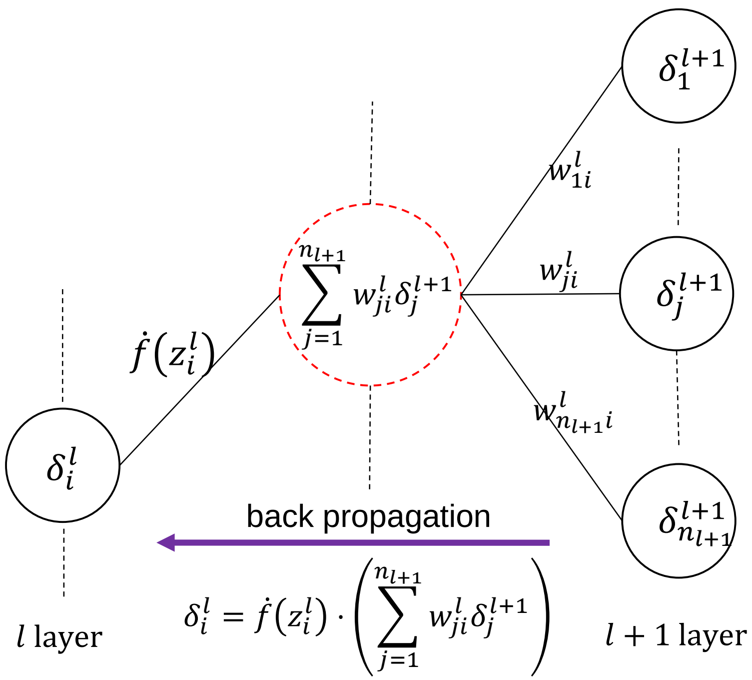
Step 6: Train the Network → Step 7: Test the Network → Step 8: Store the Network Parameters → Step 9: Use Trained Network for Application

- BP Algorithm

- Test on the training set
- Test on the testing set

- Save weights

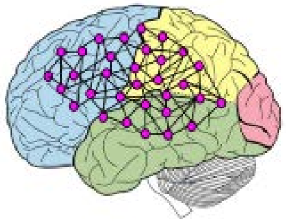




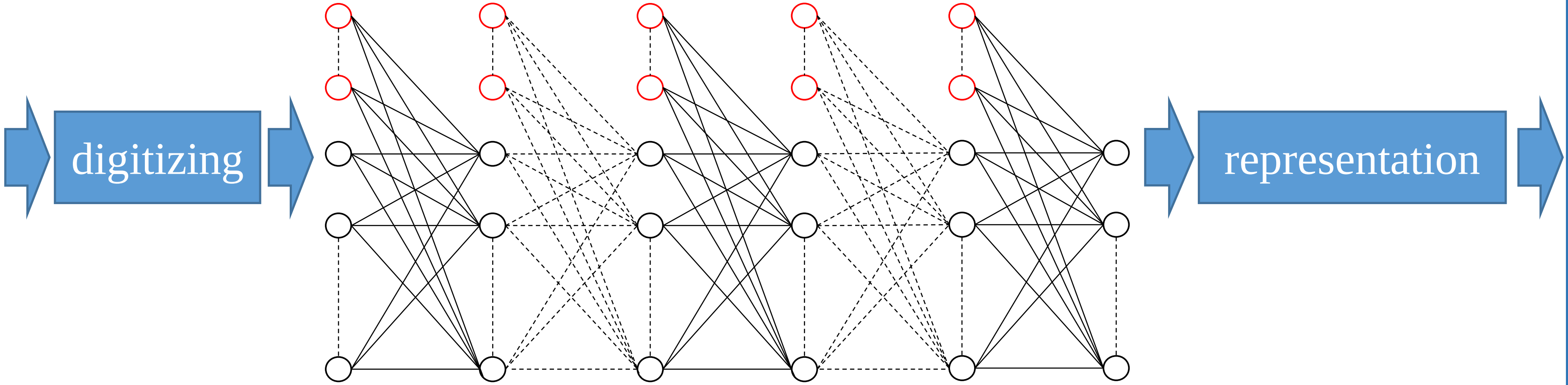
Outline

- Brief Review of Backpropagation Algorithm
- An Illustrating Example
- Experiments
- Assignment

Handwritten digits recognition problem



≈



0
1
2
3
4
5
6
7
8
9

Task:
Use Backpropagation algorithm to train a neural network to recognize handwritten digits.

Step 1: Data Preparation

Dataset: **MNIST_small**

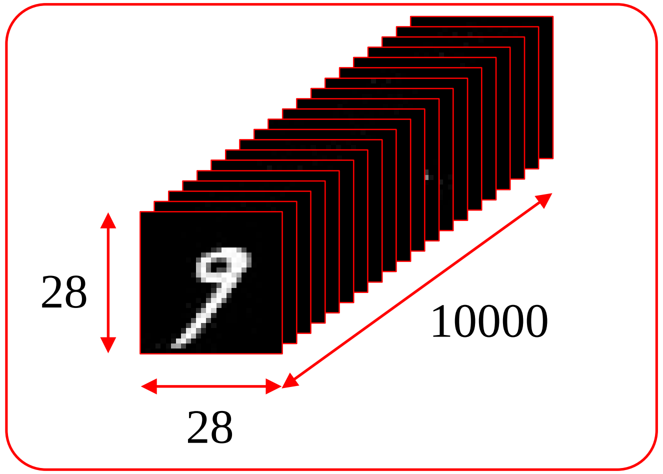
MNIST is a database of handwritten digits created by "re-mixing" the samples from MNIST's original datasets. It contains digits written by high school students and employees of the United States Census Bureau. The digits have been size-normalized and centered in 28 × 28 images.

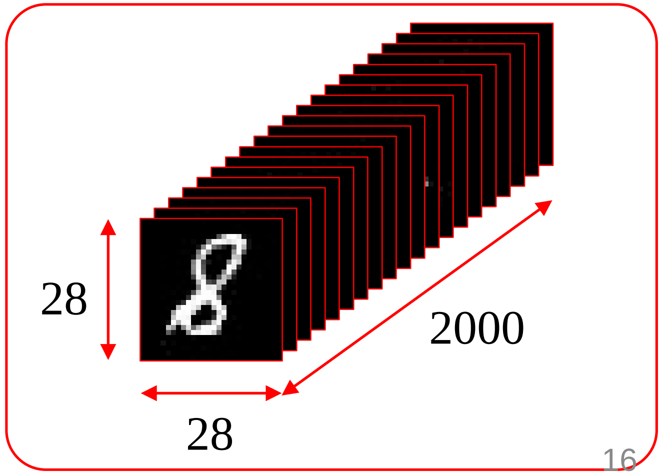
MNIST_small dataset is a subset of MNIST containing 10000 training samples and 2000 testing samples.

mnist_small_matlab.mat

名称	值
trainData	28x28x10000 double
testData	28x28x2000 double
trainLabels	10x10000 double
testLabels	10x2000 double

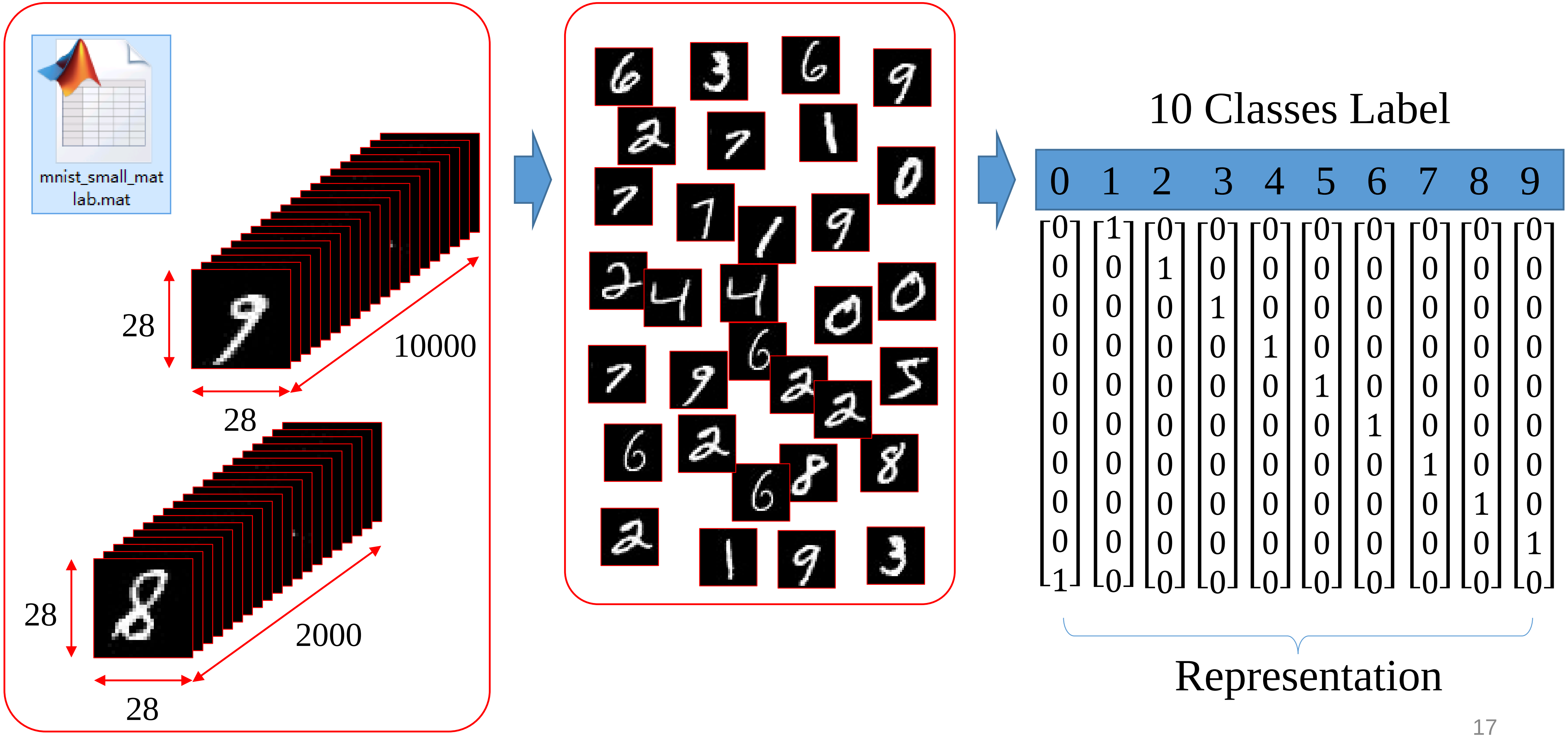
Data





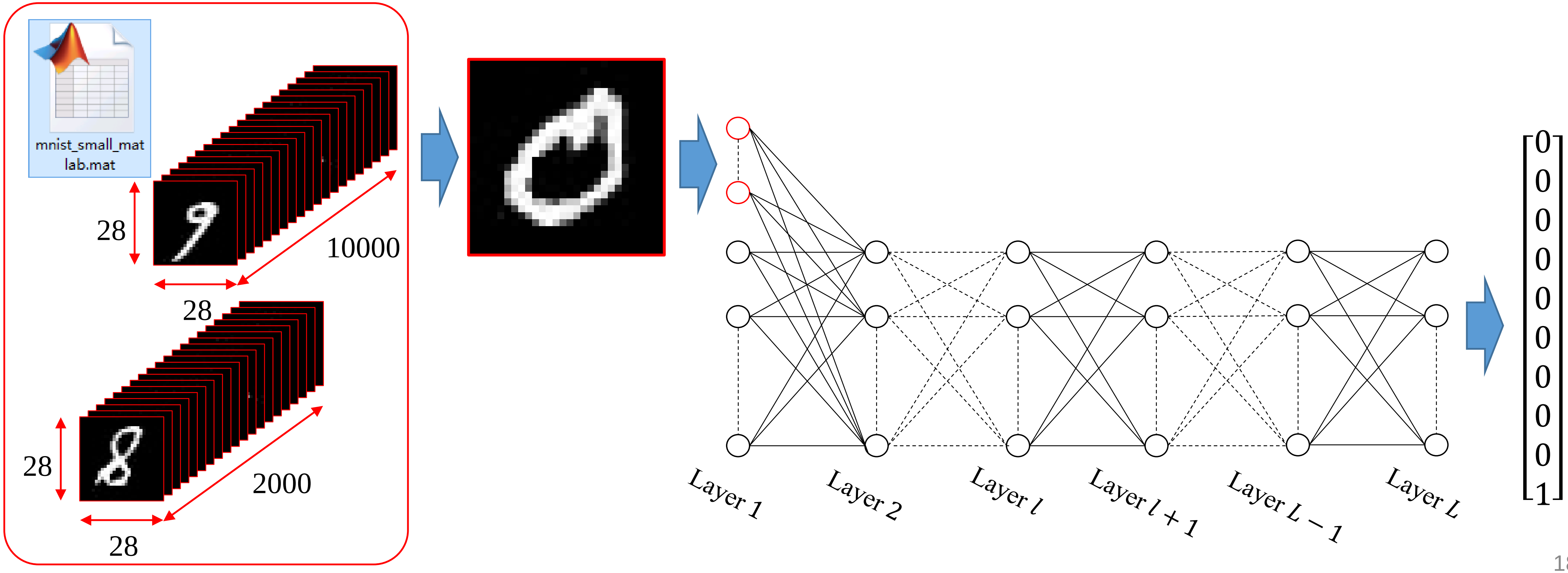
Download link:
MNIST http://yann.lecun.com/exdb/mnist/
MNIST_small: https://github.com/kswersky/nnet/blob/master/mnist_small.mat

Step 1: Data Preparation

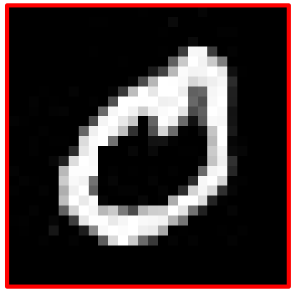


Step 1: Data Preparation

The input image is a $28 \times 28 = 784$ dimensional vector.



Step 1: Data Preparation

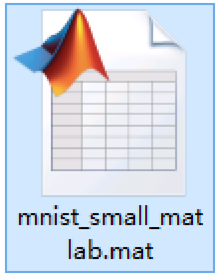


28×28

digitizing

$$\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} 784$$

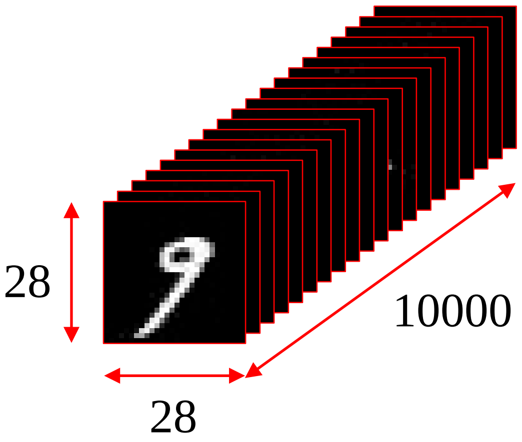
Step 1: Data Preparation



Training set

- Used for training network
- 10000 samples

Training Data

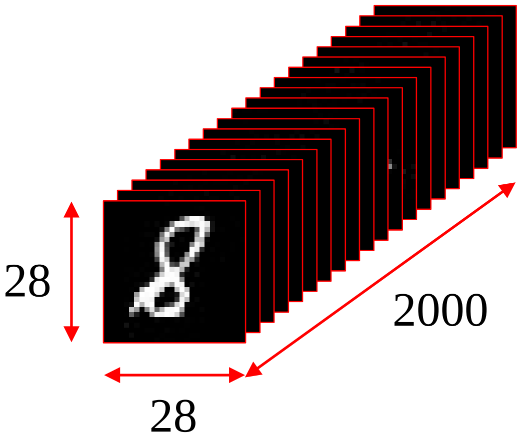


Training Data

$$\begin{bmatrix} 0 & & 1 \\ 1 & \cdots & 0 \\ 1 & & 1 \\ \vdots & \ddots & \vdots \\ 0 & & 0 \\ 1 & \cdots & 0 \\ 0 & & 1 \end{bmatrix}_{784 \times 10000}$$

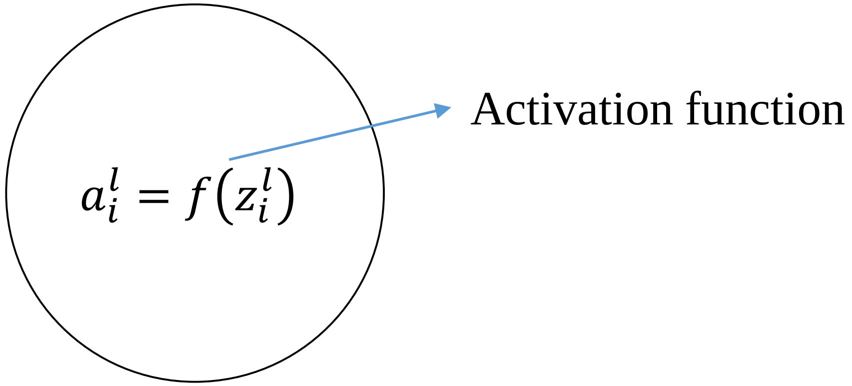
Testing set

- Used for evaluating network performance
- 2000 samples



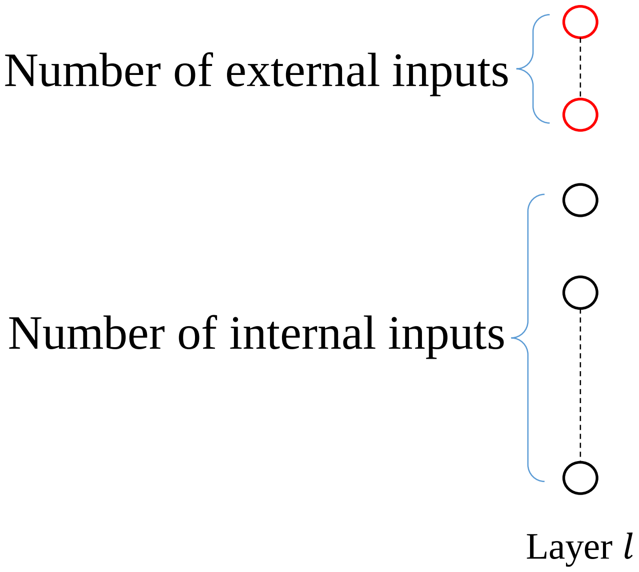
$$\begin{bmatrix} 0 & & 0 \\ 0 & \cdots & 0 \\ 1 & & 1 \\ \vdots & \ddots & \vdots \\ 0 & & 0 \\ 1 & \cdots & 1 \\ 1 & & 1 \end{bmatrix}_{784 \times 2000}$$

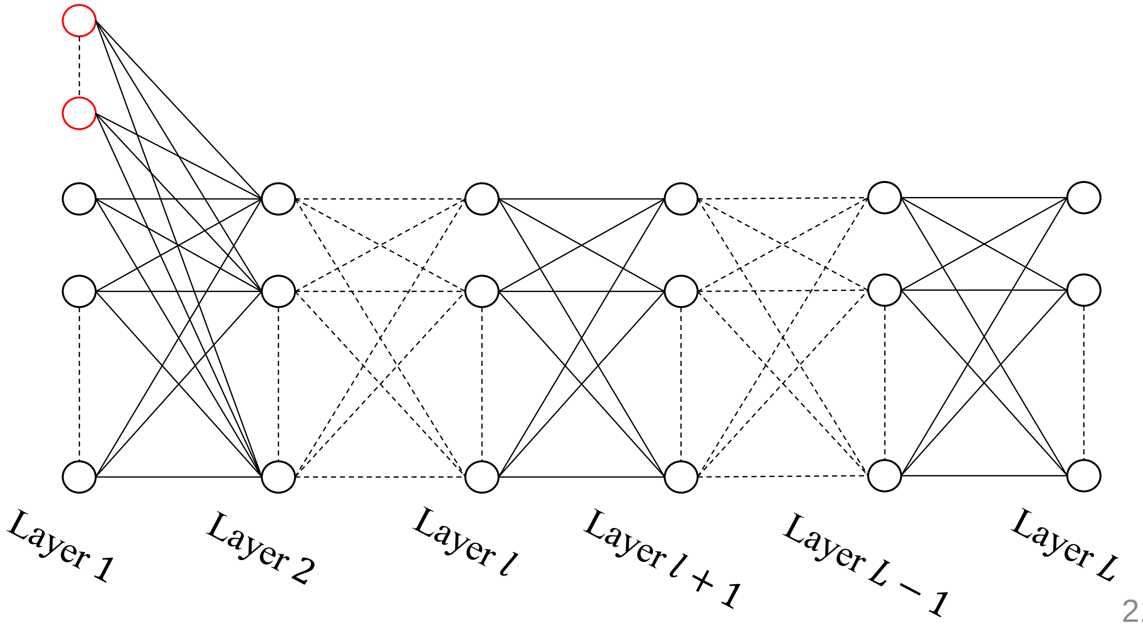
Step 2: Design Network Architecture



Network architecture design:

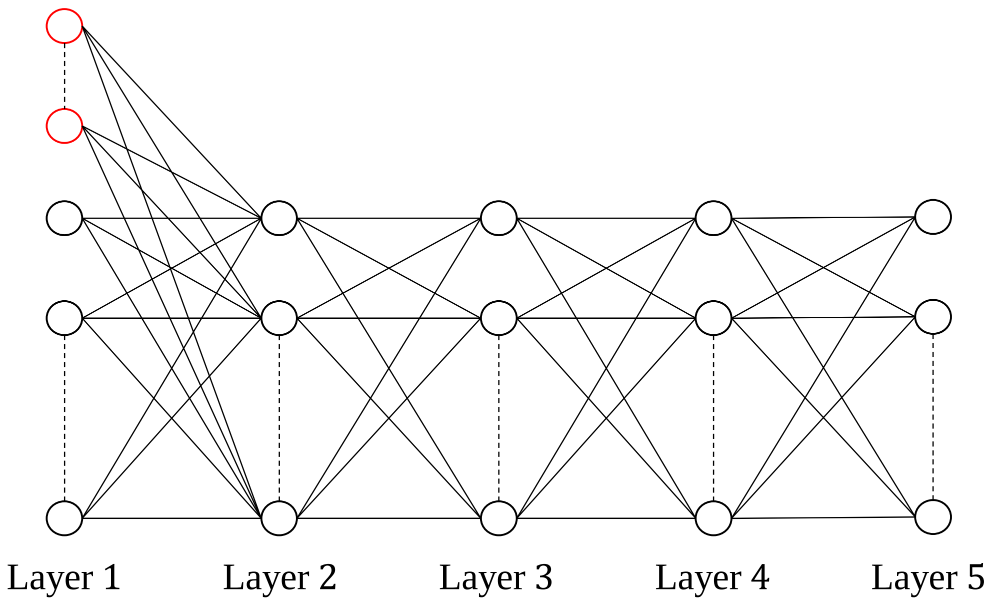
1. Number of layers
2. Number of neurons in each layer (external neurons and internal neurons)
3. Activation function





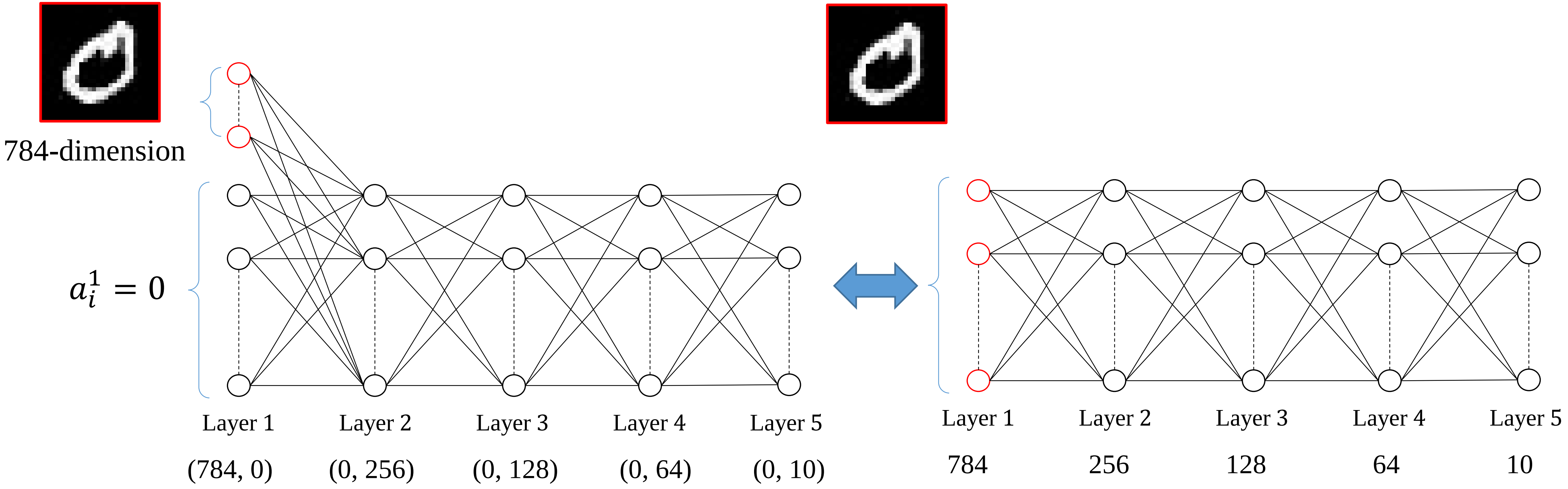
Step 2: Design Network Architecture

Define Number of Layers



Step 2: Design Network Architecture

Define Number of Neurons in Each Layer



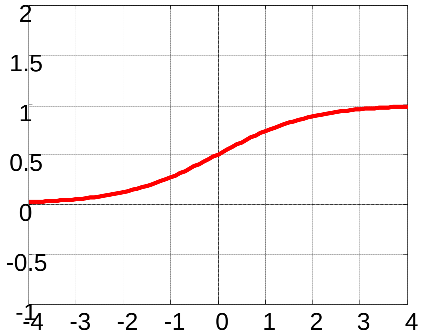
Step 2: Design Network Architecture

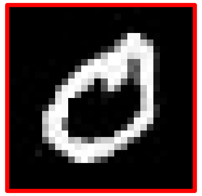
Define Activation Function

$$a_i^l = f\left(z_i^l\right)$$

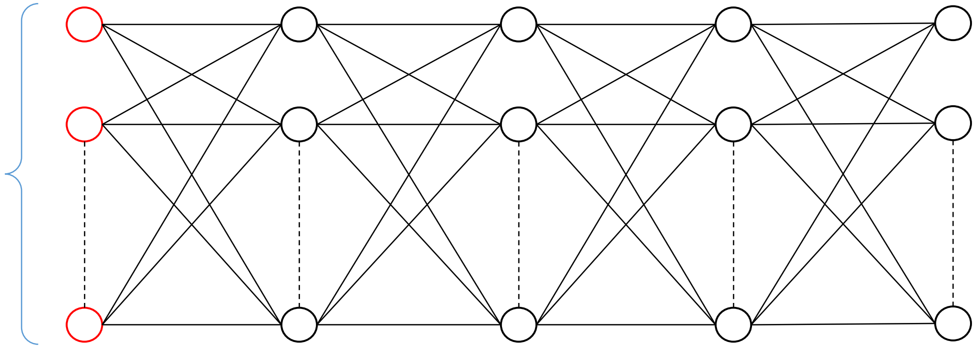
Sigmoid function

$$f(z) = \frac{1}{1+e^{-z}}$$





784-dimension



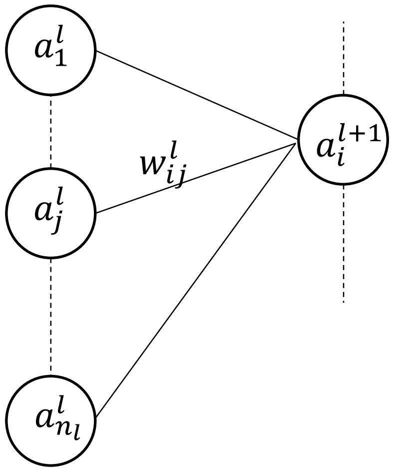
Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
784	256	128	64	10

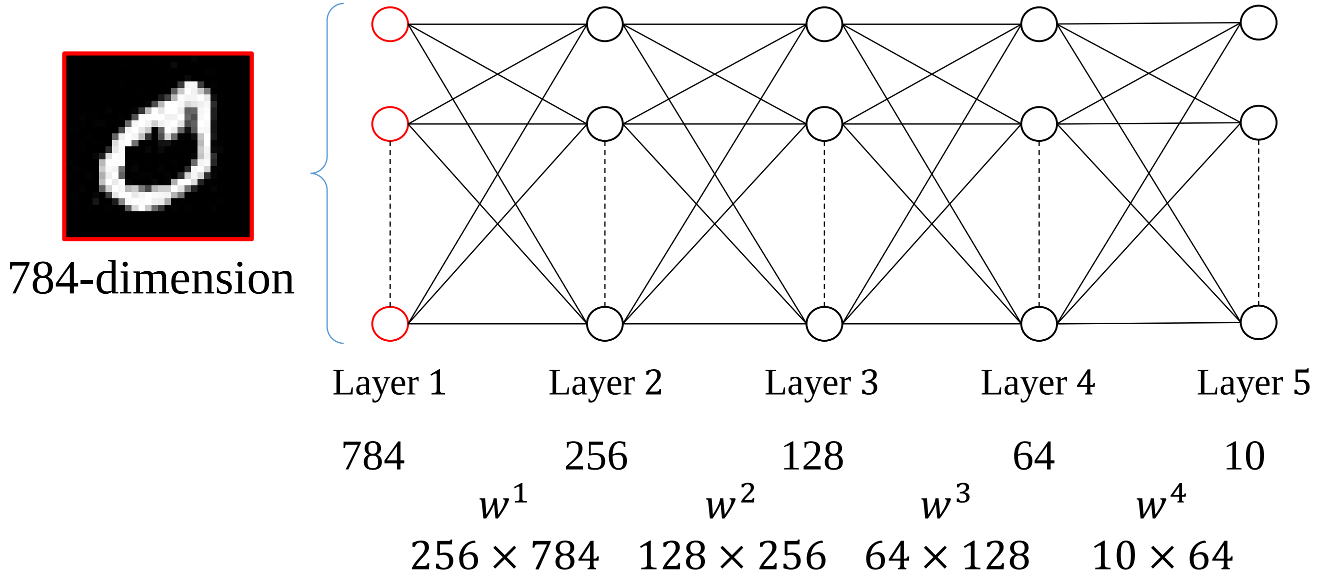
w^1	w^2	w^3	w^4
256×784	128×256	64×128	10×64

Step 3: Initialize Parameters

Initialize connection weights

Random initialization:
Gaussian distribution: $w_{ij}^{l} \sim N(0,1)$



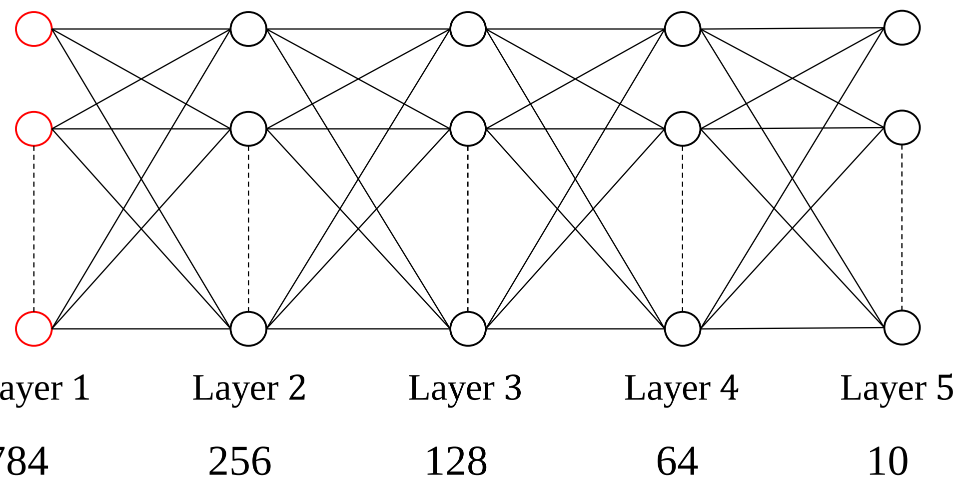


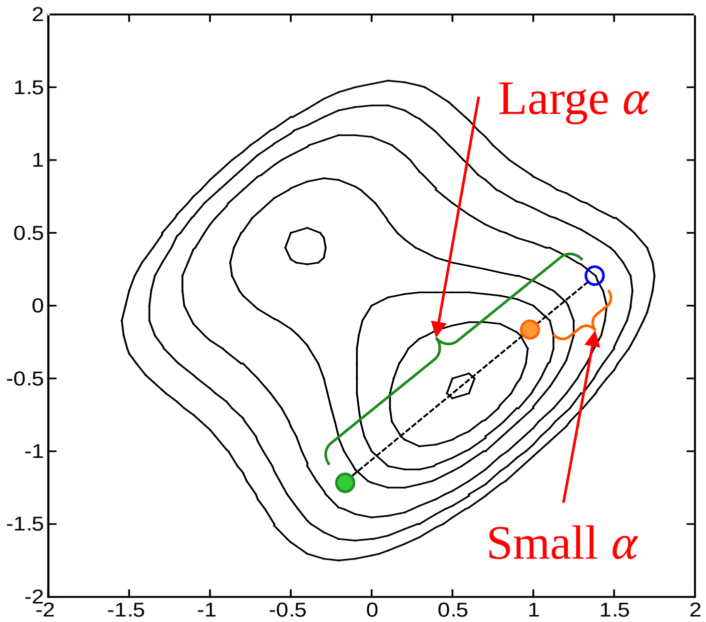
Step 3: Initialize Parameters

Learning rate:

- Small: slow learning, long learning time.
- Large: fast learning, possibly not converge to minima.

$$w_{ji}^{l} \leftarrow w_{ji}^{l} - \alpha \cdot \frac{\partial J}{\partial w_{ji}^{l}} \qquad \alpha = \cdots, 0.005, 0.01, 0.02, 0.04, \cdots$$

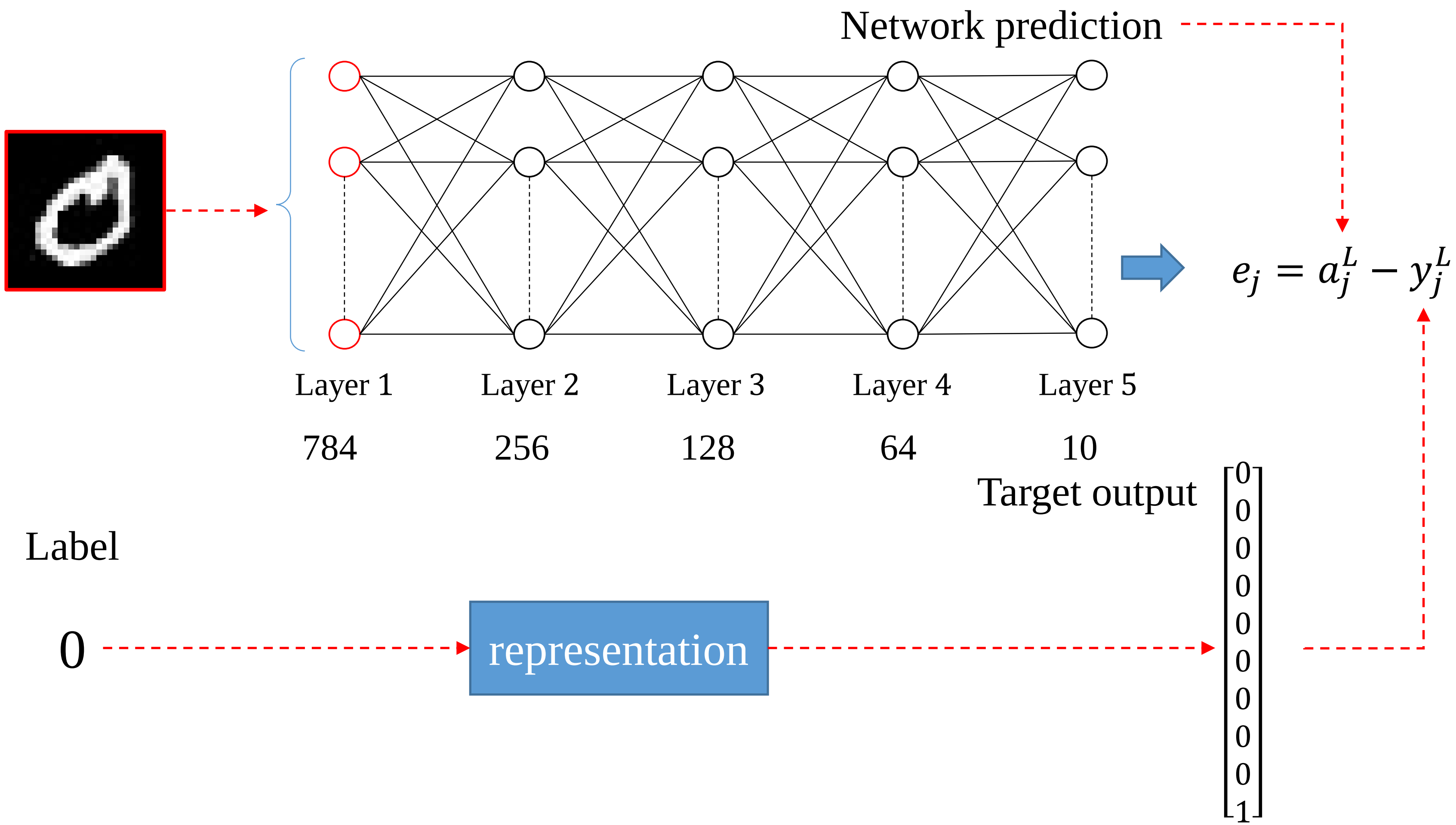




Step 4: Define Cost Function

Cost function

$$J = \frac{1}{2}\sum_{j=1}^{n_L} e_j^2$$



Step 5: Define Evaluation Index

$$\text{Acc}=\frac{\text{number of correct prediction}}{\text{number of samples}}$$

An example

Tested data 

Prediction 7 9 0 4 8 8 8 3 1

Correct prediction
7

Incorrect prediction
2

$$\text{Accuracy} = \frac{7}{9} = 77.78\%$$

Test on training set:

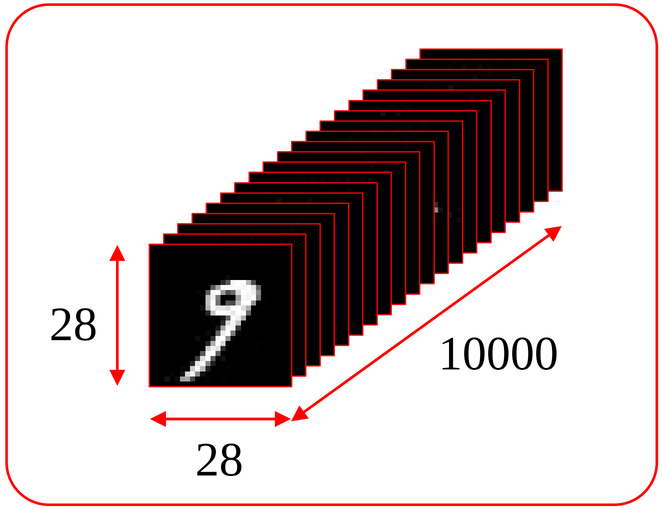
- Evaluate the ability of the model to fit given data.

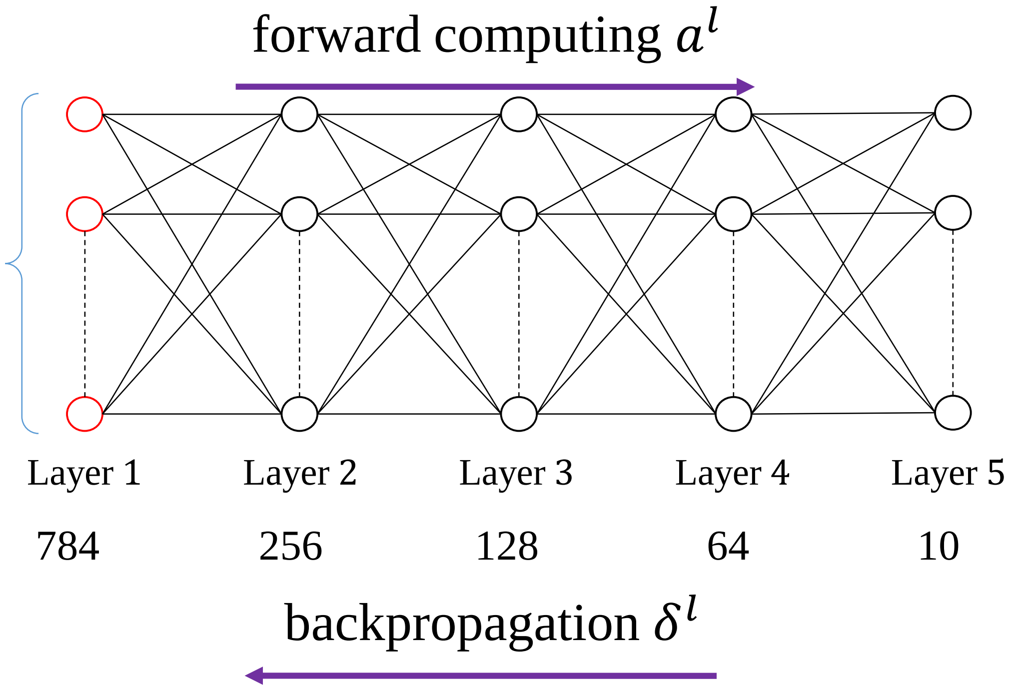
Test on testing set:

- Evaluate the ability of the model to generalize the knowledge.

Step 6: Train the Network

Training Data



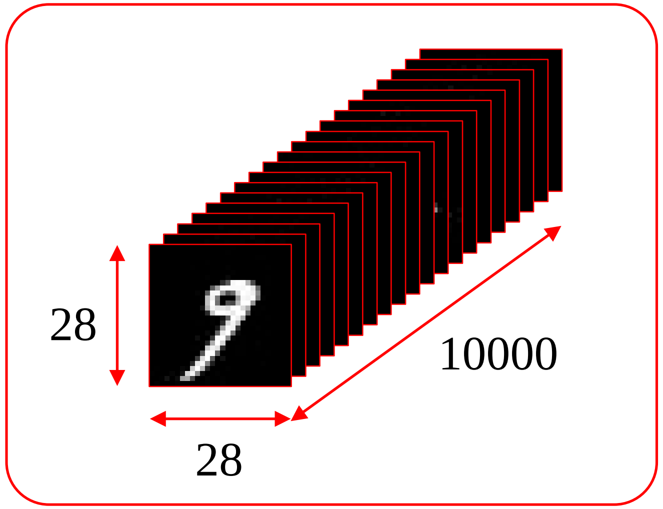


Updating weights

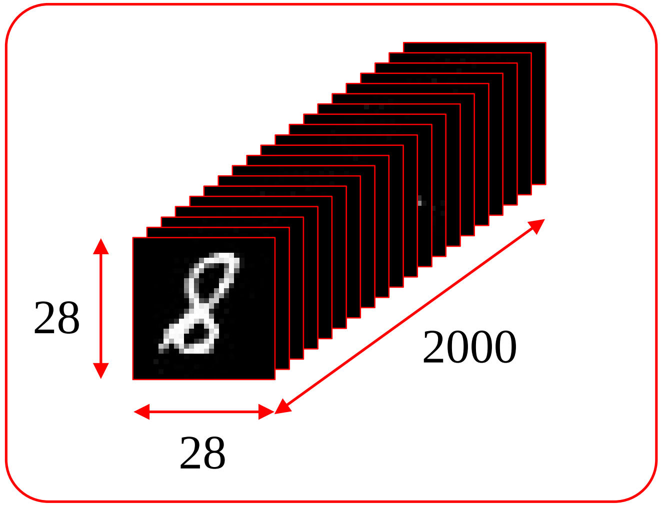
$$w_{ji}^l \leftarrow w_{ji}^l - \alpha \cdot \frac{\partial J}{\partial w_{ji}^l}$$

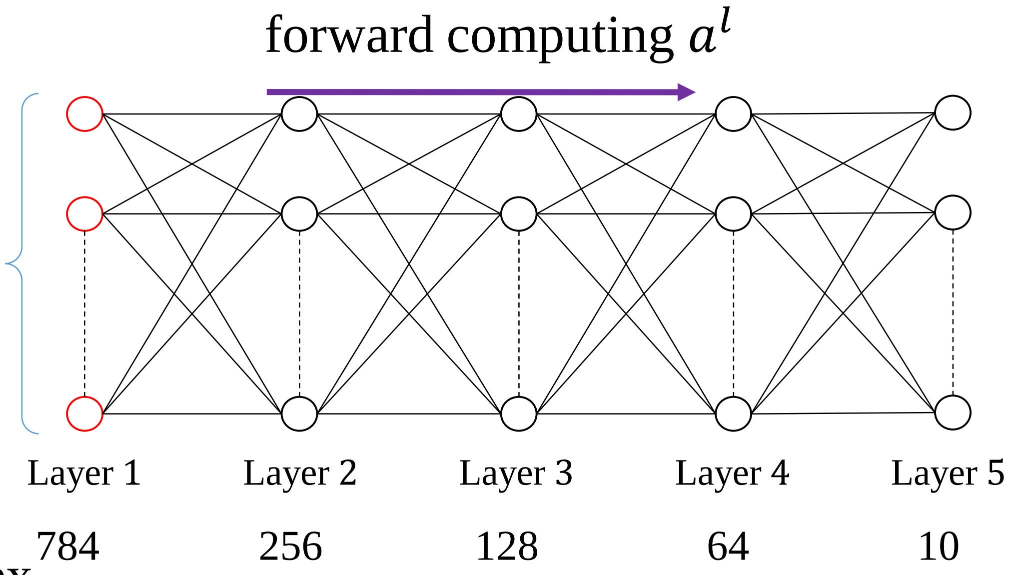
Step 7: Test the Network

Training Data



Testing Data





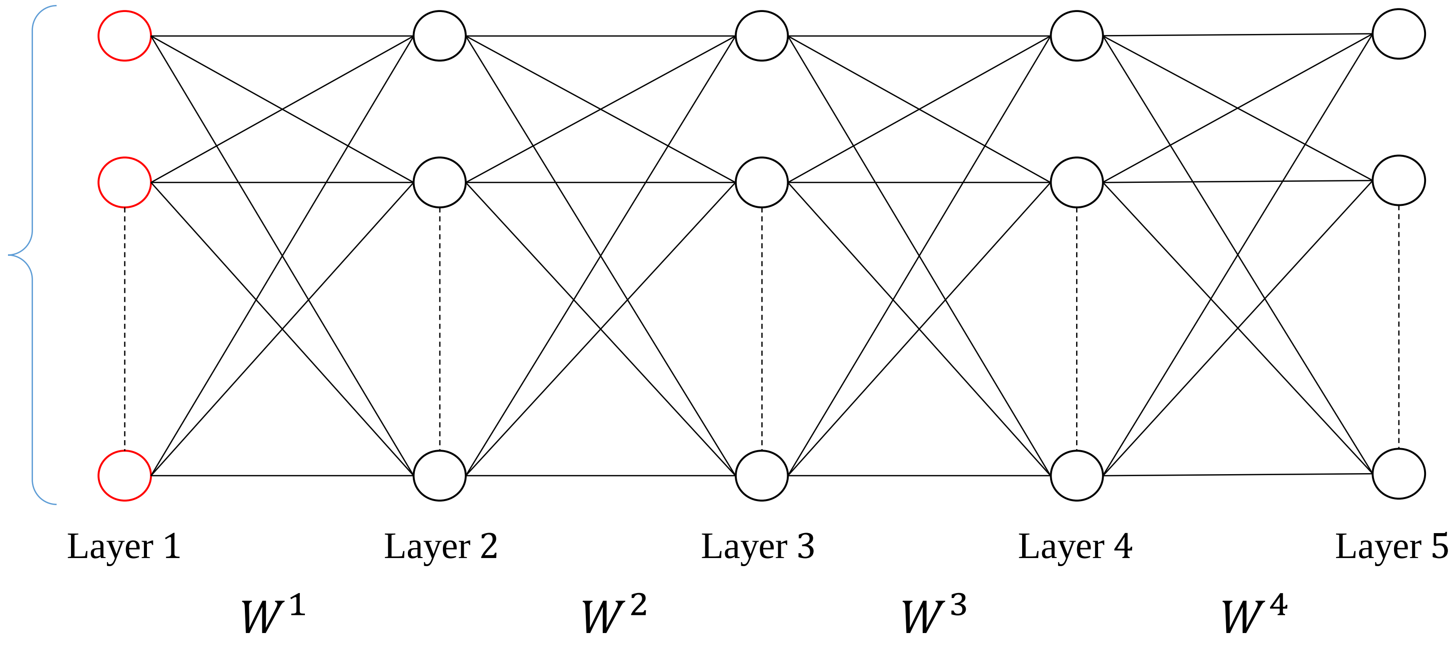
Calculating the Evaluation index on training set

$$\text{Acc} = \frac{\text{number of correct prediction}}{10000}$$

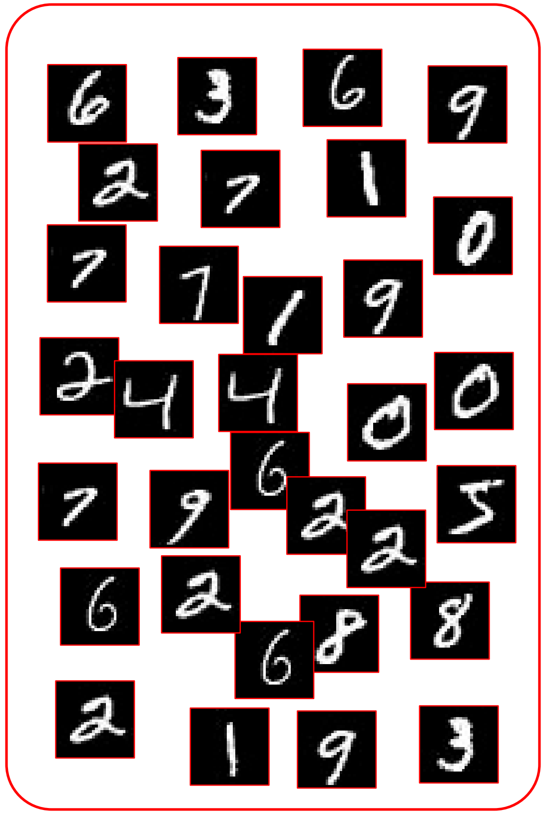
Calculating the Evaluation index on testing set

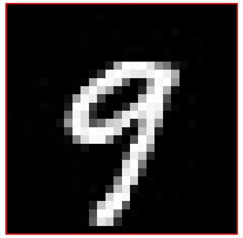
$$\text{Acc} = \frac{\text{number of correct prediction}}{2000}$$

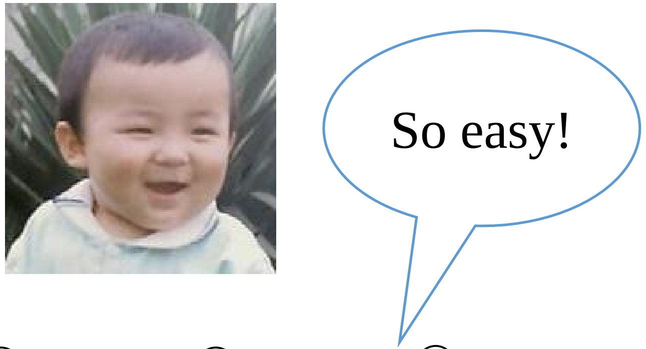
Step 8: Store the Network Parameters

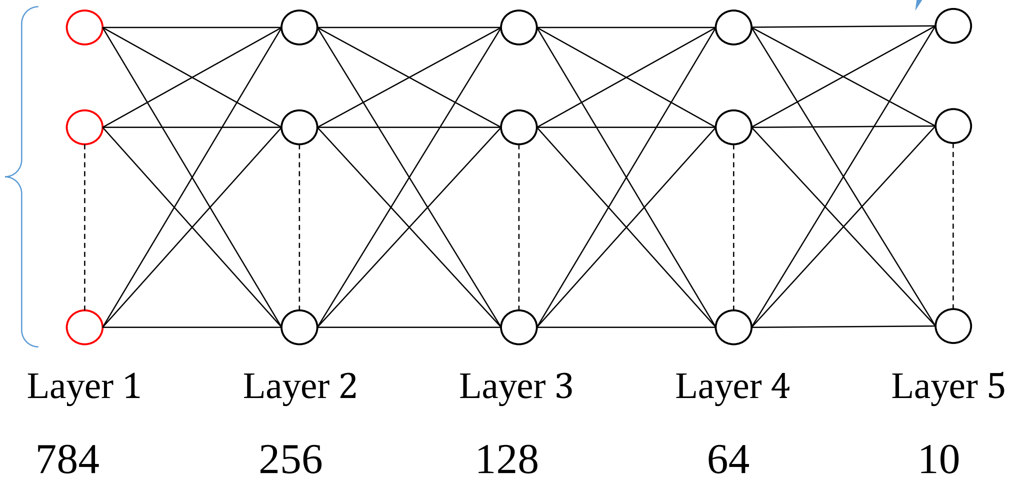


Step 9: Use Trained Network for Application







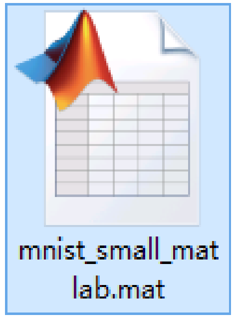


$$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix}$$

Outline

- Brief Review of Backpropagation Algorithm
- An Illustrating Example
- Experiments
- Assignment

Step 1: Data Preparation

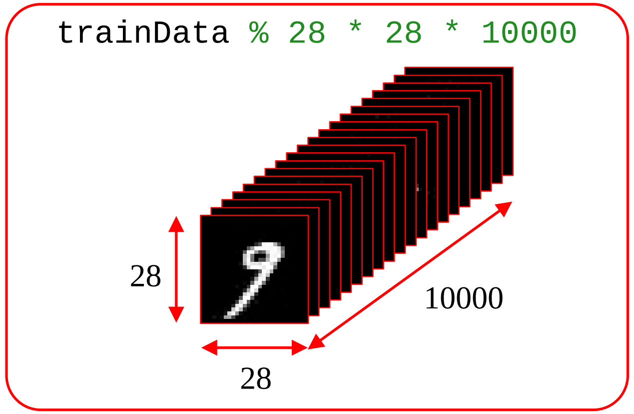


名称	值
trainData	28x28x10000 double
testData	28x28x2000 double
trainLabels	10x10000 double
testLabels	10x2000 double

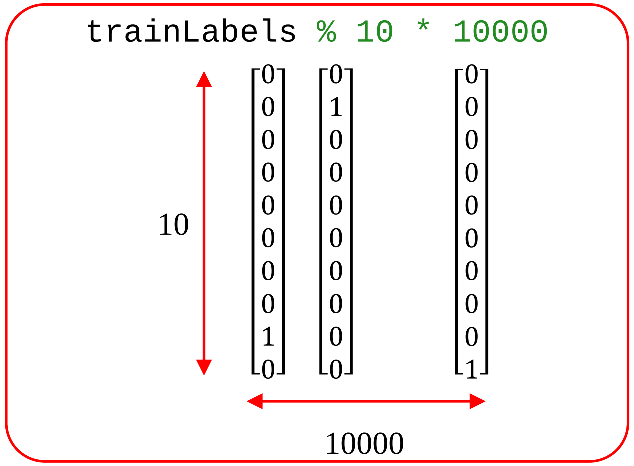
```
%% Step 1: Data Preparation

% loading dataset
load mnist_small_matlab.mat
% trainData: a matrix with size of 28x28x10000
% trainLabels: a matrix with size of 10x10000
% testData: a matrix with size of 28x28x2000
% testLabels: a matrix with size of 10x2000
```

Training Data



Label



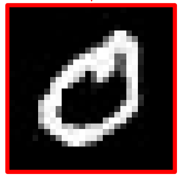
Step 1: Data Preparation

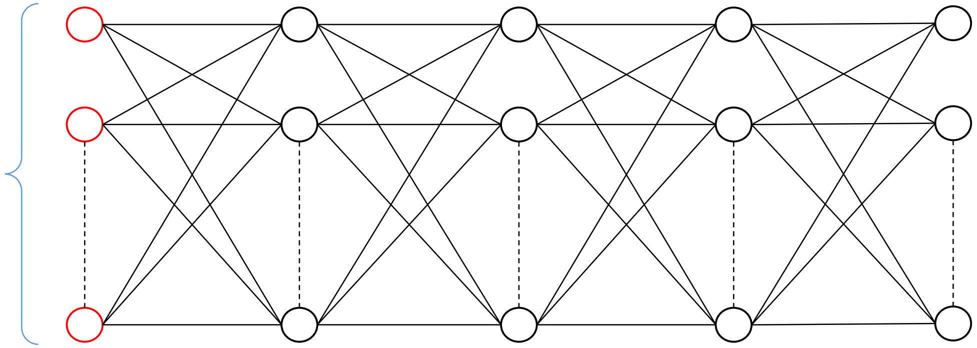
```
% prepare the external input for training set

train_size = 10000; % number of training samples
% input in the 1st layer
X_train = reshape(trainData, 784, train_size);
```

```
% prepare the external input for
testing set

test_size = 2000; % number of testing
samples
% external input in the 1st layer
X_test = reshape(testData,
                    784,
                    test_size);
```





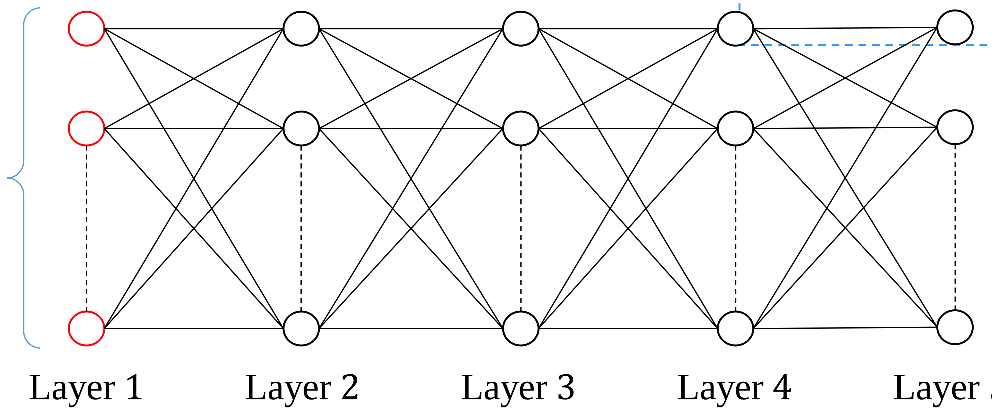
TIPs

- X_train and X_test are cells in Matlab
- reshape and zeros are built-in functions in Matlab

Step 2: Design Network Architecture

```
% define number of layers
L = 5;

% define number of neurons in each layer
% - 1st column: external neurons
% - 2nd column: internal neurons
layer_size = [784 % number of neurons in 1st layer
              256 % number of neurons in 2nd layer
              128 % number of neurons in 3rd layer
              64  % number of neurons in 4th layer
              10];% number of neurons in 5th layer
```



784	256	128	64	10

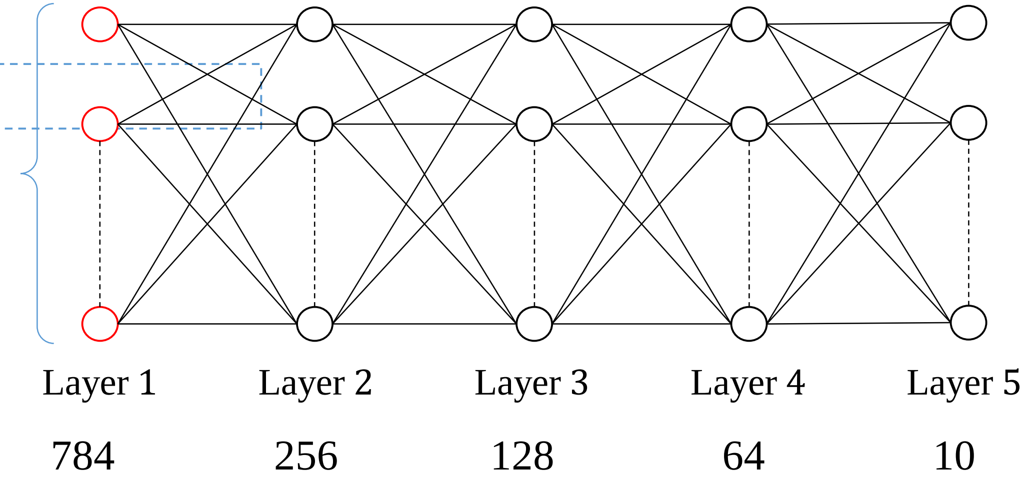
Step 3: Initialize Parameters

Gaussian distribution: $w_{ij}^{l} \sim N(0,1)$

```
% initialize weights
for l = 1:L-1
    w{l} = 0.1*randn(layer_size(l+1,1), sum(layer_size(l,:)));
end
```

Initialize learning rate

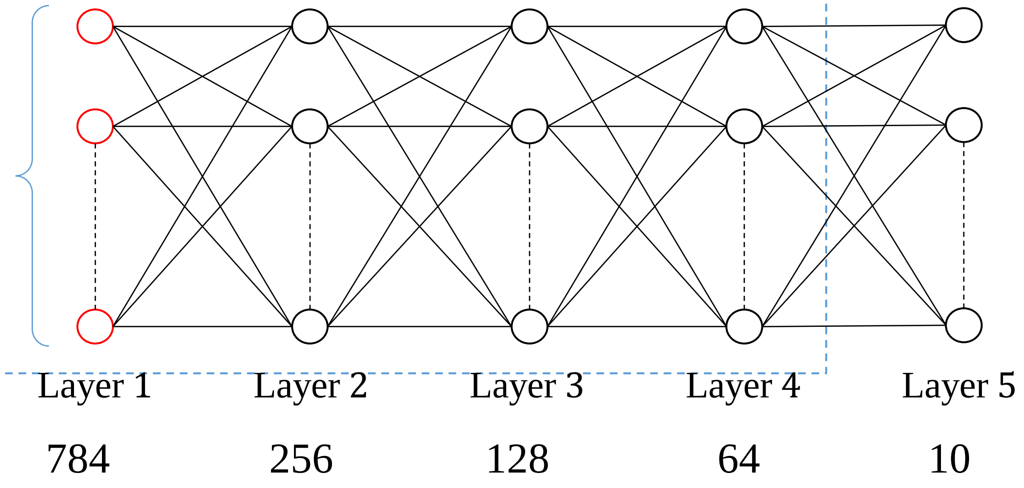
```
alpha = 0.005; % initialize learning rate
```



Step 4, 5: Define Cost Function and Evaluation Index

```
%% Step 4: Define Cost Function
function [J] = cost(a, y)
    J = 1/2 * sum((a - y).^2);
end

%% Step 5: Define Evaluation Index
function [ acc ] = accuracy(a, y)
    mini_batch = size(a, 2);
    [~,idx_a] = max(a);
    [~,idx_y] = max(y);
    acc = sum(idx_a==idx_y) / mini_batch;
end
```



Cost function

$$J = \frac{1}{2}\sum_{j=1}^{n_L} e_j^2$$

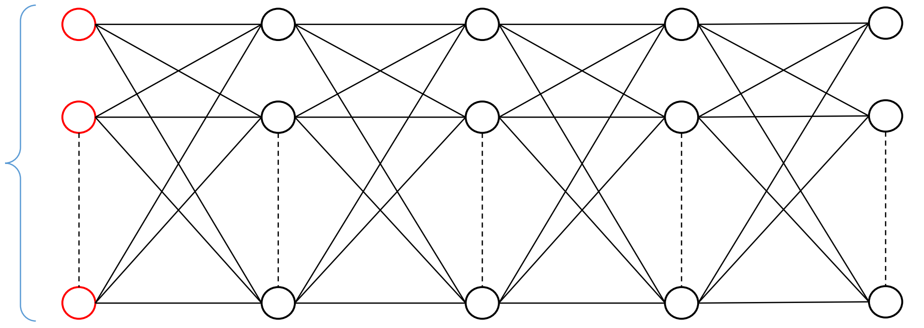
$$\text{Acc} = \frac{\text{number of correct prediction}}{\text{number of samples}}$$

Step 6: Train the Network

```
%% Step 6: Train the Network
J = []; % array to store cost of each mini batch
Acc = []; % array to store accuracy of each mini batch
max_epoch = 200; % number of training epochs
mini_batch = 100; % number of sample in each mini batch
```

```
% loop until converge
for iter = 1:max_epoch
    % for each mini-batch
        % batch forward computation
        % batch backward computation
        % cumulate and update weight
end
```

Mini-batch BP implement



BP Algorithm:

Step 1. Input the training data set $D = \{(x, y)\}$

Step 2. Initialize each w_{ij}^l, and choose a learning rate α.

Step 3. for each mini-batch sample $D_m \subseteq D$

for each $x \in D_m$

$a^1 \leftarrow x \in D_m$;

for $l = 1{:}L\text{-}1$

$a^{l+1} \leftarrow fc(w^l, a^l)$;

end

$\delta^L = \frac{\partial J}{\partial z^L}$;

for $l = L-1{:}2$

$\delta^l \leftarrow bc(w^l, \delta^{l+1})$;

end

$\frac{\partial J}{\partial w_{ji}^l} \leftarrow \frac{\partial J}{\partial w_{ji}^l} + \delta_j^{l+1} \cdot a_i^l$;

end

$w_{ji}^l \leftarrow w_{ji}^l - \alpha \cdot \frac{\partial J}{\partial w_{ji}^l}$;

end

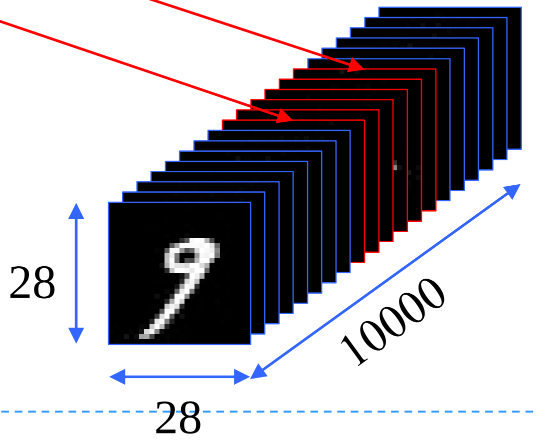
Step 4. Return to Step 3 until each w^l converge.

Step 6: Train the Network

```
% randomly permutate the indexes of samples in training set
idxs = randperm(train_size);
% for each mini-batch
for k = 1:ceil(train_size/mini_batch)
    % prepare internal inputs in 1st layer denoted by a{1}

    start_idx = (k-1)*mini_batch+1;          % start index of kth mini-batch
    end_idx = min(k*mini_batch, train_size); % end index of kth mini-batch
    a{1} = X_train(:,idxs(start_idx:end_idx));

    % prepare labels
    y = trainLabels(:, idxs(start_idx:end_idx));
    % forward computation
    % ...
    % cost function
    % ...
    % backward computation
    % ...
    % update weight
    % ...
end
```



training data

TIPs

- randperm is a built-in function in Matlab

Step 6: Train the Network

```
% forward computation
for l=1:L-1
    [a{l+1}, z{l+1}] = fc(w{l}, a{l});
end

% Compute delta of last layer
delta{L} = (a{L} - y).* a{L} .*(1-a{L});

% backward computation
for l=L-1:-1:2
    delta{l} = bc(w{l}, z{l}, delta{l+1});
end

% update weight
for l=1:L-1
    % compute the gradient
    grad_w = delta{l+1} * a{l}';
    w{l} = w{l} - alpha*grad_w;
end
```

BP Algorithm:

Step 1. Input the training data set $D = \{(x, y)\}$

Step 2. Initialize each w_{ij}^{l}, and choose a learning rate α.

Step 3. for each mini-batch sample $D_m \subseteq D$

for each $x \in D_m$

$a^1 \leftarrow x \in D_m$;

for $l = 1{:}L\text{-}1$

$a^{l+1} \leftarrow fc(w^l, a^l)$;

end

$\delta^L = \frac{\partial J}{\partial z^L}$;

for $l = L - 1{:}\,2$

$\delta^l \leftarrow bc(w^l, \delta^{l+1})$;

end

$\frac{\partial J}{\partial w_{ji}^{l}} \leftarrow \frac{\partial J}{\partial w_{ji}^{l}} + \delta_j^{l+1} \cdot a_i^l$;

end

$w_{ji}^{l} \leftarrow w_{ji}^{l} - \alpha \cdot \frac{\partial J}{\partial w_{ji}^{l}}$;

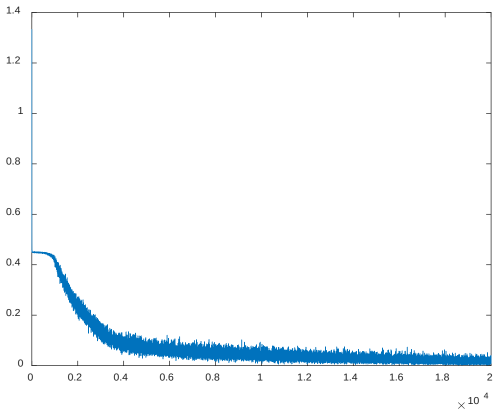
end

Step 4. Return to Step 3 until each w^l converge.

Step 6: Train the Network

Cost function

$$J = \frac{1}{2}\sum_{j=1}^{n_L}\left(a_j^L - y_j^L\right)^2$$

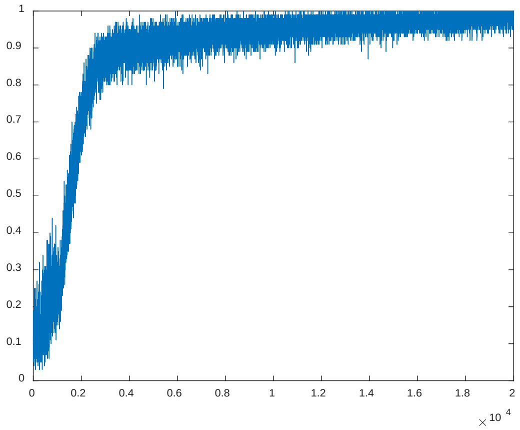



```
% training cost on training batch
J = [J 1/mini_batch*cost(a{L}, y)];
figure
plot(J);
```

Accuracy

$$\text{Acc} = \frac{\textit{number of correct prediction}}{\textit{number of samples}}$$

Use max output as prediction




```
% accuary on training batch
Acc =[Acc accuracy(a{L}, y)];
figure
plot(Acc);
```

Step 7: Test the Network

```
%test on training set
a{1} = X_train;
for l = 1:L-1
  a{l+1} = fc(w{l}, a{l});
end
train_acc = accuracy(a{L}, trainLabels);
fprintf('Accuracy on training dataset is %f%%\n', train_acc*100);
```

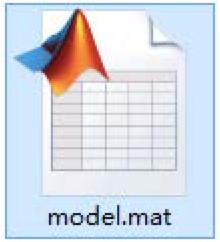
Accuracy on training dataset is 98.160000%

```
%test on testing set
a{1} = X_test;
for l = 1:L-1
   a{l+1} = fc(w{l}, a{l});
end
test_acc = accuracy(a{L}, testLabels);
fprintf('Accuracy on testing dataset is %f%%\n', test_acc*100);
```

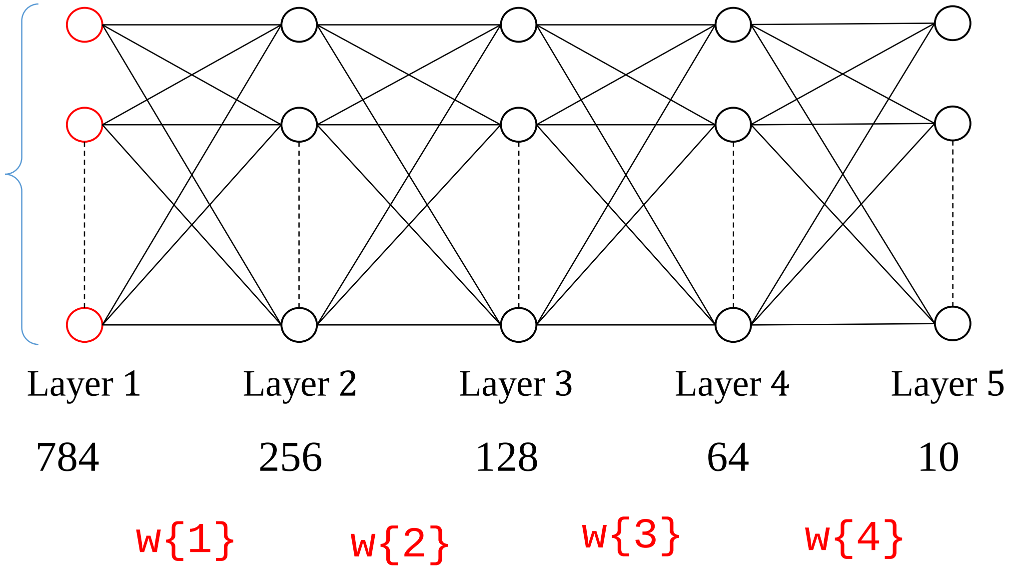
Accuracy on testing dataset is 95.550000%

Experiments: Store the Network Parameters

```
% save model
save model.mat w layer_size
```



This is very important!



Experiments: Code Structure

The main running script	Forward computing function	Backward computing function	Cost function	Evaluation Index

Experiments: Code—main.m

```
%% Step 1: Data Preparation

% loading dataset
load mnist_small_matlab.mat
% trainData: a matrix with size of 28x28x10000
% trainLabels: a matrix with size of 10x10000
% testData: a matrix with size of 28x28x2000
% testLabels: a matrix with size of 10x2000
```

```
train_size = 10000; % number of training samples
% input in the 1st layer
X_train = reshape(trainData, 784, train_size);
```

Experiments: Code—main.m

```
test_size = 2000; % number of testing samples
% external input in the 1st layer
X_test = reshape(testData, 784, test_size);
```

```
%% Step 2: Design Network Architecture
% define number of layers
L = 5;
% define number of neurons in each layer
layer_size = [784 % number of neurons in 1st layer
              256 % number of neurons in 2nd layer
              128 % number of neurons in 3rd layer
              64  % number of neurons in 4th layer
              10];% number of neurons in 5th layer
```

Experiments: Code—main.m

```
%% Step 3: Initial Parameters

% initialize weights in each layer with Gaussian distribution
for l = 1:L-1
    w{l} = 0.1 * randn(layer_size(l+1,1), sum(layer_size(l,:)));
end

alpha = 0.005; % initialize learning rate

%% Step 4: Define Cost Function
% cost function is defined in cost.m

%% Step 5: Define Evaluation Index
% accuracy defined in accuracy.m
```

Experiments: Code—main.m

```
%% Step 6: Train the Network
J = []; % array to store cost of each mini batch
Acc = []; % array to store accuracy of each mini batch
max_epoch = 200; % number of training epoch
mini_batch = 100; % number of sample of each mini batch

figure % plot the cost
for iter=1:max_epoch
    % randomly permute the indexes of samples in training set
idxs = randperm(train_size);
% for each mini-batch
for k = 1:ceil(train_size/mini_batch)
    % prepare internal inputs in 1st layer denoted by a{1}

    start_idx = (k-1)*mini_batch+1;          % start index of kth mini-batch
    end_idx = min(k*mini_batch, train_size); % end index of kth mini-batch
    a{1} = X_train(:,idxs(start_idx:end_idx));
    % prepare labels
    y = trainLabels(:, idxs(start_idx:end_idx));
```

Experiments: Code—main.m

```
% forward computation
for l=1:L-1
    [a{l+1}, z{l+1}] = fc(w{l}, a{l});
end

% Compute delta of last layer
delta{L} = (a{L} - y).* a{L} .*(1-a{L}); %delta{L}={partial J}/{partial z^L}

% backward computation
for l=L-1:-1:2
    delta{l} = bc(w{l}, z{l}, delta{l+1});
end

% update weight
for l=1:L-1
    % compute the gradient
    grad_w = delta{l+1} * a{l}';
    w{l} = w{l} - alpha*grad_w;
end
```

Experiments: Code—main.m

```
        % training cost on training batch
        J = [J 1/mini_batch*sum(cost(a{L}, y))];
        Acc =[Acc accuracy(a{L}, y)];
        % plot training error
        plot(J);
        pause(0.000001);
    end
end
% end training
% plot accuracy
figure
plot(Acc);
```

Experiments: Code—main.m

```
%% Step 7: Test the Network
%test on training set
a{1} = X_train;
for l = 1:L-1
  a{l+1} = fc(w{l}, a{l});
end
train_acc = accuracy(a{L}, trainLabels);
fprintf('Accuracy on training dataset is %f%%\n', train_acc*100);

%test on testing set
a{1} = X_test;
for l = 1:L-1
   a{l+1} = fc(w{l}, a{l});
end
test_acc = accuracy(a{L}, testLabels);
fprintf('Accuracy on testing dataset is %f%%\n', test_acc*100);

%% Step 8: Store the Network Parameters
save model.mat w layer_size
```

Experiments: Code—fc.m and bc.m

```
% fc.m
% This is forward computation function

function [a_next, z_next] = fc(w, a)
    % define the activation function
    f = @(s) 1 ./ (1 + exp(-s));

    % forward computing
    z_next = w * a;
    a_next = f(z_next);
end
```

```
% bc.m
% This is backward computation function

function delta = bc(w, z, delta_next)

    % activation function
    f = @(s) 1 ./ (1 + exp(-s));

    % derivative of activation  function
    df = @(s) f(s) .* (1 - f(s));

    % backward computing
    delta = (w' * delta_next) .* df(z);

end
```

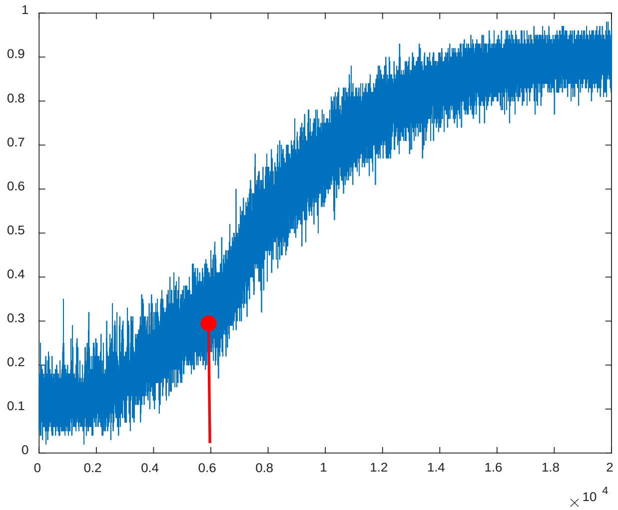
Experiments: Code—cost.m and accuracy.m

```
%% Step 4: Define Cost Function
function [J] = cost(a, y)
    J = 1/2 * sum((a - y).^2);
end
```

```
%% Step5: Define Evaluation Index
function [ acc ] = accuracy( a, y )
    mini_batch = size(a, 2);
    [~,idx_a] = max(a);
    [~,idx_y] = max(y);
    acc = sum(idx_a==idx_y) / mini_batch;
end
```

Results: Learning Rate

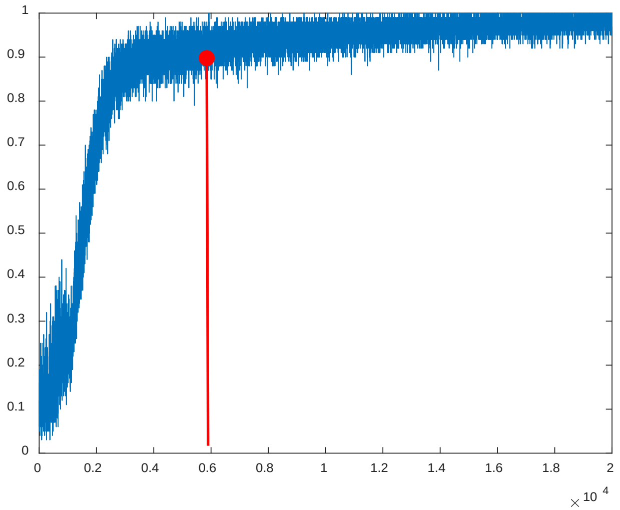
```
% learning rate
alpha = 0.001;
```



Accuracy
- Training=90.24%
- Testing=89.10%

Too Slow

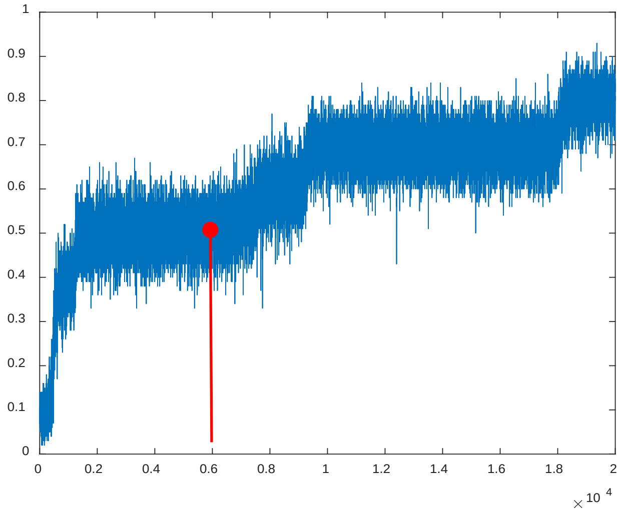
```
% learning rate
alpha = 0.005;
```



Accuracy
- Training=98.15%
- Testing=95.25%

Good

```
% learning rate
alpha = 0.08;
```



Accuracy
- Training=79.66%
- Testing=76.65%

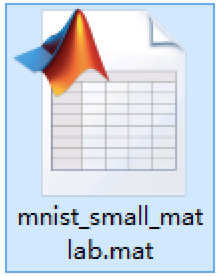
Too Fast

Outline

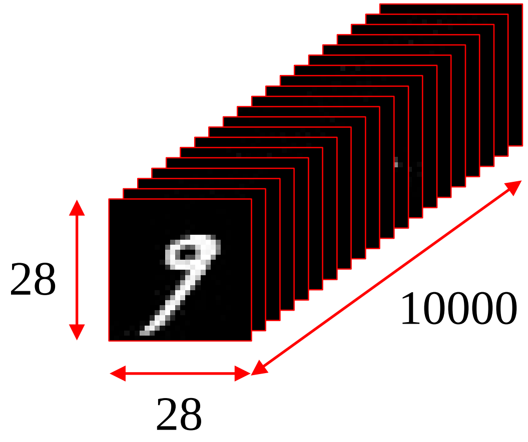
- Brief Review of Backpropagation Algorithm
- An Illustrating Example
- Experiments
- Assignment

Assignment

1. Copy and run the handwritten digits recognition code by MATLAB using only one layer of external input.



2. Try different network layers with different neurons and plot the results (such as $L = 2, 3, 8$).



Thanks