Neural Network and Classification

生醫光電所 吳育德

Phil Kim, MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence, 2017, chapter 4.

Binary Classification

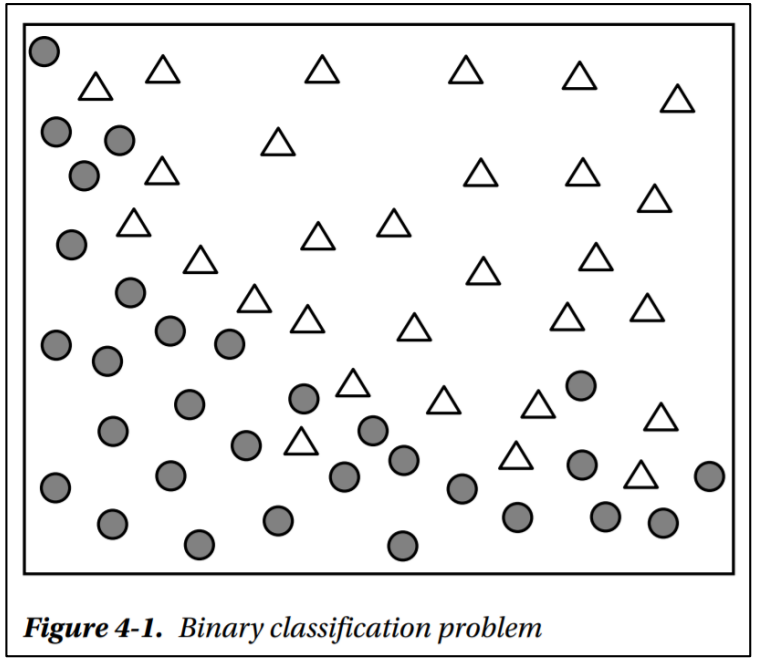


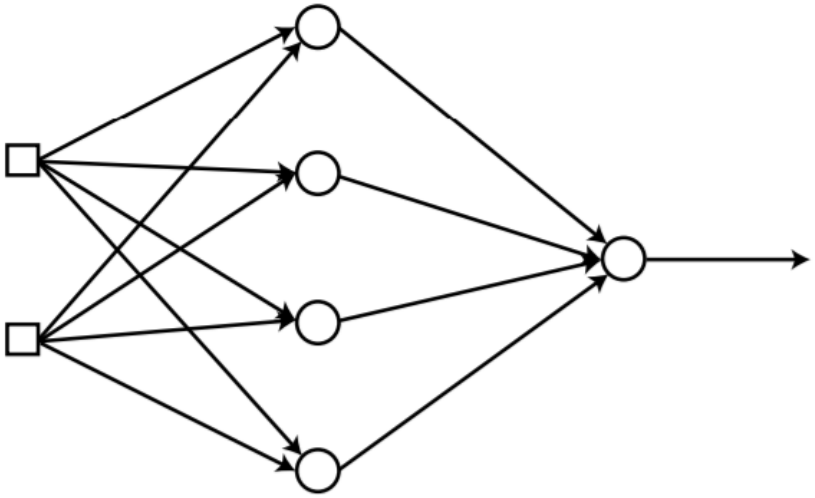
Figure 4-1. *Binary classification problem*

$\{5, 7, \triangle\}$
$\{9, 8, \bullet\}$
…
$\{6, 5, \bullet\}$

$Class\ \triangle \rightarrow 1$

$Class\ \bullet \rightarrow 0$

Figure 4-2. Training data binary classification.



$> 0.5 \rightarrow 1$

$< 0.5 \rightarrow 0$

Learning process of the binary classification：

(1) One node for the output layer. The sigmoid function is used for the activation function.

(2) Switch the class titles of the training data into numbers using the maximum and minimum values of the sigmoid function.

$$Class\ \triangle \rightarrow 1$$
$$Class\ \bullet \rightarrow 0$$

(3) Initialize the weights of the neural network with adequate values.

(4) Enter the training data { input, correct output } into the neural network.
Calculate the error and determine the delta δ of the output nodes.

$$e = d - y$$
$$\delta = e$$

(5) Propagate the output delta backwards and calculate the delta of the subsequent hidden nodes.

$$e^{(k)} = W^T \delta$$
$$\delta^{(k)} = \varphi'\left(v^{(k)}\right) e^{(k)}$$

(6) Repeat Step 5 until it reaches the hidden layer on the immediate right of the input layer.

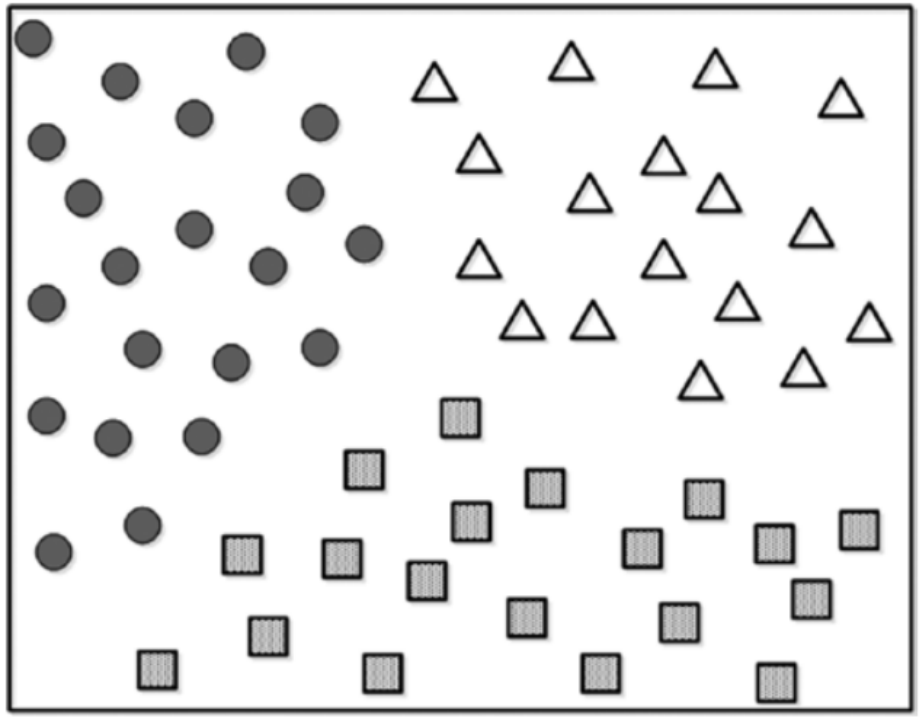
(7) Adjust the weights of the neural network using this learning rule：

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

(8) Repeat Steps 4-7 for all training data points.

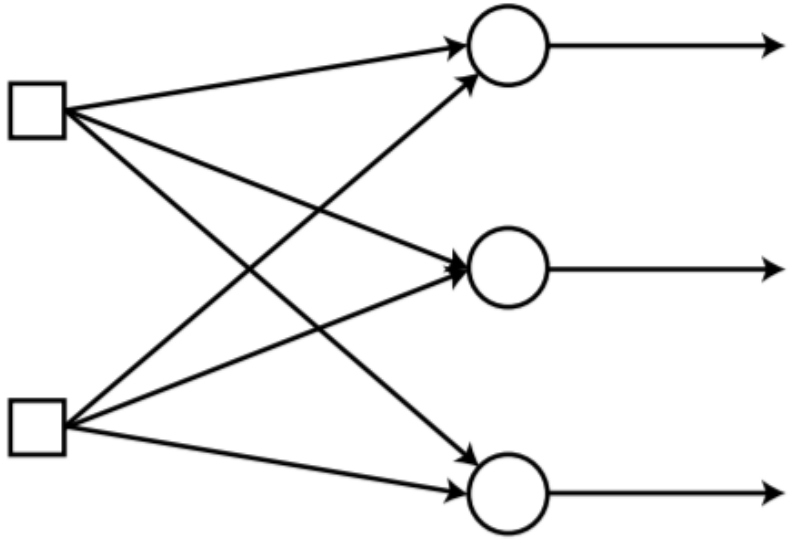
(9) Repeat Steps 4-8 until the neural network has been trained properly.

Multiclass Classification



$\{5, 7, Class\ 1\}$
$\{9, 8, Class\ 3\}$
$\{2, 4, Class\ 2\}$
…
$\{6, 5, Class\ 3\}$

$\{5, 7, \mathbf{1}, \mathbf{0}, \mathbf{0}\}$
$\{9, 8, \mathbf{0}, \mathbf{0}, \mathbf{1}\}$
$\{2, 4, \mathbf{0}, \mathbf{1}, \mathbf{0}\}$
…
$\{6, 5, \mathbf{0}, \mathbf{0}, \mathbf{1}\}$



$$Class\ 1 \rightarrow [1, 0, 0]$$
$$Class\ 2 \rightarrow [0, 1, 0]$$
$$Class\ 3 \rightarrow [0, 0, 1]$$

one-hot encoding or
1-of-N encoding

- In general, multiclass classifiers employ the softmax function as the activation function of the output node.

- The output from the *i*-th output node of the softmax function is：

$$y_i = \varphi(v_i) = \frac{e^{v_i}}{e^{v_1} + e^{v_2} + e^{v_3} + \cdots + e^{v_M}} = \frac{e^{v_i}}{\sum_{k=1}^{M} e^{v_k}}$$

v_i is the weighted sum of the *i*-th output node.
M is the number of output nodes.
$\varphi(v_1) + \varphi(v_2) + \varphi(v_3) + \cdots + \varphi(v_M) = 1$

- Example:

$$v = \begin{bmatrix} 2 \\ 1 \\ 0.1 \end{bmatrix} \quad \Rightarrow \quad \varphi(v) = \begin{bmatrix} \frac{e^2}{e^2 + e^1 + e^{0.1}} \\ \frac{e^1}{e^2 + e^1 + e^{0.1}} \\ \frac{e^{0.1}}{e^2 + e^1 + e^{0.1}} \end{bmatrix} = \begin{bmatrix} 0.6590 \\ 0.2424 \\ 0.0986 \end{bmatrix}$$

The training process of the multiclass classification neural network is：

(1)Construct the number of output nodes to be the number of classes. The softmax function is used as the activation function.

(2)Switch the names of the classes into numeric vectors via the one-hot encoding

$$Class\ 1 \rightarrow [1,0,0]$$
$$Class\ 2 \rightarrow [0,1,0]$$
$$Class\ 3 \rightarrow [0,0,1]$$

(3) Initialize the weights of the neural network with adequate values.

(4) Enter the training data { input, correct output } into the neural network. Calculate the error and determine the delta δ.

$$e = d - y$$
$$\delta = e$$

(5) Propagate the output delta backwards and calculate the delta of the subsequent hidden nodes.

$$e^{(k)} = W^T \delta$$
$$\delta^{(k)} = \varphi'\left(v^{(k)}\right) e^{(k)}$$

(6) Repeat Step 5 until it reaches the hidden layer on the immediate right of the input layer.

(7) Adjust the weights of the neural network using this learning rule：

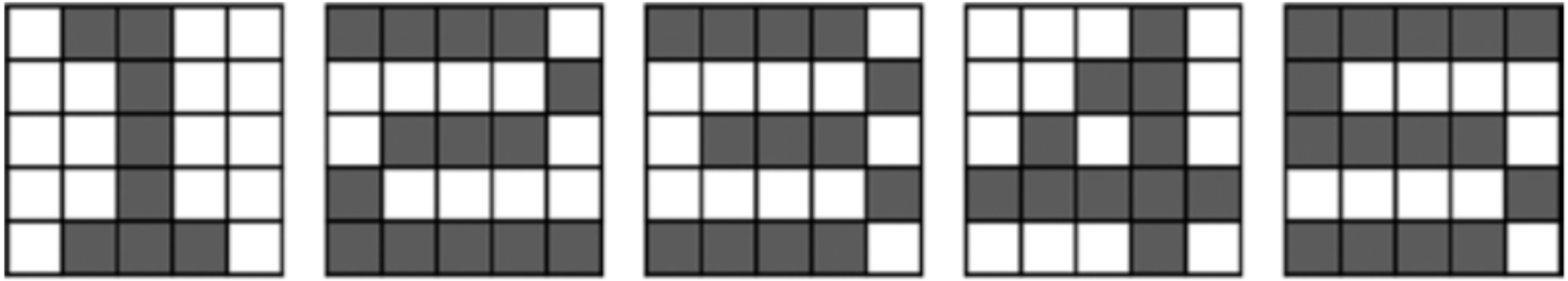
$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

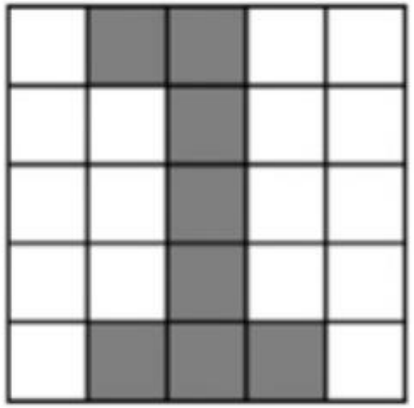
(8) Repeat Steps 4-7 for all the training data points.

(9) Repeat Steps 4-8 until the neural network has been trained properly.

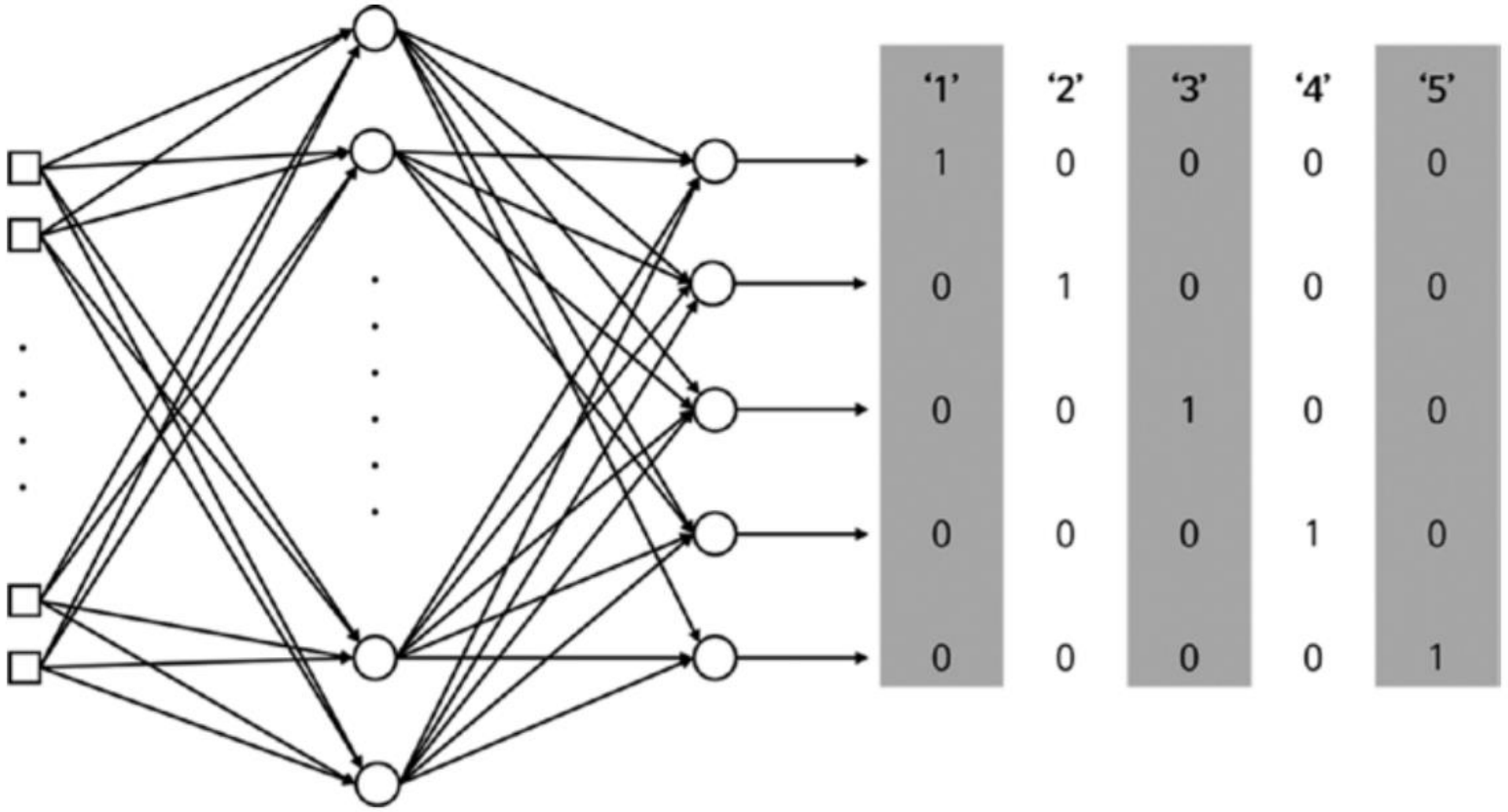
Example: Multiclass Classification

- The input images are five-by-five pixel squares, which display five numbers from 1 to 5.





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



- As each image is set on a matrix, we set 25 input nodes. In addition, as we have five digits to classify, the network contains five output nodes.

- The softmax function is used as the activation function of the output node. The hidden layer has 50 nodes and the sigmoid function is used as the activation function.

- The function MultiClass implements the learning rule of multiclass classification using the SGD method. It takes the input arguments of the weights and training data and returns the trained weights.

$$[W1, W2] = MultiClass(W1, W2, X, D)$$

W1 and W2 are the weight matrices of the input-hidden and hidden-output layers, respectively.
X and D are the input and correct output of the training data, respectively.

- MultiClass.m follows the same procedure as in Chapter 3, which applies the delta rule to the training data, calculates the weight updates, dW1 and dW2, and adjusts the neural network's weights.

- As this neural network is compatible with only the vector format inputs, the **two-dimensional matrix should be transformed into a 25x1 vector**.

$$X = reshape(X(:, :, k), 25, 1);$$

- The previous back-propagation algorithm applies to the hidden layer.

$$e1 = W2' * delta;$$
$$delta1 = y1 .* (1 - y1) .* e1;$$

```
function [W1, W2] = MultiClass(W1, W2, X, D)
  alpha = 0.9;

  N = 5;
  for k = 1:N
    x = reshape(X(:, :, k), 25, 1);
    d = D(k, :)';

    v1 = W1*x;
    y1 = Sigmoid(v1);
    v  = W2*y1;
    y  = Softmax(v);

    e     = d - y;
    delta = e;

    e1     = W2'*delta;
    delta1 = y1.*(1-y1).*e1;

    dW1 = alpha*delta1*x';
    W1 = W1 + dW1;

    dW2 = alpha*delta*y1';
    W2 = W2 + dW2;
  end
end
```

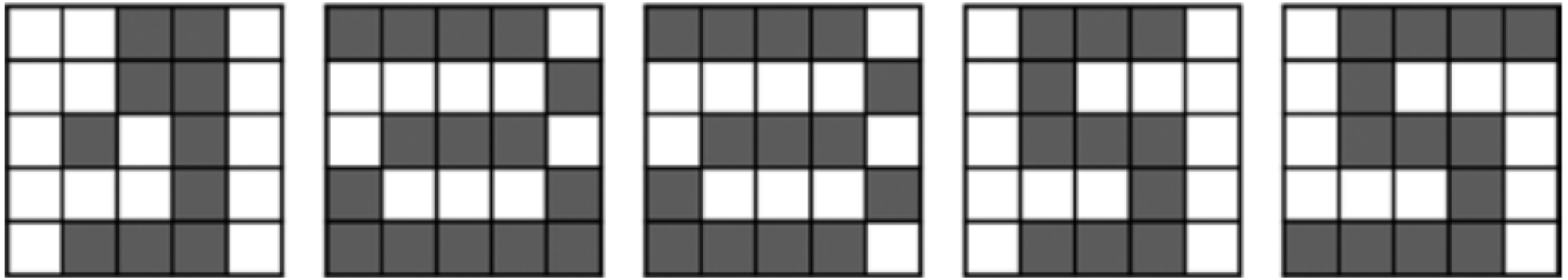
- This program calls MultiClass and trains the neural network 10,000 times. Once the training process has been finished, the program enters the training data into the neural network and displays the output.

```
clear all
rng(3);
X  = zeros(5, 5, 5);

X(:, :, 1) = [ 0 1 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 1 1 1 0 ];

X(:, :, 2) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               1 0 0 0 0;
               1 1 1 1 1 ];

X(:, :, 3) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0 ];
```

```
X(:, :, 4) = [ 0 0 0 1 0;
               0 0 1 1 0;
               0 1 0 1 0;
               1 1 1 1 1;
               0 0 0 1 0 ];

X(:, :, 5) = [ 1 1 1 1 1;
               1 0 0 0 0;
               1 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0 ];

D = [ 1 0 0 0 0;
      0 1 0 0 0;
      0 0 1 0 0;
      0 0 0 1 0;
      0 0 0 0 1 ];

W1 = 2*rand(50, 25) - 1;
W2 = 2*rand( 5, 50) - 1;
```

```
for epoch = 1:10000              % train
  [W1, W2] = MultiClass(W1, W2, X, D);
end


N = 5;                           % inference
for k = 1:N
  x  = reshape(X(:, :, k), 25, 1);
  v1 = W1*x;
  y1 = Sigmoid(v1);
  v  = W2*y1;
  y  = Softmax(v)
end
```

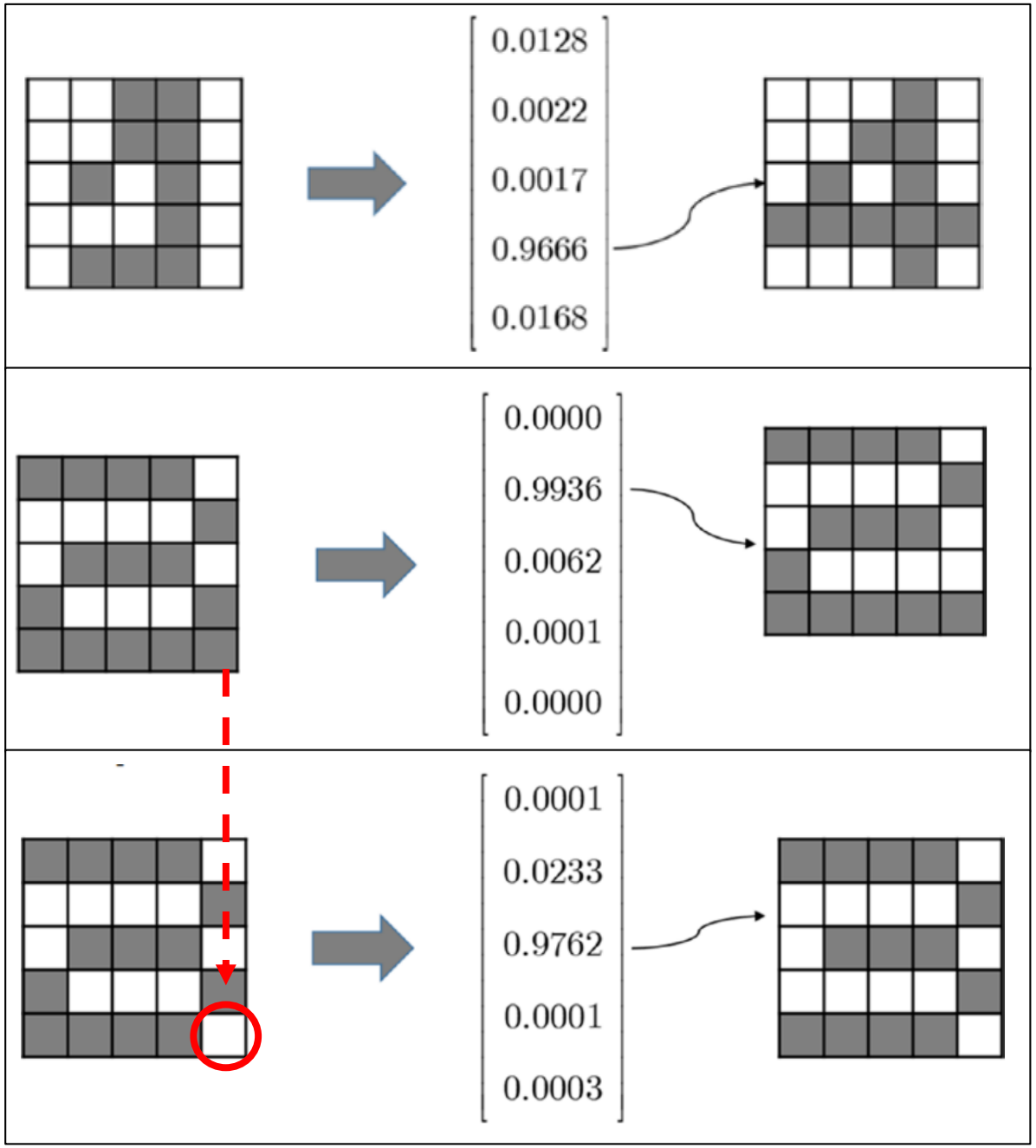
- Consider the slightly contaminated images and watch how the neural network responds to them.

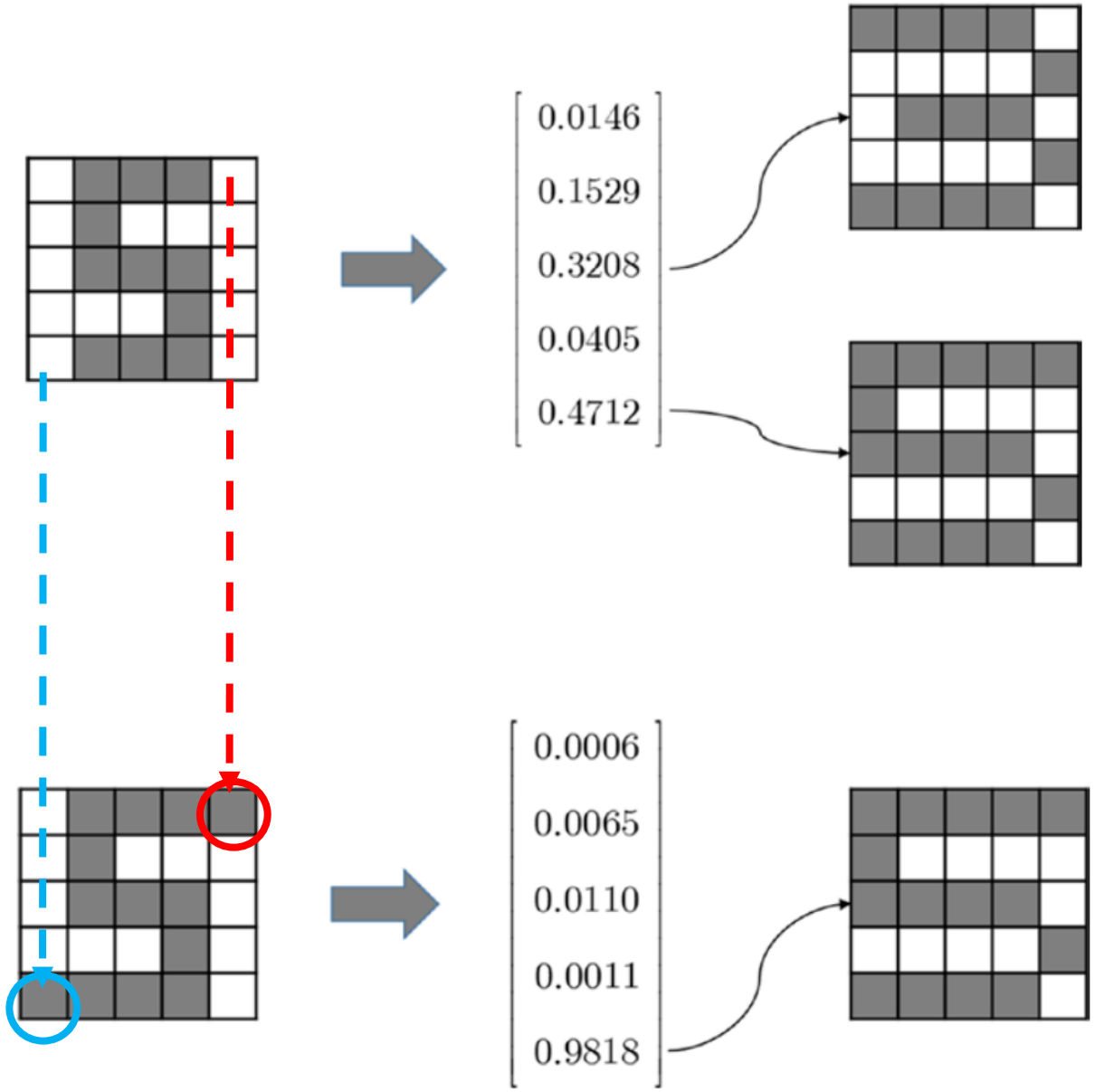


- This program starts with the execution of the TestMultiClass command and trains the neural network. This process yields the weight matrices W1 and W2.

```
clear all

TestMultiClass;                    % W1, W2
X  = zeros(5, 5, 5);

X(:, :, 1) = [ 0 0 1 1 0;
               0 0 1 1 0;
               0 1 0 1 0;
               0 0 0 1 0;
               0 1 1 1 0 ];

X(:, :, 2) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               1 0 0 0 1;
               1 1 1 1 1 ];

X(:, :, 3) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               1 0 0 0 1;
               1 1 1 1 0 ];
```

```
X(:, :, 4) = [ 0 1 1 1 0;
               0 1 0 0 0;
               0 1 1 1 0;
               0 0 0 1 0;
               0 1 1 1 0 ];

X(:, :, 5) = [ 0 1 1 1 1;
               0 1 0 0 0;
               0 1 1 1 0;
               0 0 0 1 0;
               1 1 1 1 0 ];

N = 5;                             % inference
for k = 1:N
  x  = reshape(X(:, :, k), 25, 1);
  v1 = W1*x;
  y1 = Sigmoid(v1);
  v  = W2*y1;
  y  = Softmax(v)
end
```





The neural network **should be trained to have more variety in the training data in order to improve its performance**.

Deep Learning

Phil Kim, MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence, 2017, chapter 5.

- **The deep neural network is the multi-layer neural network** that contains two or more hidden layers.

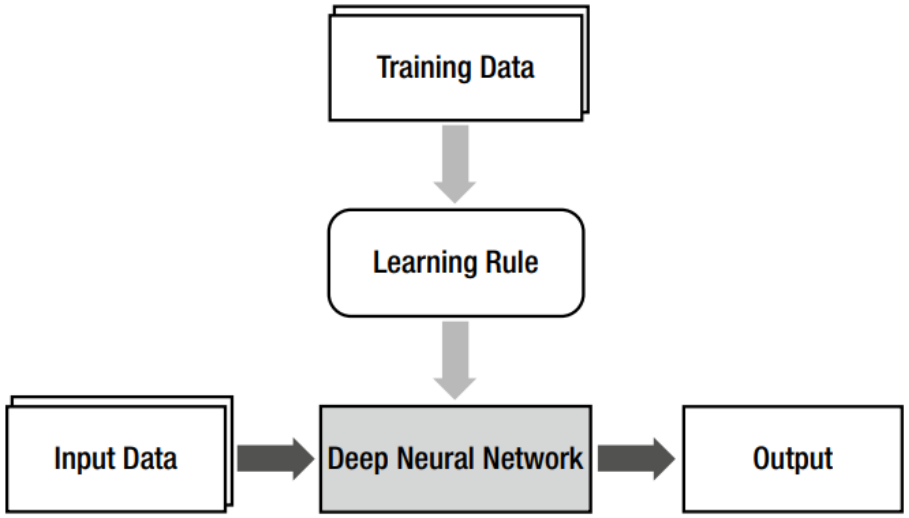
- Multi-layer neural network took 30 years to solve the problems of **learning rule** of the single-layer neural network, which was eventually solved by the **back-propagation algorithm**.

- The backpropagation training with the **additional hidden layers often resulted in poorer performance**. Deep Learning provided a solution to this problem.

Improvement of the Deep Neural Network

- The neural network with deeper layers yielded poorer performance was that the network was not properly trained.

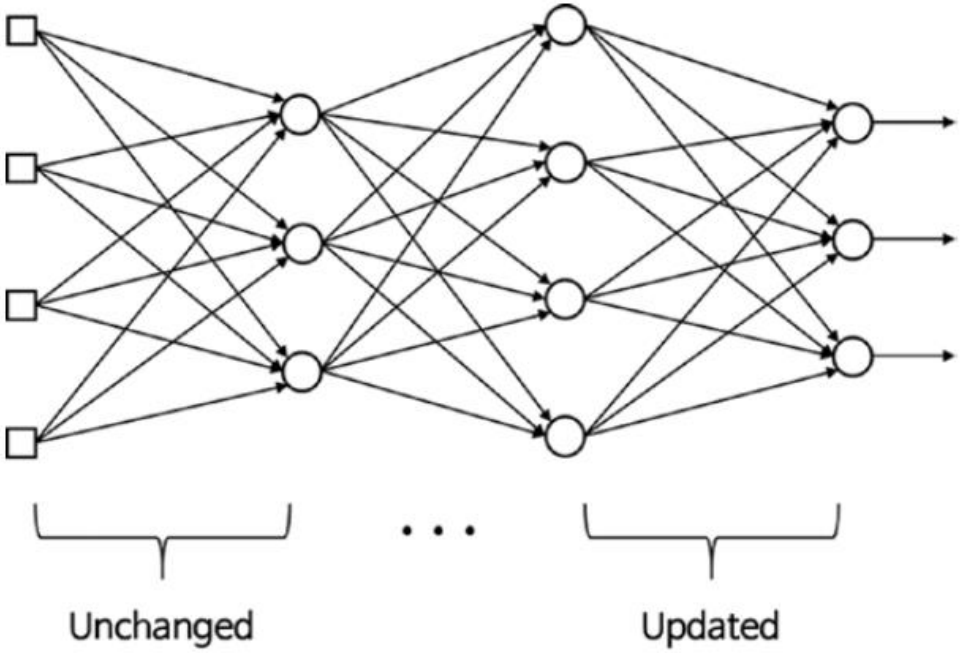
- The backpropagation algorithm experiences three difficulties in training deep neural network：
  - **Vanishing gradient**
  - **Overfitting**
  - **Computational load**



Vanishing Gradient

- The vanishing gradient in the training process occurs when the output error is more likely to **fail to reach the farther nodes**.

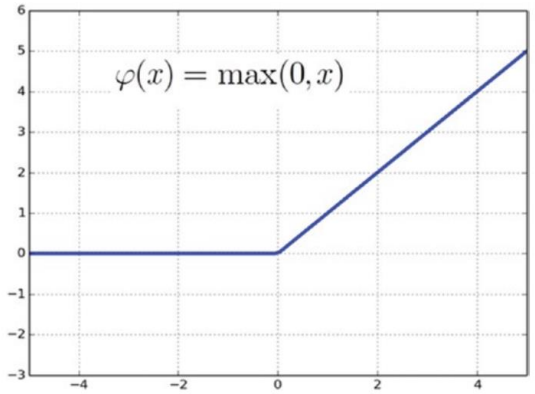
- As the error hardly reaches the first hidden layer, **the weight cannot be adjusted**.



Vanishing Gradient： ReLU

- A solution to the vanishing gradient is using the Rectified Linear Unit (**ReLU**) function as the activation function.

$$\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} = max(0, x)$$



- The sigmoid function limits the node's outputs to the unity, the ReLU function does not exert such limits and better transmit the error than the sigmoid function.

- We also need the derivative of the ReLU function.

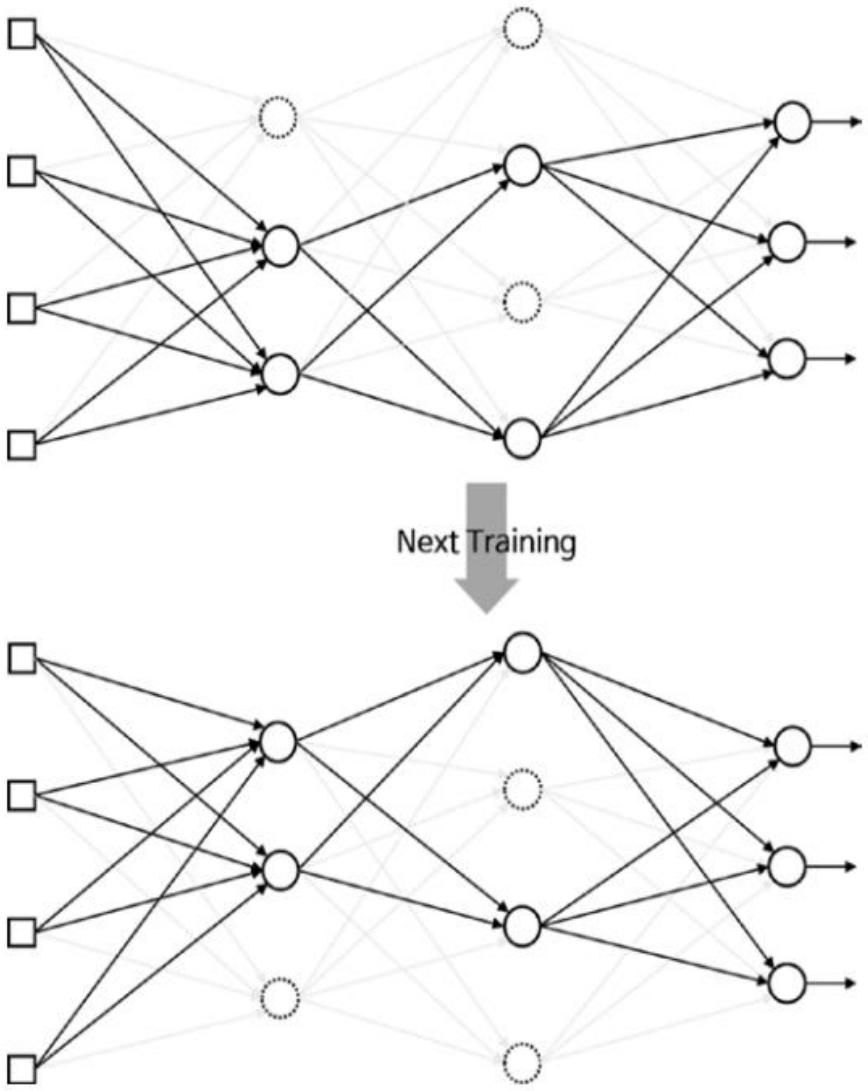
$$\varphi'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

Overfitting

- The reason that the deep neural network is especially vulnerable to overfitting is that the model becomes **more complicated as it includes more hidden layers**, and hence more weight.

Overfitting：Dropout

- **Train only some of the randomly selected nodes**
- 50% and 25% for hidden and input layers are dropped out
- Continuously alters the nodes and weights in the training process
- Use massive training data is very helpful to reduce potential bias



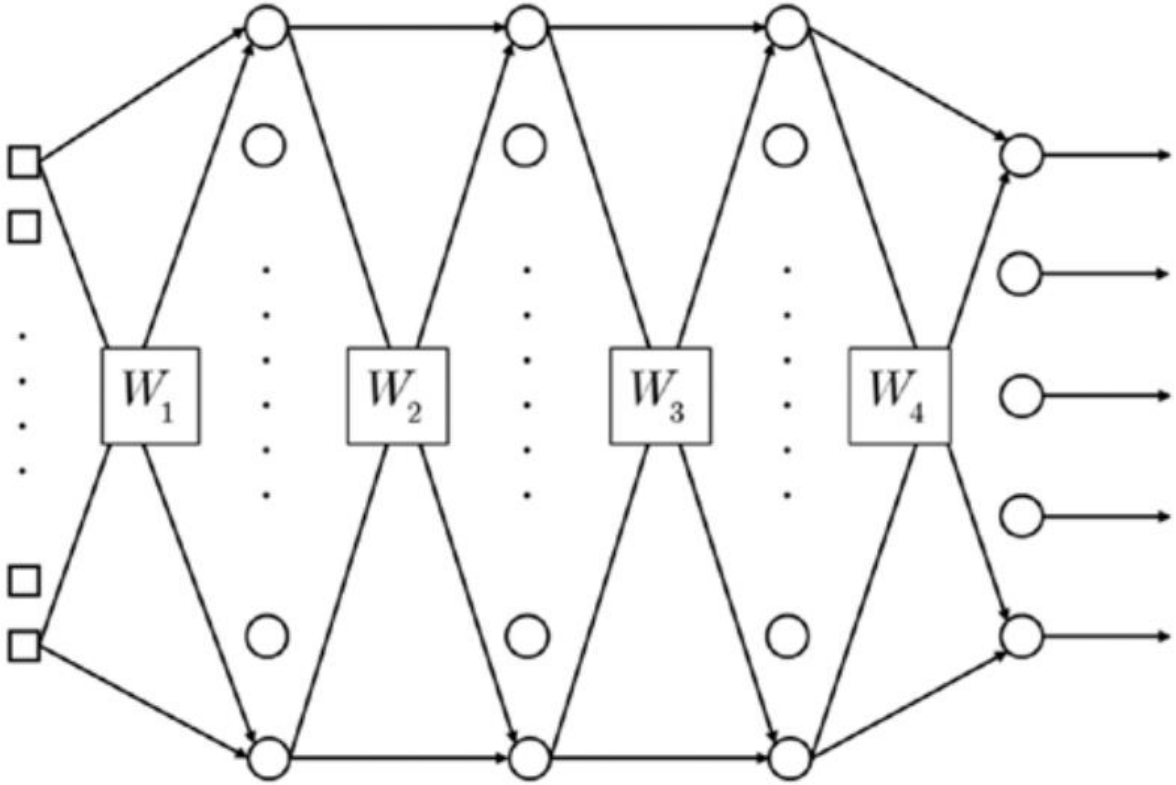
Computational Load

- The number of weights increases geometrically with the number of hidden layers, thus requiring more training data. This ultimately requires more calculations to be made.

- This trouble has been relieved to a considerable extent by the introduction of high-performance hardware, such as GPU, and algorithms, such as batch normalization.

Example: ReLU and Dropout

- The network has 25 input nodes
- Five output nodes for the five classes.
- Output nodes employ the softmax activation function.
- Three hidden layers, each hidden layer contains 20 nodes.



ReLU Function

- DeepReLU trains the deep neural network using the back-propagation algorithm. It takes the weights of the network and training data and returns the trained weights.

$$[W1, W2, W3, W4] = DeepReLU(W1, W2, W3, W4, X, D)$$

W1, W2, W3, and W4 are weight matrices of input - hidden1, hidden1 - hidden2, hidden2 - hidden3, and hidden3 - output layers.

X and D are input and correct output matrices of the training data.

- The process is identical to the previous training codes but the hidden nodes employ the ReLU in place of sigmoid.

```
function y = ReLU(x)
  y = max(0, x);
end
```

```
function [W1, W2, W3, W4] = DeepReLU(W1, W2, W3, W4, X, D)
  alpha = 0.01;

  N = 5;
  for k = 1:N
    x  = reshape(X(:, :, k), 25, 1);
    v1 = W1*x;
    y1 = ReLU(v1);

    v2 = W2*y1;
    y2 = ReLU(v2);

    v3 = W3*y2;
    y3 = ReLU(v3);

    v  = W4*y3;
    y  = Softmax(v);

    d     = D(k, :)';

    e     = d - y;
    delta = e;
```

```
    e3     = W4'*delta;
    delta3 = (v3 > 0).*e3;

    e2     = W3'*delta3;
    delta2 = (v2 > 0).*e2;

    e1     = W2'*delta2;
    delta1 = (v1 > 0).*e1;

    dW4 = alpha*delta*y3';
    W4  = W4 + dW4;

    dW3 = alpha*delta3*y2';
    W3  = W3 + dW3;

    dW2 = alpha*delta2*y1';
    W2  = W2 + dW2;

    dW1 = alpha*delta1*x';
    W1  = W1 + dW1;
  end
end
```

- Consider the back-propagation algorithm portion, which adjusts the weights using the back-propagation algorithm.

- This process starts from the delta of the output node, calculates the error of the hidden node, and uses it for the next error. It repeats the same steps through delta3, delta2, and delta1.

```
...
e     = d - y;
delta = e;

e3     = W4'*delta;
delta3 = (v3 > 0).*e3;

e2     = W3'*delta3;
delta2 = (v2 > 0).*e2;

e1     = W2'*delta2;
delta1 = (v1 > 0).*e1;
...
```

- The definition of the derivative of the ReLU function：

$$\varphi'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

- In the calculation of the delta of the third hidden layer, delta3, the derivative of the ReLU function is coded ：

$$(v3 > 0) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

- Following is TestDeepReLU.m file, which tests the DeepReLU function. This program calls the DeepReLU function and trains the network 10,000 times and displays the output.

```
clear all
X  = zeros(5, 5, 5);

X(:, :, 1) = [ 0 1 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 1 1 1 0 ];

X(:, :, 2) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               1 0 0 0 0;
               1 1 1 1 1 ];

X(:, :, 3) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0 ];
```

```
X(:, :, 4) = [ 0 0 0 1 0;
               0 0 1 1 0;
               0 1 0 1 0;
               1 1 1 1 1;
               0 0 0 1 0 ];

X(:, :, 5) = [ 1 1 1 1 1;
               1 0 0 0 0;
               1 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0 ];

D = [ 1 0 0 0 0;
      0 1 0 0 0;
      0 0 1 0 0;
      0 0 0 1 0;
      0 0 0 0 1 ];

W1 = 2*rand(20, 25) - 1;
W2 = 2*rand(20, 20) - 1;
W3 = 2*rand(20, 20) - 1;
W4 = 2*rand( 5, 20) - 1;
```

```
for epoch = 1:10000              % train
  [W1, W2, W3, W4] = DeepReLU(W1, W2, W3, W4, X, D);
end

N = 5;                           % inference
for k = 1:N
  x  = reshape(X(:, :, k), 25, 1);
  v1 = W1*x;
  y1 = ReLU(v1);

  v2 = W2*y1;
  y2 = ReLU(v2);

  v3 = W3*y2;
  y3 = ReLU(v3);

  v  = W4*y3;
  y  = Softmax(v)
end
```

Dropout

- The function DeepDropout trains the example using the back-propagation algorithm. It takes the neural network's weights and training data and returns the trained weights.

$$[W1, W2, W3, W4] = DeepDropout(W1, W2, W3, W4, X, D)$$

- This code imports the training data, calculates the weight updates (dW1, dW2, dW3, and dW4) using the delta rule, and adjusts the weight of the neural network.

```
function [W1, W2, W3, W4] = DeepDropout(W1, W2, W3, W4, X, D)
  alpha = 0.01;

  N = 5;
  for k = 1:N
    x  = reshape(X(:, :, k), 25, 1);
    v1 = W1*x;
    y1 = Sigmoid(v1);
    y1 = y1 .* Dropout(y1, 0.2);

    v2 = W2*y1;
    y2 = Sigmoid(v2);
    y2 = y2 .* Dropout(y2, 0.2);

    v3 = W3*y2;
    y3 = Sigmoid(v3);
    y3 = y3 .* Dropout(y3, 0.2);

    v  = W4*y3;
    y  = Softmax(v);

    d     = D(k, :)';

    e     = d - y;
    delta = e;

    e3     = W4'*delta;
    delta3 = y3.*(1-y3).*e3;

    e2     = W3'*delta3;
    delta2 = y2.*(1-y2).*e2;

    e1     = W2'*delta2;
    delta1 = y1.*(1-y1).*e1;

    dW4 = alpha*delta*y3';
    W4  = W4 + dW4;

    dW3 = alpha*delta3*y2';
    W3  = W3 + dW3;

    dW2 = alpha*delta2*y1';
    W2  = W2 + dW2;

    dW1 = alpha*delta1*x';
    W1  = W1 + dW1;
  end
end
```

- It differs from the previous ones in that once the output is calculated from the sigmoid activation function of the hidden node, the Dropout function modifies the final output of the node.

- For example, the output of the first hidden layer is calculated as：

$$y1 = Sigmoid(v1);$$
$$y1 = y1 .* Dropout(y1, 0.2);$$

- Executing these lines switches the outputs from 20% of the first hidden nodes to 0; it drops out 20% of the first hidden nodes.

- Example：

$$y1 = rand(6,1)$$
$$ym = Dropout(y1, 0.5)$$
$$y1 = y1 .* ym$$

```
function ym = Dropout(y, ratio)
    [m, n] = size(y);
    ym     = zeros(m, n);

    num     = round(m*n*(1-ratio));
    idx     = randperm(m*n, num);
    ym(idx) = 1 / (1-ratio);
end
```

$$y_1 = \begin{bmatrix} 0.5356 \\ 0.9537 \\ 0.5442 \\ 0.0821 \\ 0.3663 \\ 0.8509 \end{bmatrix} \quad ym = \begin{bmatrix} 2 \\ 2 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix} \quad y_1 * ym = \begin{bmatrix} 1.0712 \\ 1.9075 \\ 0 \\ 0 \\ 0 \\ 1.7017 \end{bmatrix}$$

- ym contains zeros for as many elements as the ratio and 1 / (1 - ratio) for the other elements to compensate for the loss of output due to the dropped elements

- This code is almost identical to the other test codes. The only difference is that it calls the DeepDropout function when it calculates the output of the trained network.

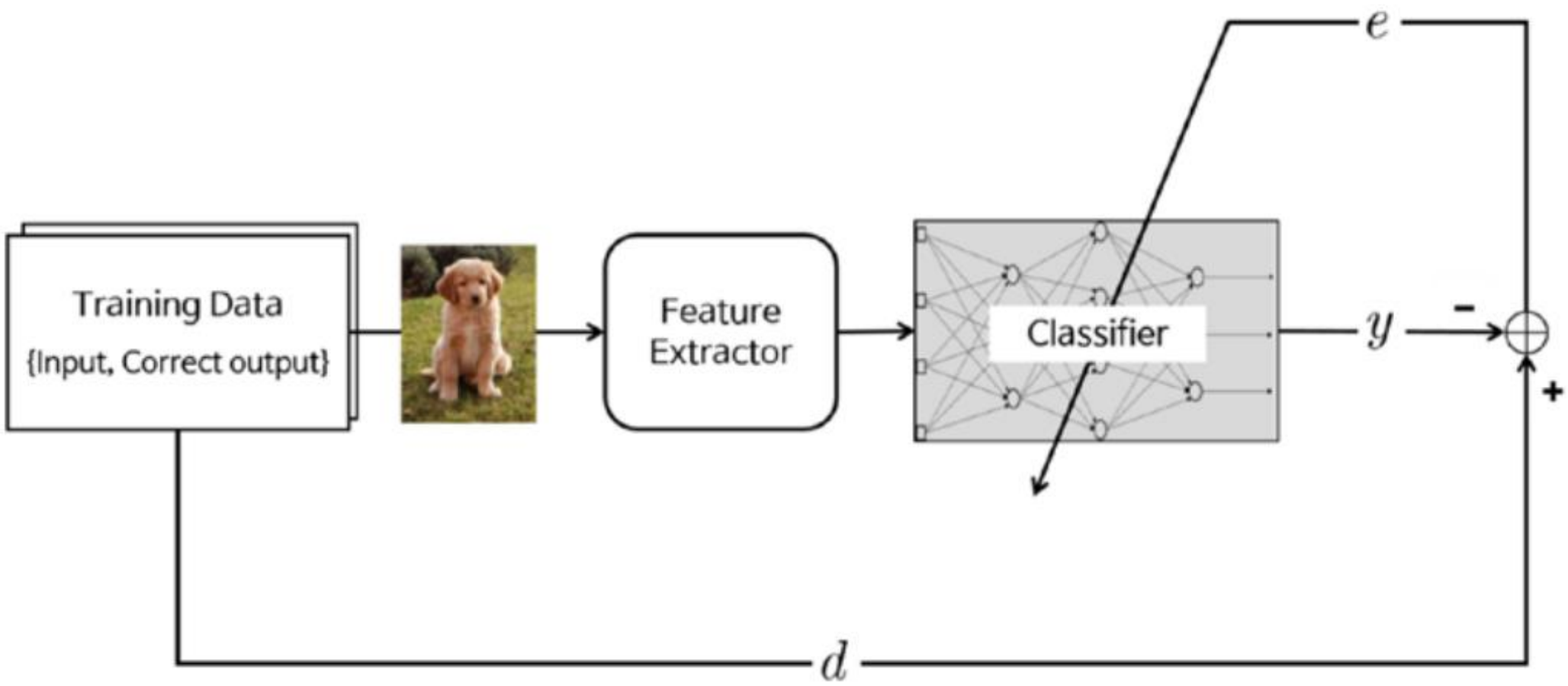
```
clear all

X  = zeros(5, 5, 5);

X(:, :, 1) = [ 0 1 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 1 1 1 0 ];

X(:, :, 2) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               1 0 0 0 0;
               1 1 1 1 1 ];

X(:, :, 3) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0 ];
```

```
X(:, :, 4) = [ 0 0 0 1 0;
               0 0 1 1 0;
               0 1 0 1 0;
               1 1 1 1 1;
               0 0 0 1 0 ];

X(:, :, 5) = [ 1 1 1 1 1;
               1 0 0 0 0;
               1 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0 ];

D = [ 1 0 0 0 0;
      0 1 0 0 0;
      0 0 1 0 0;
      0 0 0 1 0;
      0 0 0 0 1 ];

W1 = 2*rand(20, 25) - 1;
W2 = 2*rand(20, 20) - 1;
W3 = 2*rand(20, 20) - 1;
W4 = 2*rand( 5, 20) - 1;
```

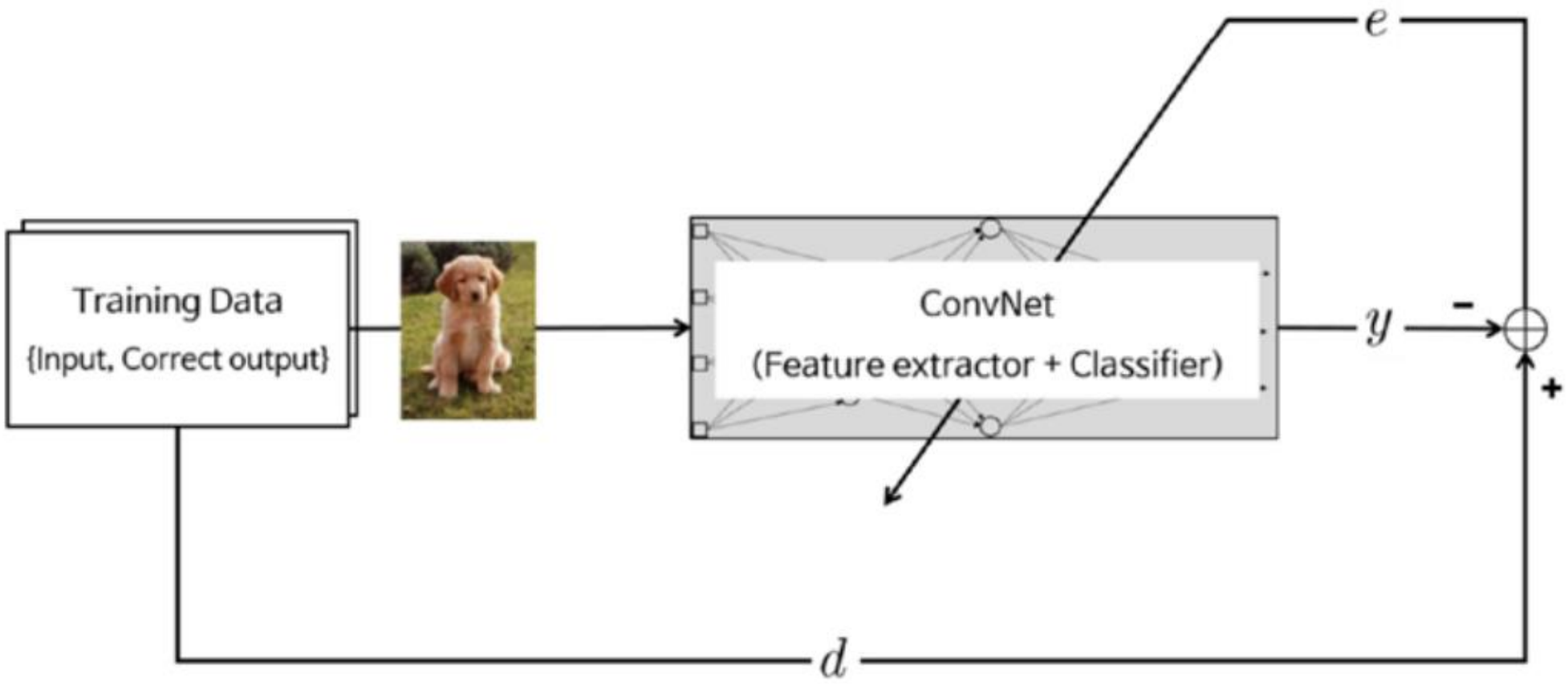
```
for epoch = 1:20000              % train
  [W1, W2, W3, W4] = DeepDropout(W1, W2, W3, W4, X, D);
end

N = 5;                           % inference
for k = 1:N
  x  = reshape(X(:, :, k), 25, 1);
  v1 = W1*x;
  y1 = Sigmoid(v1);

  v2 = W2*y1;
  y2 = Sigmoid(v2);

  v3 = W3*y2;
  y3 = Sigmoid(v3);

  v  = W4*y3;
  y  = Softmax(v)
end
```

Convolutional Neural Network (ConvNet)

Phil Kim, MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence, 2017, chapter 6.

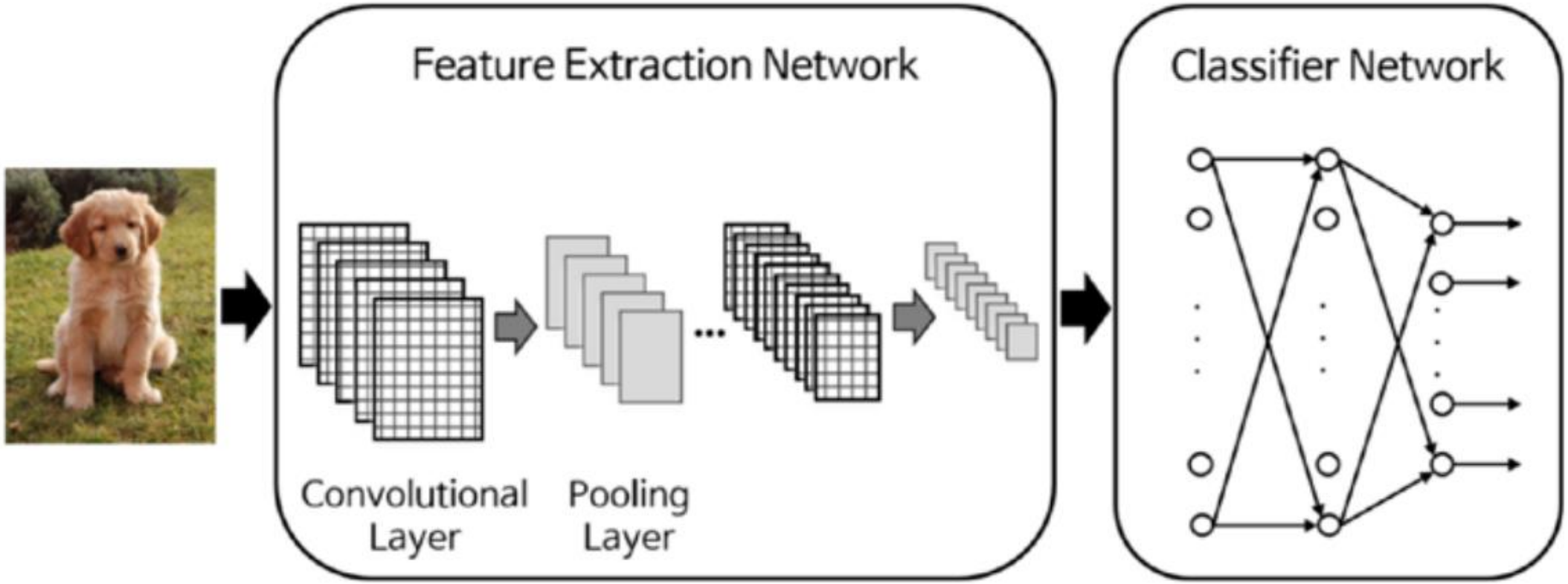


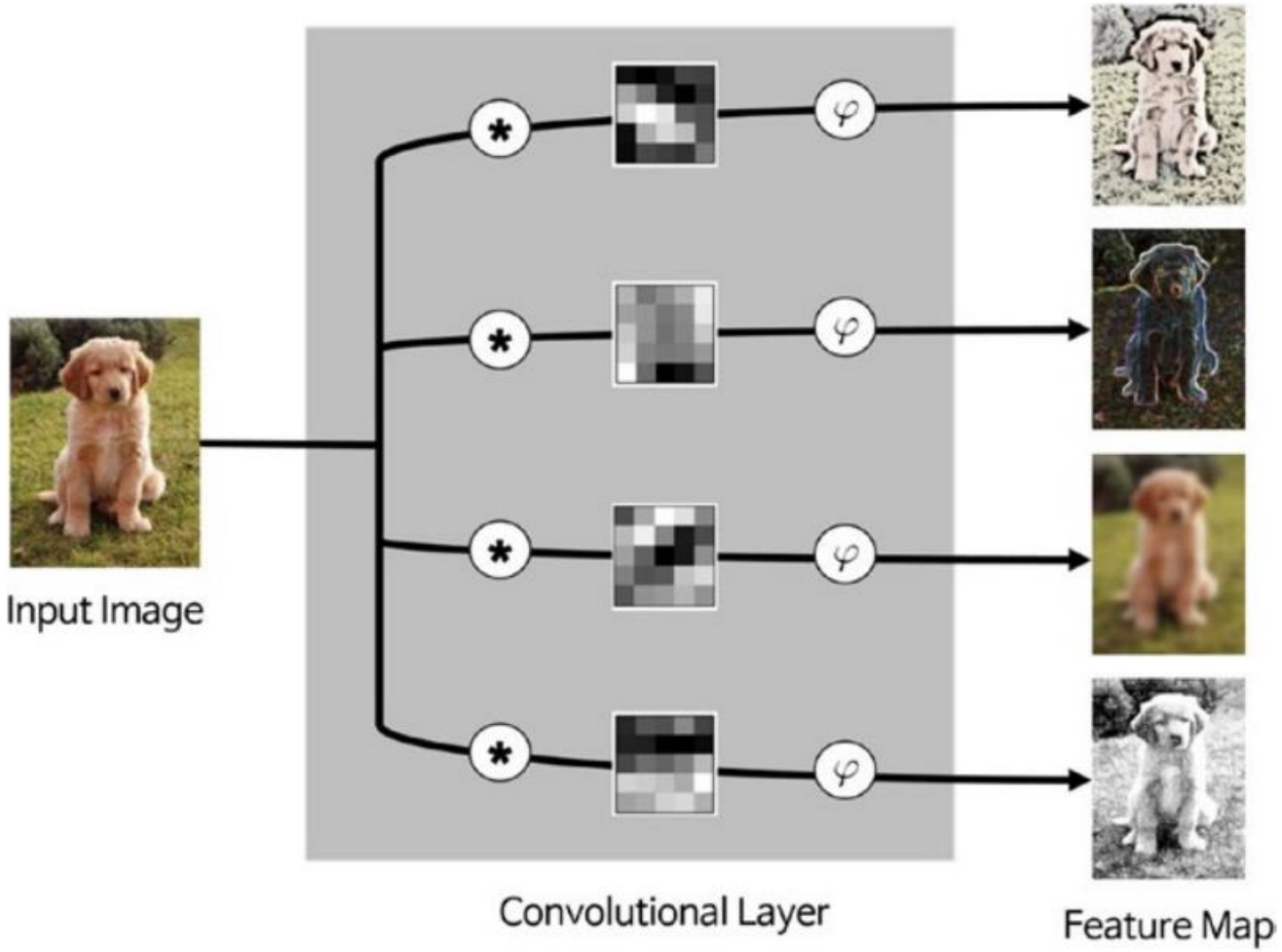
- Before ConvNet, the feature extractor has been designed by experts of specific areas, and was independent of Machine Learning.



- ConvNet **includes the feature extractor** in the training process rather than designing it manually.

- Feature extractor of ConvNet is composed of special kinds of neural networks, of which the weights are determined via the training process.

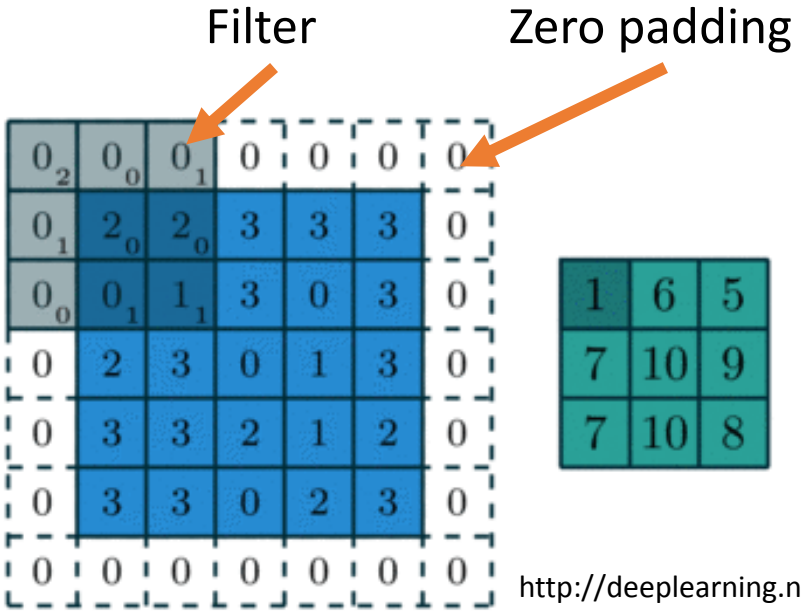

Feature Extraction Network
Classifier Network
Convolutional Layer
Pooling Layer



- Circled * mark (or $\otimes$) denotes the convolution operation, and the φ mark is the activation function.
- The square grayscale icons between these operators indicate the convolution filters.

Introduction : CNN – Convolutional Layer

- Kernel size : filter size(F)
- Stride : sliding length of filter per step(S)
- Padding : control the output feature maps' size
  - Same :
    output feature width = ceil (W / S)
    W : input feature width
    S : stride
  - Valid :
    output feature width = ceil ((W - F+1) / S)
    W : input feature width
    S : stride



http://deeplearning.net

Input Image

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

X

weight

1	0	1
0	1	0
1	0	1

429

= 18+51+35+121+204

Input Image

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

X

weight

1	0	1
0	1	0
1	0	1

429 686

= 51+78+109+204+244

Input Image

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

X

weight

1	0	1
0	1	0
1	0	1

429 686

633

= 35+154+225+15+204

Input Image

0	0	0	0	0	0	0
0	18	54	51	239	244	0
0	55	121	75	78	95	0
0	35	24	204	113	109	0
0	3	154	104	235	25	0
0	15	253	225	159	78	0
0	0	0	0	0	0	0

X

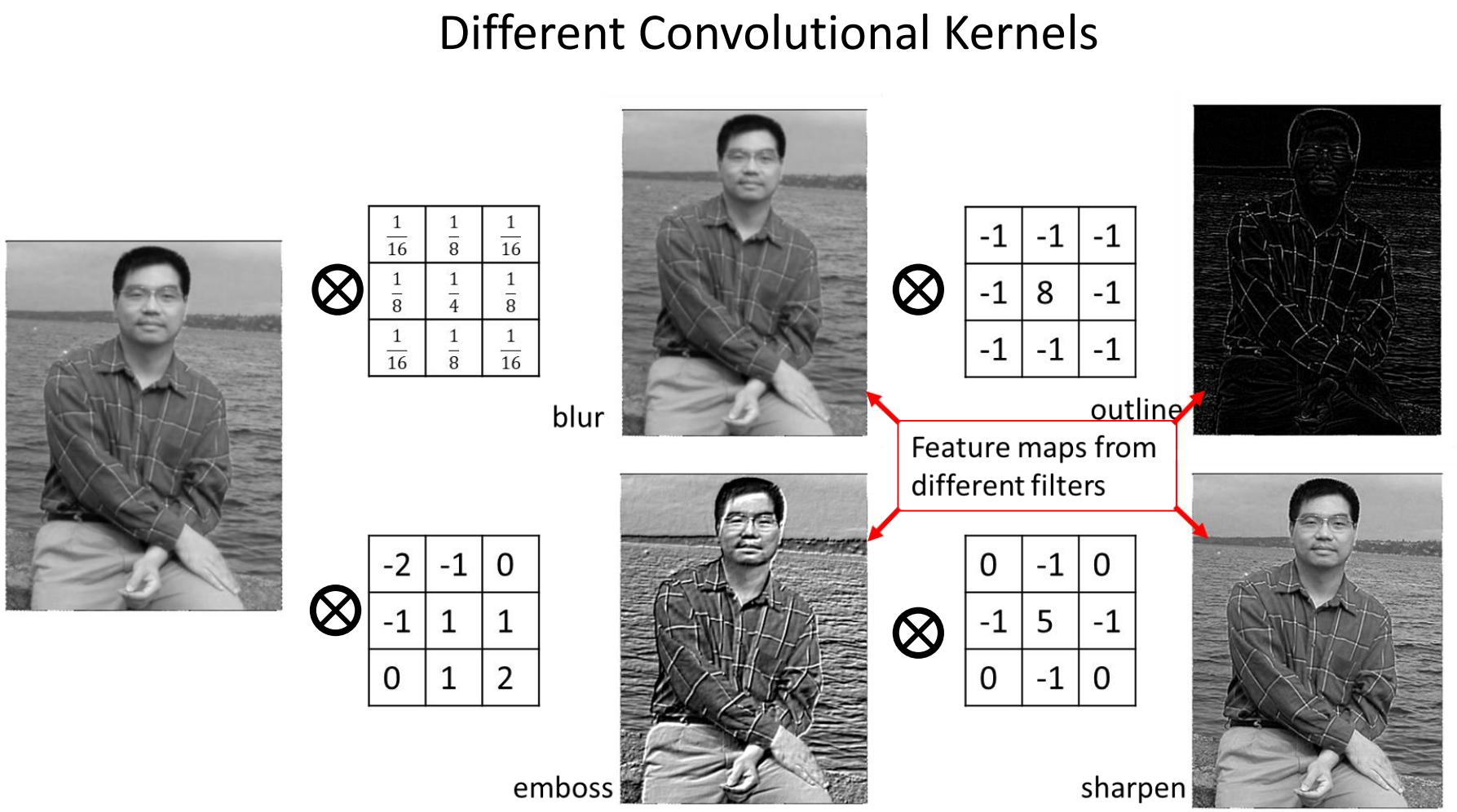
weight

1	0	1
0	1	0
1	0	1

139

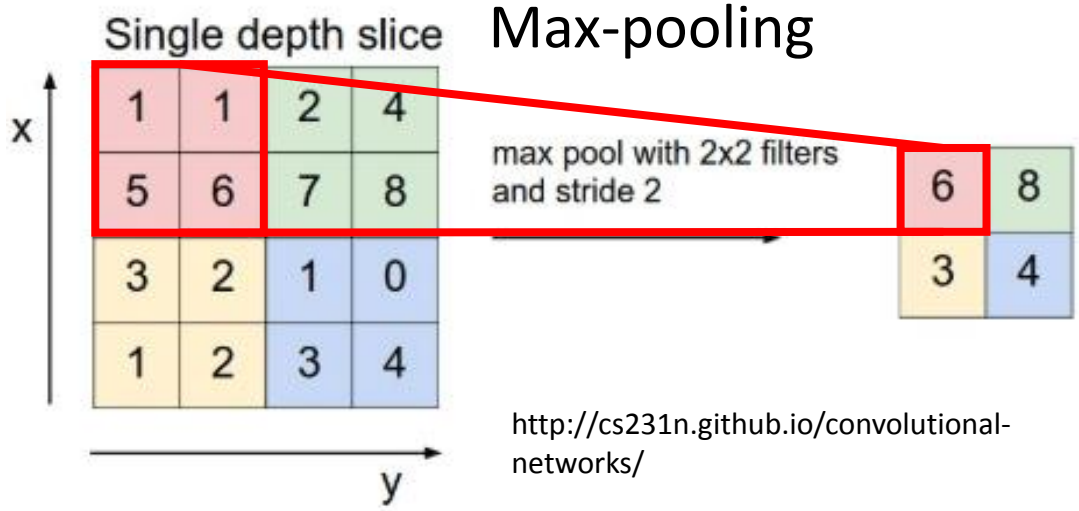
= 18+121

Introduction : CNN – Convolutional Layer

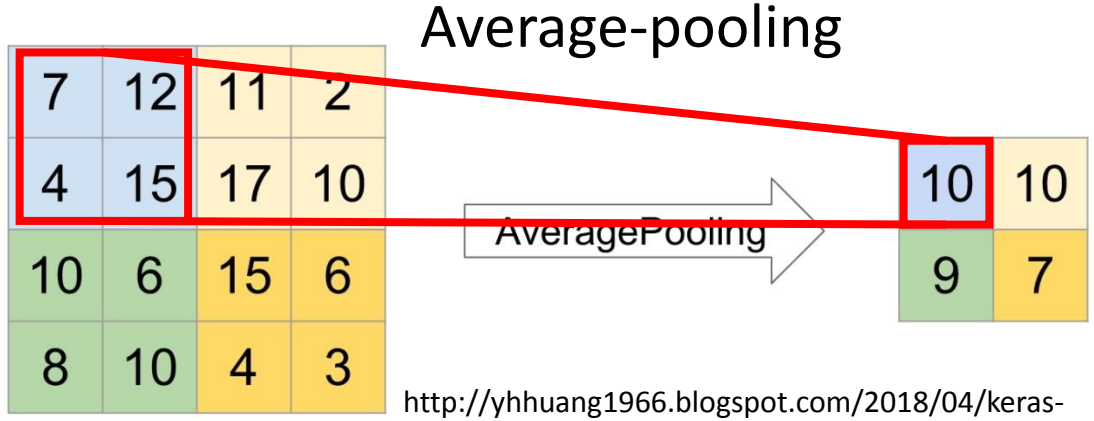


Introduction : CNN – Pooling Layer

- Kernel size : pooling kernel size
- Stride : usually equal to kernel size
- Padding : control the output feature maps' size



http://cs231n.github.io/convolutional-networks/



http://yhhuang1966.blogspot.com/2018/04/keras-cnn.html

- The pooling process is a type of convolution operation. The difference from the convolution layer is that the convolution filter is stationary, and the convolution areas do not overlap.

- The pooling layer compensates for eccentric and tilted objects to some extent. For example, the pooling layer can improve the recognition of a cat, which may be off-center in the input image.

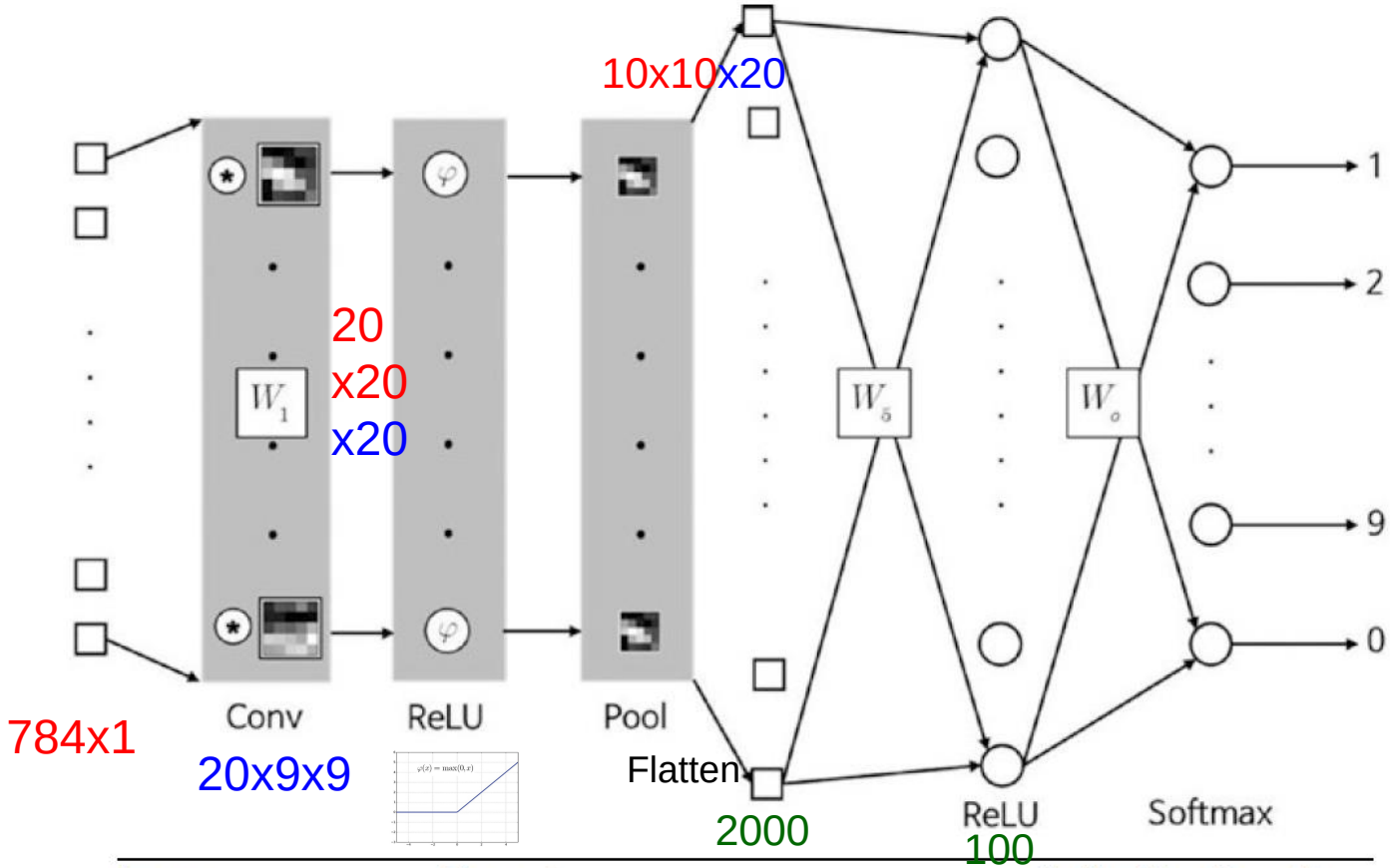
- As the pooling process reduces the image size, it is highly beneficial for relieving the computational load and preventing overfitting.

28-by-28 pixel black-and-white images from MNIST database

MNISTdatabase:70,000 images of handwritten numbers. In general, 60,000 images are used for training, and the remaining 10,000 images are used for the validation test

Phil Kim - MatLab Deep Learning with Machine Learning, Neural Networks and Artificial Intelligence 2017, Apress

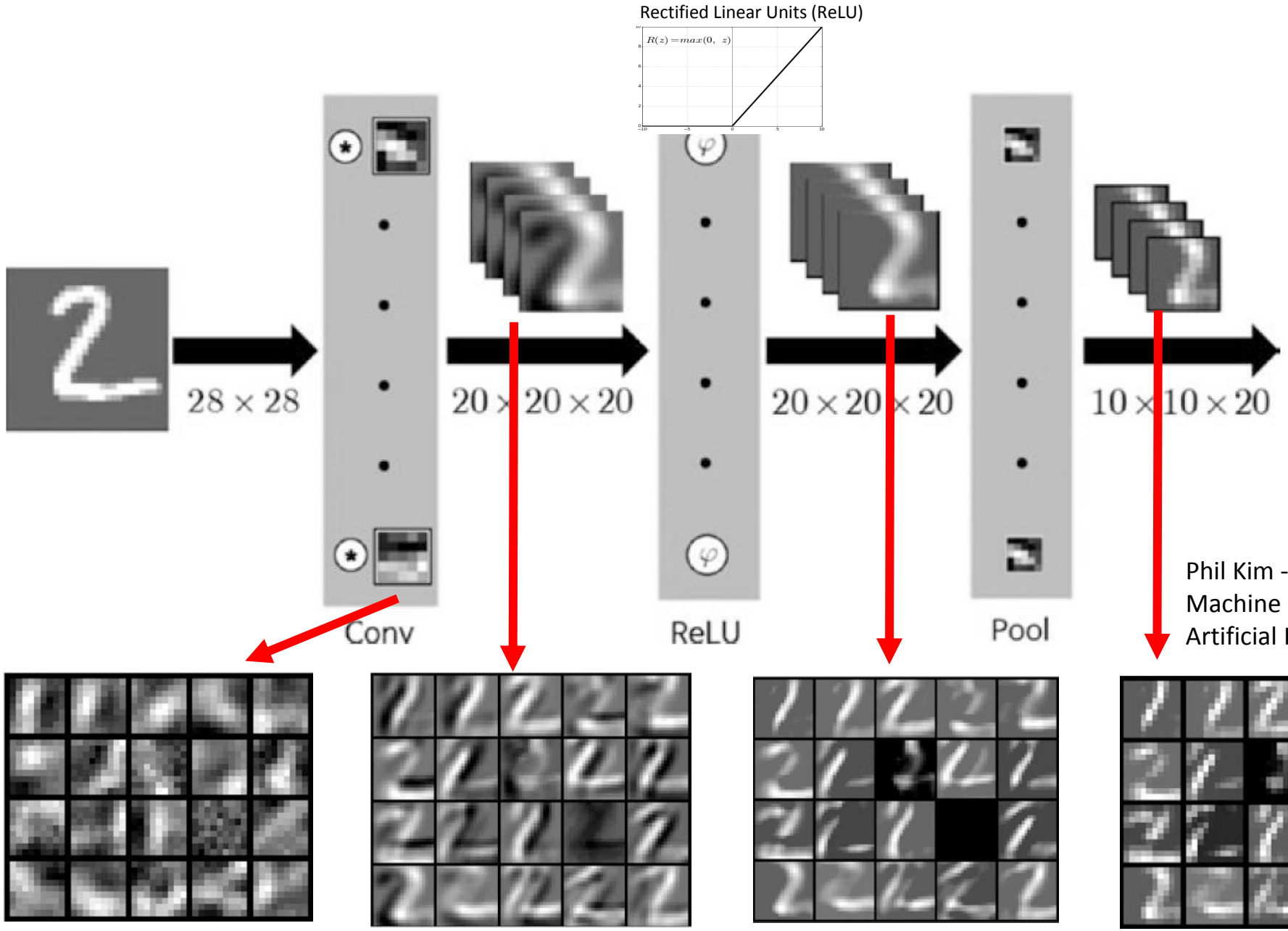
Architecture of convolutional neural network (CNN)



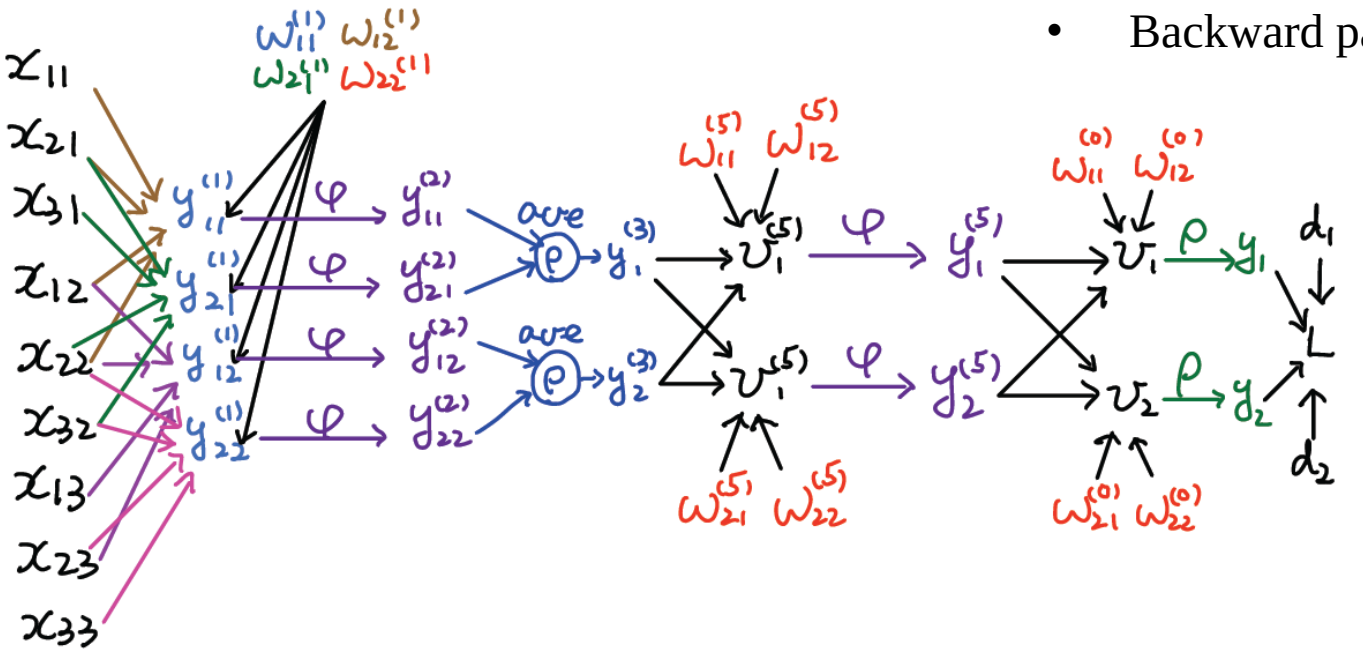
Layer	Remark	Activation Function
Input	28×28 nodes	-
Convolution	20 convolution filters (9×9)	ReLU
Pooling	1 mean pooling (2×2)	-
Hidden	100 nodes	ReLU
Output	10 nodes	Softmax

Phil Kim - MatLab Deep Learning with Machine Learning, Neural Networks and Artificial Intelligence 2017, Apress

While image passes through the convolution and pooling layers



Phil Kim - MatLab Deep Learning with Machine Learning, Neural Networks and Artificial Intelligence 2017, Apress



- Forward pass

$$y_{ij}^{(1)} = \sum_{n=1}^{2}\sum_{m=1}^{2} w_{3-m,3-n}^{(1)}\, x_{i-1+m,j-1+n}$$

$$y_{ij}^{(2)} = \varphi\left(y_{ij}^{(1)}\right), \varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \le 0 \end{cases} = max(0,x), \varphi'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \le 0 \end{cases}$$

$$y_k^{(3)} = \frac{1}{2}\sum_{i=1}^{2} y_{ik}^{(2)}$$

$$v_k^{(5)} = \sum_{j=1}^{2} w_{kj}^{(5)} y_j^{(3)}$$

$$y_k^{(5)} = \varphi\left(v_k^{(5)}\right), \varphi'\left(v_k^{(5)}\right) = \left(y_k^{(5)} > 0\right).$$

$$v_k = \sum_{j=1}^{2} w_{kj}^{(o)} y_j^{(5)}$$

$$y_k = \rho(v_k),\ \rho(x) = \frac{1}{1+e^{-x}}\ , \rho'(x) = (1-\rho(x))\rho(x)$$

$$L = \frac{1}{2}\sum_{k=1}^{2}(-d_k \ln(y_k) - (1-d_k)\ln(1-y_k))$$

- Backward pass:

$$\frac{\partial L}{\partial y_k} = -(d_k - y_k)/y_k(1-y_k)$$

$$\frac{\partial L}{\partial v_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial v_k} = -(d_k - y_k) \equiv -e_k \equiv -\delta_k$$

$$\frac{\partial L}{\partial w_{kj}^{(o)}} = \frac{\partial L}{\partial v_k}\frac{\partial v_k}{\partial w_{kj}^{(o)}} = -\delta_k y_j^{(5)}$$

$$\frac{\partial L}{\partial y_j^{(5)}} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial y_j^{(5)}} + \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial y_j^{(5)}}$$

$$= -(\delta_1 w_{1j}^{(o)} + \delta_2 w_{2j}^{(o)}) \equiv -e_j^{(5)}$$

$$\frac{\partial L}{\partial v_j^{(5)}} = \frac{\partial L}{\partial y_j^{(5)}}\frac{\partial y_j^{(5)}}{\partial v_j^{(5)}} = -e_j^{(5)}\varphi'\left(v_j^{(5)}\right) \equiv -\delta_j^{(5)}$$

$$\frac{\partial L}{\partial w_{ji}^{(5)}} = \frac{\partial L}{\partial v_j^{(5)}}\frac{\partial v_j^{(5)}}{\partial w_{ji}^{(5)}} = -\delta_j^{(5)} y_j^{(3)}$$

$$\frac{\partial L}{\partial y_j^{(3)}} = \frac{\partial L}{\partial v_1^{(5)}}\frac{\partial v_1^{(5)}}{\partial y_j^{(3)}} + \frac{\partial L}{\partial v_2^{(5)}}\frac{\partial v_2^{(5)}}{\partial y_j^{(3)}}$$

$$= -(\delta_1^{(5)} w_{1j}^{(5)} + \delta_2^{(5)} w_{2j}^{(5)}) \equiv -e_j^{(4)}$$

$$\frac{\partial L}{\partial y_{ij}^{(2)}} = \frac{\partial L}{\partial y_j^{(3)}}\frac{\partial y_j^{(3)}}{\partial y_{ij}^{(2)}} = -e_j^{(4)}\frac{1}{2}$$

$$\frac{\partial L}{\partial y_{ij}^{(1)}} = \frac{\partial L}{\partial y_{ij}^{(2)}}\frac{\partial y_{ij}^{(2)}}{\partial y_{ij}^{(1)}} = -\frac{1}{2}e_j^{(4)}\varphi'\left(y_{ij}^{(1)}\right) \equiv -\delta_{ij}^{(2)}$$

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \sum_{j=1}^{2}\sum_{i=1}^{2}\frac{\partial L}{\partial y_{ij}^{(1)}}\frac{\partial y_{ij}^{(1)}}{\partial w_{ij}^{(1)}}$$

$$= -\sum_{n=1}^{2}\sum_{m=1}^{2}\delta_{mn}^{(2)} x_{i-1+m,j-1+n}$$

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \circledast \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} *. \begin{bmatrix} w_{22} & w_{21} \\ w_{12} & w_{11} \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$$

$$\begin{aligned} y_{11} &= w_{22}x_{11} + w_{21}x_{12} + w_{12}x_{21} + w_{11}x_{22} \\ y_{12} &= w_{22}x_{12} + w_{21}x_{13} + w_{12}x_{22} + w_{11}x_{23} \\ y_{21} &= w_{22}x_{21} + w_{21}x_{22} + w_{12}x_{31} + w_{11}x_{32} \\ y_{22} &= w_{22}x_{22} + w_{21}x_{23} + w_{12}x_{32} + w_{11}x_{33} \end{aligned}$$

$$\Rightarrow y_{ij} = \sum_{n=1}^{2}\sum_{m=1}^{2} w_{3-m,3-n}\, x_{i-1+m,j-1+n}$$

Assume $\frac{\partial L}{\partial y_{ij}} \equiv \delta_{ij}$ is known, L is cost function, the update of w_{22}, w_{21}, w_{12} and w_{11} need

$$\begin{aligned} \frac{\partial L}{\partial w_{22}} &= \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial w_{22}} \\ &= \delta_{11}x_{11} + \delta_{12}x_{12} + \delta_{21}x_{21} + \delta_{22}x_{22} \end{aligned}$$

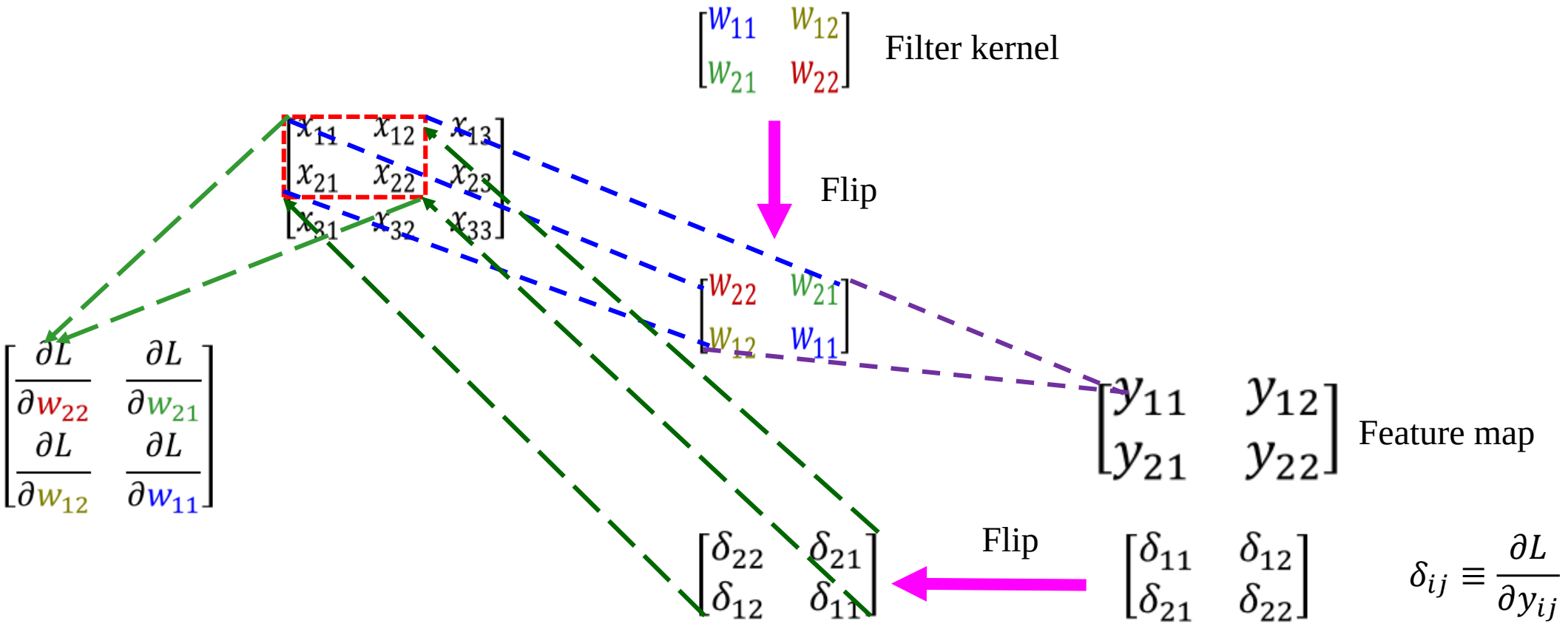
$$\begin{aligned} \frac{\partial L}{\partial w_{12}} &= \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial w_{12}} \\ &= \delta_{11}x_{21} + \delta_{12}x_{22} + \delta_{21}x_{31} + \delta_{22}x_{32} \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial w_{21}} &= \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial w_{21}} \\ &= \delta_{11}x_{12} + \delta_{12}x_{13} + \delta_{21}x_{22} + \delta_{22}x_{23} \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial w_{11}} &= \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial w_{11}} \\ &= \delta_{11}x_{22} + \delta_{12}x_{23} + \delta_{21}x_{32} + \delta_{22}x_{33} \end{aligned}$$

Back-propagation with convolution

$$\begin{bmatrix} \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{21}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{11}} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} *. \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \circledast \begin{bmatrix} \delta_{22} & \delta_{21} \\ \delta_{12} & \delta_{11} \end{bmatrix}$$



$$\delta_{ij} \equiv \frac{\partial L}{\partial y_{ij}}$$

Back-propagation with average pooling operation

Feature map at layer L

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$$

forward

$$\Rightarrow$$

average pooling

Feature map at layer L+1

$$\begin{bmatrix} x_{21} \rightarrow y_{11}^{(3)} & x_{23} \rightarrow y_{12}^{(3)} \\ x_{32} \rightarrow y_{21}^{(3)} & x_{33} \rightarrow y_{22}^{(3)} \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{4}\frac{\partial L}{\partial y_{11}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{11}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{12}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{12}^{(3)}} \\ \frac{1}{4}\frac{\partial L}{\partial y_{11}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{11}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{12}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{12}^{(3)}} \\ \frac{1}{4}\frac{\partial L}{\partial y_{21}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{21}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{22}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{22}^{(3)}} \\ \frac{1}{4}\frac{\partial L}{\partial y_{21}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{21}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{22}^{(3)}} & \frac{1}{4}\frac{\partial L}{\partial y_{22}^{(3)}} \end{bmatrix}$$

backward

$$\Leftarrow$$

$$\begin{bmatrix} \frac{\partial L}{\partial y_{11}^{(3)}} & \frac{\partial L}{\partial y_{12}^{(3)}} \\ \frac{\partial L}{\partial y_{21}^{(3)}} & \frac{\partial L}{\partial y_{22}^{(3)}} \end{bmatrix}$$

Back-propagation with max pooling operation

Feature map at layer L

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$$

forward

$$\Rightarrow$$

max pooling

Feature map at layer L+1

$$\begin{bmatrix} x_{21} \rightarrow y_{11}^{(3)} & x_{23} \rightarrow y_{12}^{(3)} \\ x_{32} \rightarrow y_{21}^{(3)} & x_{33} \rightarrow y_{22}^{(3)} \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \dfrac{\partial L}{\partial y_{11}^{(3)}} & 0 & \dfrac{\partial L}{\partial y_{12}^{(3)}} & 0 \\ 0 & \dfrac{\partial L}{\partial y_{21}^{(3)}} & \dfrac{\partial L}{\partial y_{22}^{(3)}} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

backward

$$\Leftarrow$$

$$\begin{bmatrix} \dfrac{\partial L}{\partial y_{11}^{(3)}} & \dfrac{\partial L}{\partial y_{12}^{(3)}} \\ \dfrac{\partial L}{\partial y_{21}^{(3)}} & \dfrac{\partial L}{\partial y_{22}^{(3)}} \end{bmatrix}$$

- MnistConv.m trains the network using the back-propagation algorithm

[W1, W5, Wo] = MnistConv(W1, W5, Wo, X, D)

W1: convolution filter matrix

W5: pooling-hidden layer weight matrix

Wo : hidden- output layer weight matrix

X: training input data

D: correct output

MnistConv.m

```
function [W1, W5, Wo] = MnistConv(W1, W5, Wo, X, D)
%
alpha = 0.01;
beta  = 0.95;

momentum1 = zeros(size(W1));
momentum5 = zeros(size(W5));
momentumo = zeros(size(Wo));

N = length(D);

bsize = 100;
blist = 1:bsize:(N-bsize+1);

% One epoch loop
%
for batch = 1:length(blist)
  dW1 = zeros(size(W1));
  dW5 = zeros(size(W5));
  dWo = zeros(size(Wo));
```

```
% Mini-batch loop
 %
 begin = blist(batch);
 for k = begin:begin+bsize-1
   % Forward pass = inference
   %
   x  = X(:, :, k);            % Input,  28*28
   y1 = Conv(x, W1);           % W1:9*9*20, y1:20*20*20
   y2 = ReLU(y1);              % y2:20*20*20
   y3 = Pool(y2);              % Pooling, y3:10*10*20
   y4 = reshape(y3, [], 1);    % Flattening, y4:2000*1
   v5 = W5*y4;                 % W5:100*2000, v5:100*1
   y5 = ReLU(v5);              % y5:100*1
   v  = Wo*y5;                 % Wo:10*100, v:10x1
   y  = Softmax(v);            % y:10*1

   % One-hot encoding
   %
   d = zeros(10, 1);
   d(sub2ind(size(d), D(k), 1)) = 1;
```

The number of batches, bsize, is set to be 100. As we have a total 8,000 training data points, the weights are adjusted 80 (=8,000/100) times for every epoch. The variable blist contains the location of the first training data point to be brought into the minibatch. Starting from this location, the code brings in 100 data points and forms the training data for the minibatch.

blist = [1, 101, 201, 301, ..., 7801, 7901]

MnistConv.m (continued)

```
% Backpropagation
   %
   e      = d - y;                    % Output layer 10*1
   delta  = e;                        % 10*1

   e5     = Wo' * delta;              % Hidden(ReLU) layer (100*10)*10*1=100*1
   delta5 = (y5 > 0) .* e5;           % (100*1).*(100*1)=100*1

   e4     = W5' * delta5;           % Pooling layer (2000*100)*100*1=2000*1

   e3     = reshape(e4, size(y3)); % 2000*1 → 10*10*20

   e2 = zeros(size(y2));             % 20*20*20
   W3 = ones(size(y2)) / (2*2);      % 20*20*20
   for c = 1:20
     e2(:, :, c) = kron(e3(:, :, c), ones([2 2])) .* W3(:, :, c); % 20*20*20
   end

   delta2 = (y2 > 0) .* e2;           % ReLU layer (20*20*20).*(20*20*20)

   delta1_x = zeros(size(W1));        % Convolutional layer 9*9*20
   for c = 1:20
     delta1_x(:, :, c) = conv2(x(:, :), rot90(delta2(:, :, c), 2), 'valid');
   end

   dW1 = dW1 + delta1_x;     % 9*9*20
   dW5 = dW5 + delta5*y4';   % delta5*y4' :(100*1)*(1*2000)=100*2000
   dWo = dWo + delta *y5';   % delta *y5' :(10*1)*(1*100)=10*100
 end
```

```
   % Update weights
   %
   dW1 = dW1 / bsize;
   dW5 = dW5 / bsize;
   dWo = dWo / bsize;

   momentum1 = alpha*dW1 + beta*momentum1;
   W1        = W1 + momentum1;

   momentum5 = alpha*dW5 + beta*momentum5;
   W5        = W5 + momentum5;

   momentumo = alpha*dWo + beta*momentumo;
   Wo        = Wo + momentumo;
end

end
```

- The following is the back-propagation from the output layer to the subsequent layer to the pooling layer.

```
...
e      = d - y;
delta  = e;

e5     = Wo' * delta;
delta5 = e5 .* (y5> 0);

e4     = W5' * delta5;
e3     = reshape(e4, size(y3));
...
```

$$\frac{\partial L}{\partial y_k} = -(d_k - y_k)/y_k(1 - y_k)$$

$$\frac{\partial L}{\partial v_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial v_k} = -(d_k - y_k) \equiv -e_k \equiv -\delta_k$$

$$\frac{\partial L}{\partial w_{kj}^{(o)}} = \frac{\partial L}{\partial v_k}\frac{\partial v_k}{\partial w_{kj}^{(o)}} = -\delta_k y_j^{(5)}$$

$$\frac{\partial L}{\partial y_j^{(5)}} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial y_j^{(5)}} + \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial y_j^{(5)}}$$

$$= -(\delta_1 w_{1j}^{(o)} + \delta_2 w_{2j}^{(o)}) \equiv -e_j^{(5)}$$

$$\frac{\partial L}{\partial v_j^{(5)}} = \frac{\partial L}{\partial y_j^{(5)}}\frac{\partial y_j^{(5)}}{\partial v_j^{(5)}} = -e_j^{(5)}\varphi'\left(v_j^{(5)}\right) = -e_j^{(5)}\left(y_k^{(5)} > 0\right). \equiv -\delta_j^{(5)}$$

$$\frac{\partial L}{\partial w_{ji}^{(5)}} = \frac{\partial L}{\partial v_j^{(5)}}\frac{\partial v_j^{(5)}}{\partial w_{ji}^{(5)}} = -\delta_j^{(5)} y_j^{(3)}$$

$$\frac{\partial L}{\partial y_j^{(3)}} = \frac{\partial L}{\partial v_1^{(5)}}\frac{\partial v_1^{(5)}}{\partial y_j^{(3)}} + \frac{\partial L}{\partial v_2^{(5)}}\frac{\partial v_2^{(5)}}{\partial y_j^{(3)}}$$

$$= -(\delta_1^{(5)} w_{1j}^{(5)} + \delta_2^{(5)} w_{2j}^{(5)}) \equiv -e_j^{(4)}$$

- Two more layers to go: the pooling and convolution layers. The following shows the back-propagation that passes through the pooling layer-ReLU-convolution layer.

```
...
e2 = zeros(size(y2));                % Pooling
W3 = ones(size(y2)) / (2*2);
for c = 1:20
  e2(:, :, c) = kron(e3(:, :, c), ones([2 2])) .* W3(:, :, c);
end

delta2 = (y2 > 0) .* e2;

delta1_x = zeros(size(W1));
for c = 1:20
  delta1_x(:, :, c) = conv2(x(:, :), rot90(delta2(:, :, c), 2),
'valid');
end
...
```

$$\frac{\partial L}{\partial y_{ij}^{(2)}} = \frac{\partial L}{\partial y_j^{(3)}} \frac{\partial y_j^{(3)}}{\partial y_{ij}^{(2)}} = -e_j^{(4)} \frac{1}{2}$$

$$\frac{\partial L}{\partial y_{ij}^{(1)}} = \frac{\partial L}{\partial y_{ij}^{(2)}} \frac{\partial y_{ij}^{(2)}}{\partial y_{ij}^{(1)}} = -\frac{1}{2} e_j^{(4)} \varphi'\left(y_{ij}^{(1)}\right) \equiv -\delta_{ij}^{(2)}$$

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \sum_{j=1}^{2} \sum_{i=1}^{2} \frac{\partial L}{\partial y_{ij}^{(1)}} \frac{\partial y_{ij}^{(1)}}{\partial w_{ij}^{(1)}}$$

$$= -\sum_{n=1}^{2} \sum_{m=1}^{2} \delta_{mn}^{(2)} x_{i-1+m, j-1+n}$$

- MnistConv.m calls Conv.m, which takes the input image and the convolution filter matrix and returns the feature maps.

```
function y = Conv(x, W)
%
%

[wrow, wcol, numFilters] = size(W);
[xrow, xcol, ~         ] = size(x);

yrow = xrow - wrow + 1;
ycol = xcol - wcol + 1;

y = zeros(yrow, ycol, numFilters);

for k = 1:numFilters
  filter = W(:, :, k);
  filter = rot90(squeeze(filter), 2);
  y(:, :, k) = conv2(x, filter, 'valid');
end

end
```

- MnistConv.m also calls Pool.m, which takes the feature map and returns the image after the 2x2 mean pooling process.

```
function y = Pool(x)
%
% 2x2 mean pooling
%
[xrow, xcol, numFilters] = size(x);

y = zeros(xrow/2, xcol/2, numFilters);
for k = 1:numFilters
  filter = ones(2) / (2*2);    % for mean
  image  = conv2(x(:, :, k), filter, 'valid');

  y(:, :, k) = image(1:2:end, 1:2:end);
end

end
```

- This code calls the two-dimensional convolution function, conv2, just as the function Conv does. This is because the pooling process is a type of a convolution operation.

- The mean pooling of this example is implemented using the convolution operation with the following filter:

$$W = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

- The filter of the pooling layer is predefined, while that of the convolution layer is determined through training.

TestMnistConv.m

```m
clear all

Images = loadMNISTImages('./MNIST/t10k-images.idx3-ubyte');
Images = reshape(Images, 28, 28, []);
Labels = loadMNISTLabels('./MNIST/t10k-labels.idx1-ubyte');
Labels(Labels == 0) = 10;    % 0 --> 10

rng(1);

% Learning
%
W1 = 1e-2*randn([9 9 20]);
W5 = (2*rand(100, 2000) - 1) * sqrt(6) / sqrt(360 + 2000);
Wo = (2*rand( 10,  100) - 1) * sqrt(6) / sqrt( 10 +  100);

X = Images(:, :, 1:8000);
D = Labels(1:8000);

for epoch = 1:3
  epoch
  [W1, W5, Wo] = MnistConv(W1, W5, Wo, X, D);
end

save('MnistConv.mat');
```

```m
% Test
%
X = Images(:, :, 8001:10000);
D = Labels(8001:10000);

acc = 0;
N   = length(D);
for k = 1:N
  x = X(:, :, k);                % Input,          28x28

  y1 = Conv(x, W1);              % Convolution,  20x20x20
  y2 = ReLU(y1);                 %
  y3 = Pool(y2);                 % Pool,         10x10x20
  y4 = reshape(y3, [], 1);       %                  2000
  v5 = W5*y4;                    % ReLU,             360
  y5 = ReLU(v5);                 %
  v  = Wo*y5;                    % Softmax,           10
  y  = Softmax(v);               %

  [~, i] = max(y);
  if i == D(k)
    acc = acc + 1;
  end
end

acc = acc / N;
fprintf('Accuracy is %f\n', acc);
```

2,000 test data points

convert the 10x1 output into a digit to compare with the correct output

accuracy 94.65%

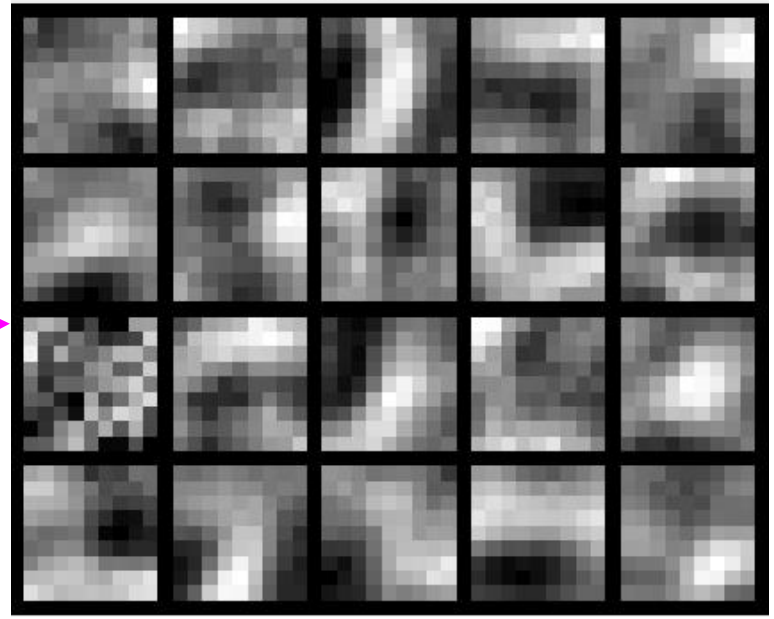
PlotFeatures.m

```
clear all

load('MnistConv.mat')

k  = 2;
x  = X(:, :, k);
y1 = Conv(x, W1);                % Convolution,  20x20x20
y2 = ReLU(y1);                   %
y3 = Pool(y2);                   % Pool,         10x10x20
y4 = reshape(y3, [], 1);         %                  2000
v5 = W5*y4;                      % ReLU,             360
y5 = ReLU(v5);                   %
v  = Wo*y5;                      % Softmax,           10
y  = Softmax(v);                 %

figure;
display_network(x(:));
title('Input Image')

convFilters = zeros(9*9, 20);
for i = 1:20
  filter           = W1(:, :, i);
  convFilters(:, i) = filter(:);
end
figure
display_network(convFilters);
title('Convolution Filters')
```

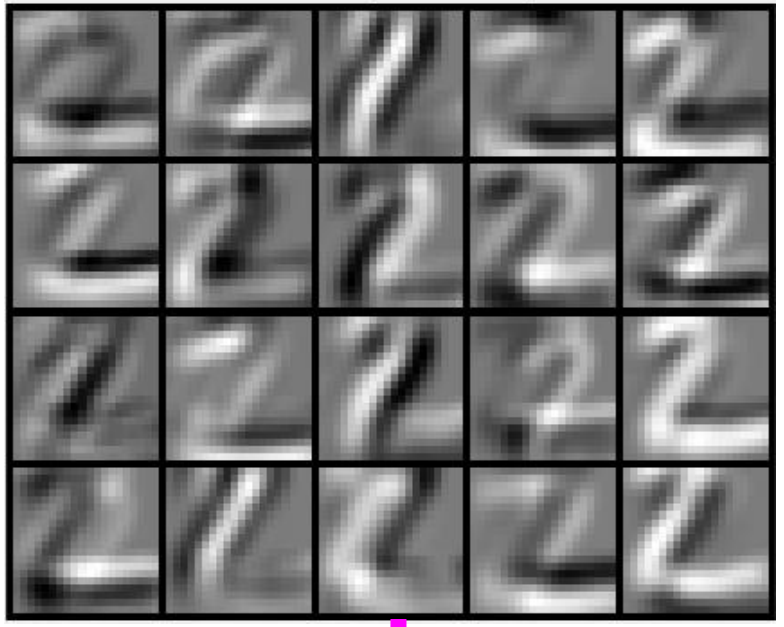
```
fList = zeros(20*20, 20);
for i = 1:20
  feature     = y1(:, :, i);
  fList(:, i) = feature(:);
end
figure
display_network(fList);
title('Features [Convolution]')

fList = zeros(20*20, 20);
for i = 1:20
  feature     = y2(:, :, i);
  fList(:, i) = feature(:);
end
figure
display_network(fList);
title('Features [Convolution + ReLU]')

fList = zeros(10*10, 20);
for i = 1:20
  feature     = y3(:, :, i);
  fList(:, i) = feature(:);
end
figure
display_network(fList);
title('Features [Convolution + ReLU + MeanPool]')
```

Input Image

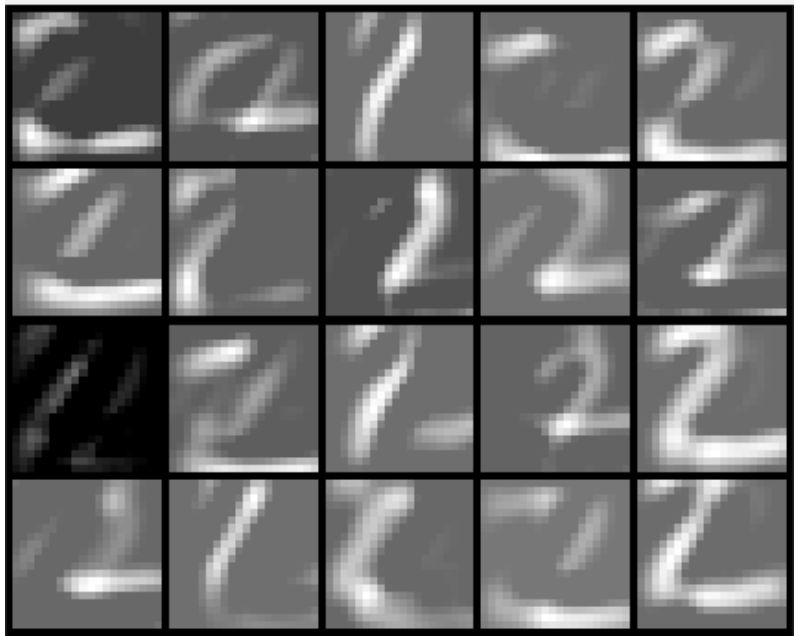
Convolution Filters



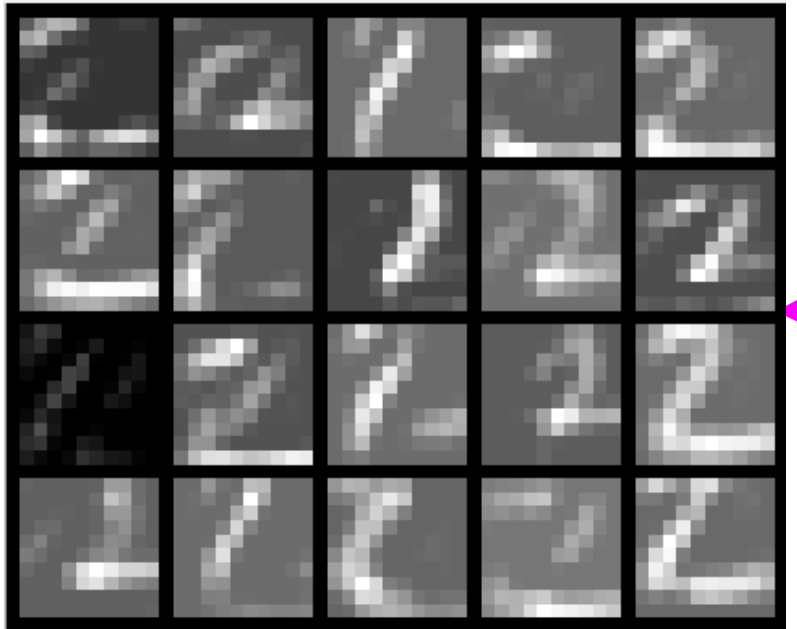
Features [Convolution]

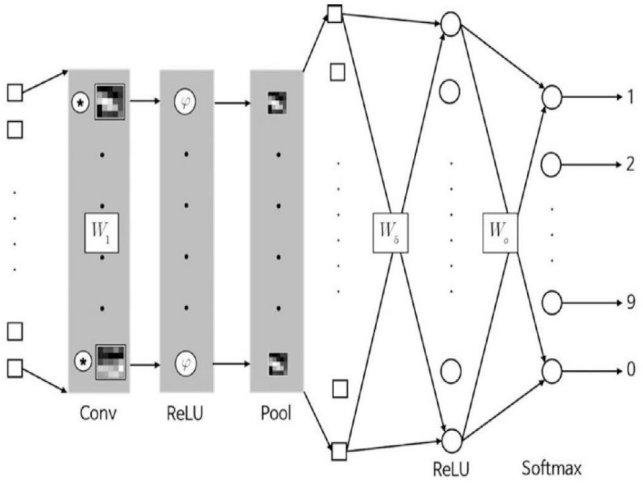


Features [Convolution + ReLU]



Features [Convolution + ReLU + MeanPool]

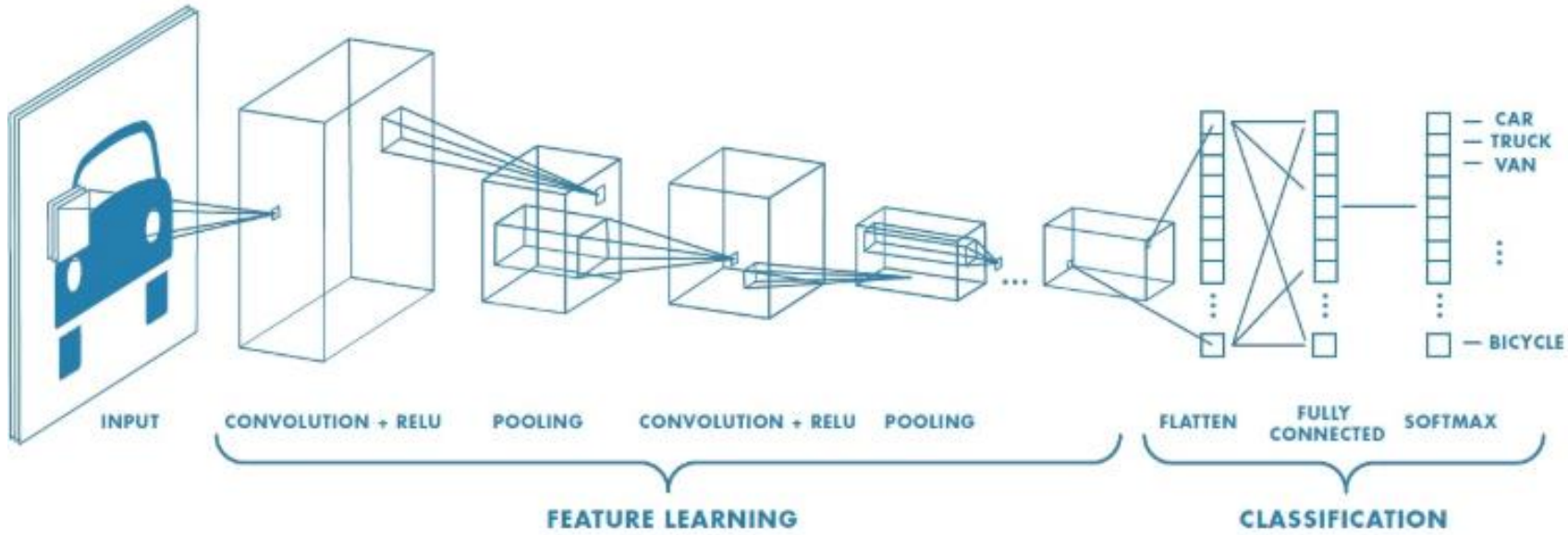


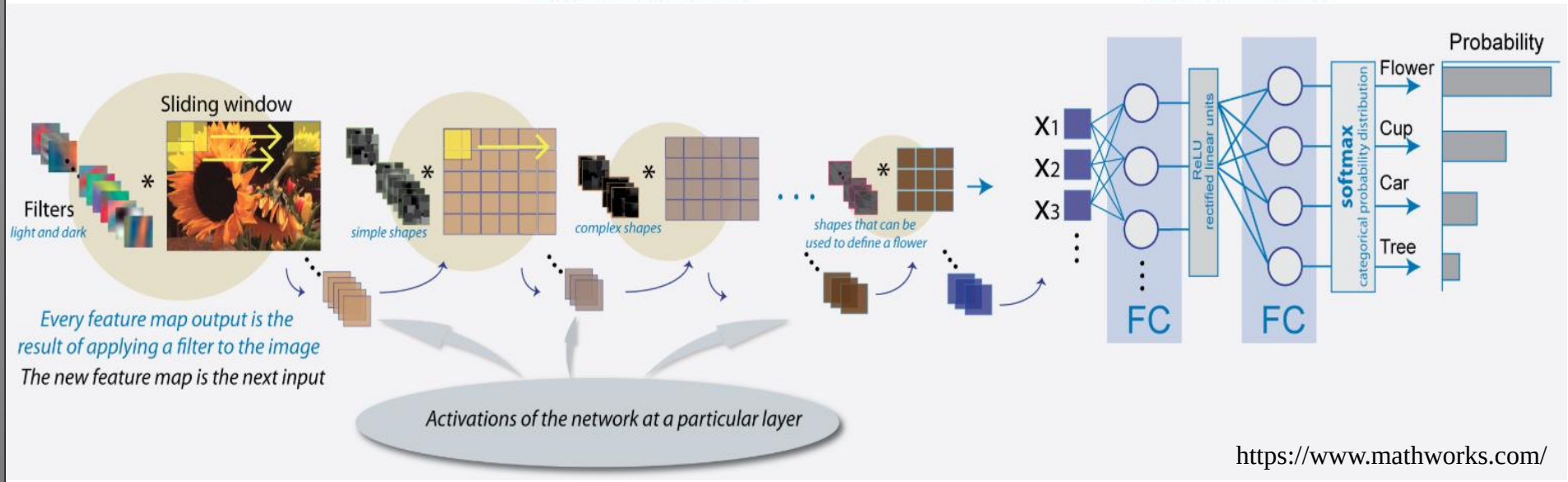


Convolutional Neural Network, CNN

Key Component of CNN

- Convolutional Layers
- Pooling Layers
- Fully-Connected Layers
- Output
- Loss
- Calculate output error with label
- Forward propagation
- Backpropagation





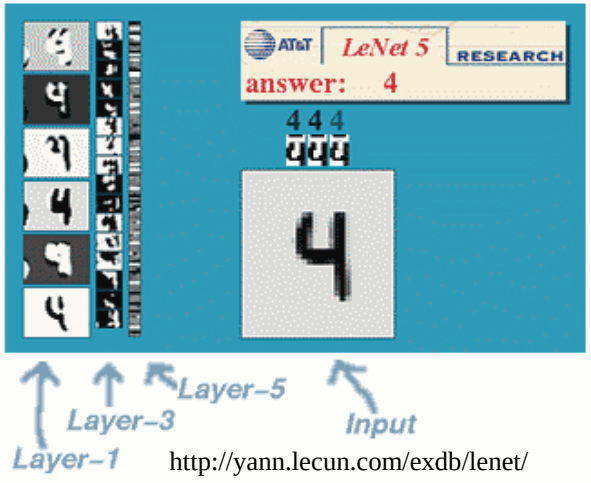
https://www.mathworks.com/

Introduction : CNN – LeNet, 1998

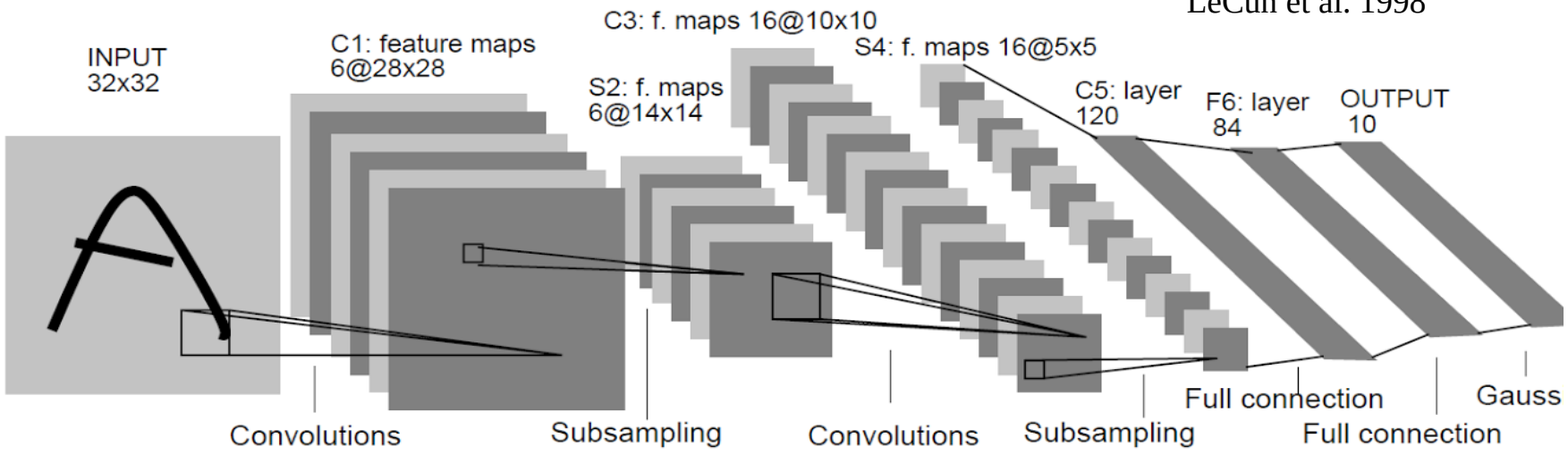
LeCun et al. 1998

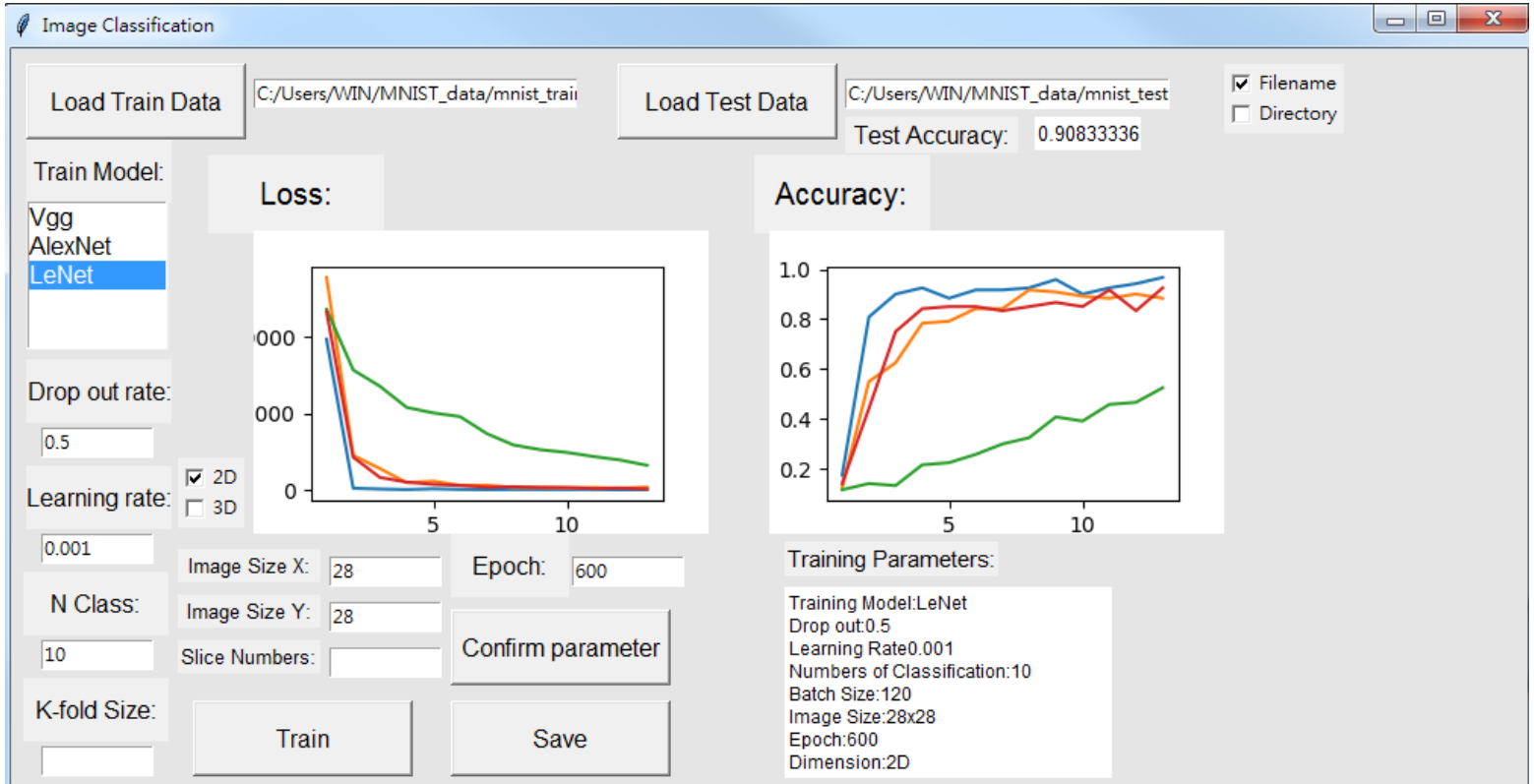
MNIST

Fig. 4. Size-normalized examples from the MNIST database.



http://yann.lecun.com/exdb/lenet/

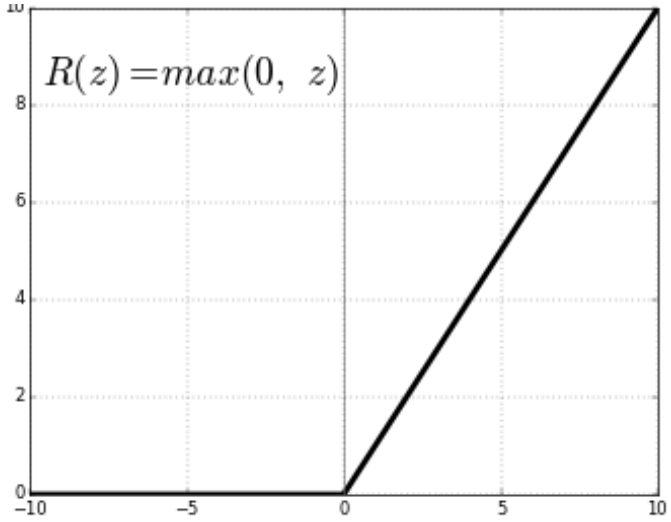




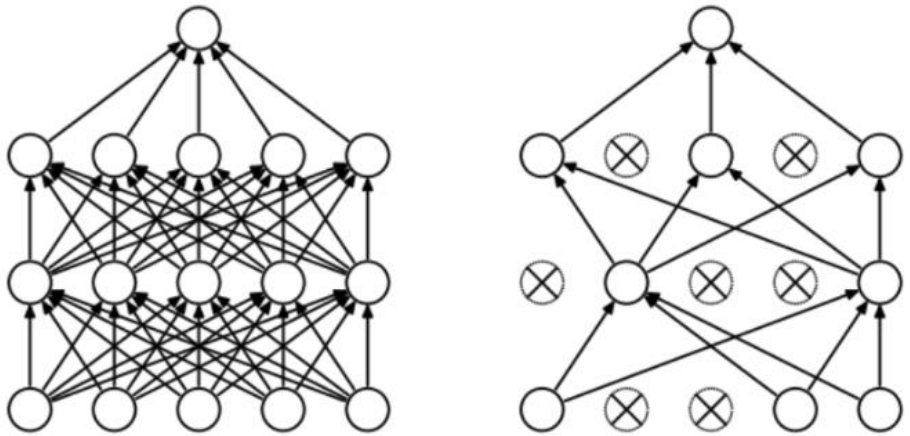
Introduction : CNN – AlexNet, 2012

- Main idea
- Activation function
  - **ReLU**
- Dropout
- Data Augmentation
  - Patch
  - RGB PCA

Rectified Linear Units (ReLU)



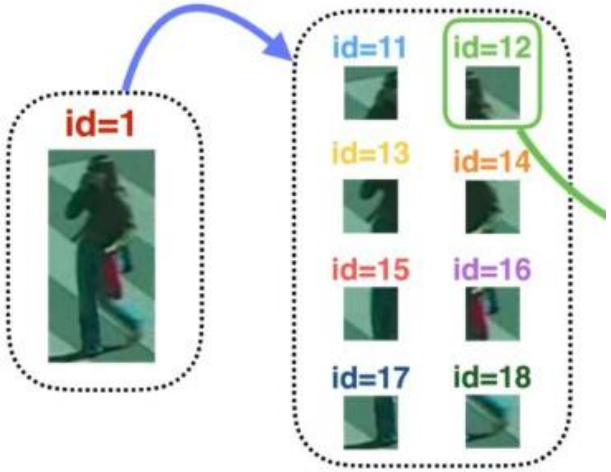
Dropout



(a) Standard Neural Net (b) After applying dropout.

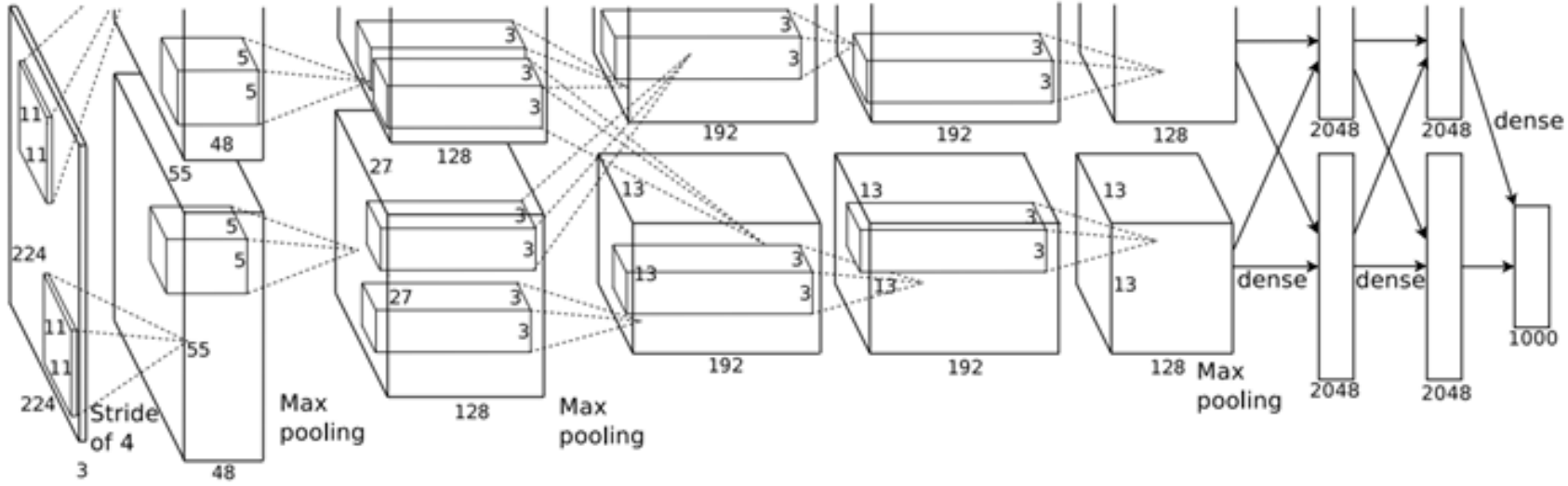
Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

Patch



https://www.disneyresearch.com/

AlexNet Architecture :

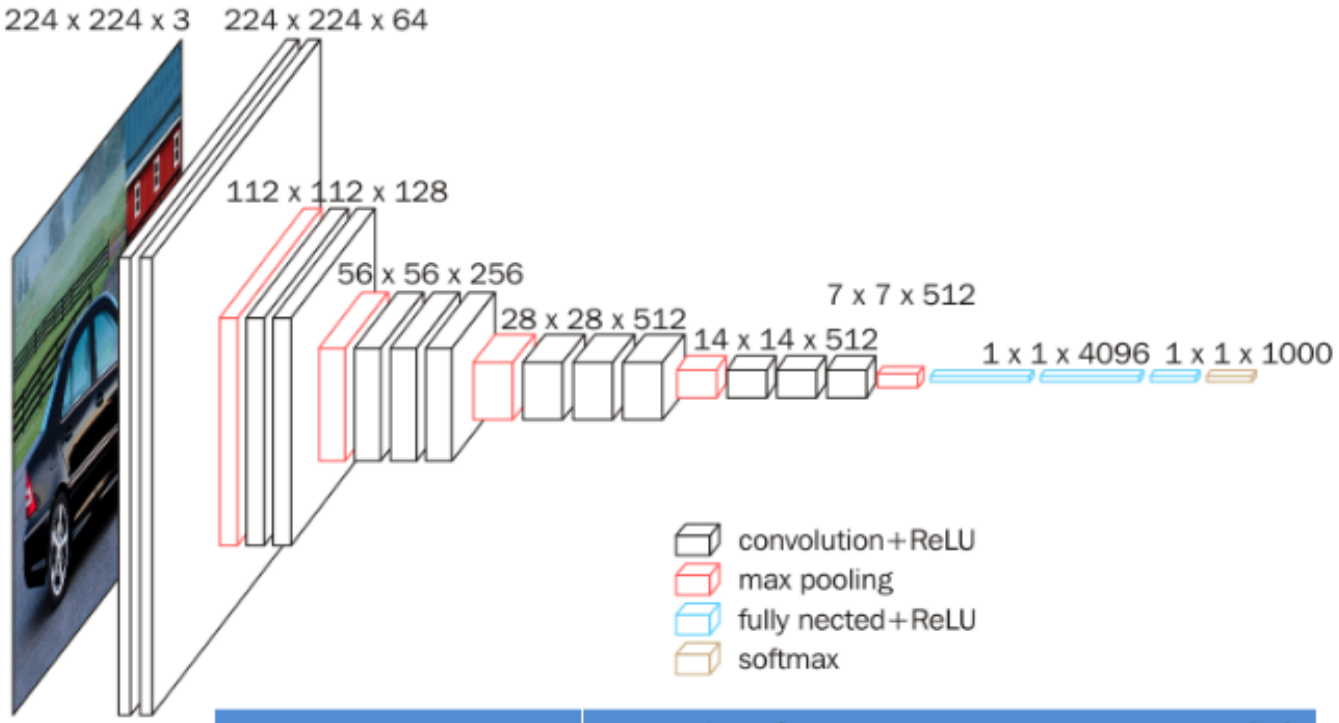


Introduction : CNN – VGG Net, 2014

VGG-16

Visual Geometry Group
Department of Engineering Science, University of Oxford

UNIVERSITY OF OXFORD



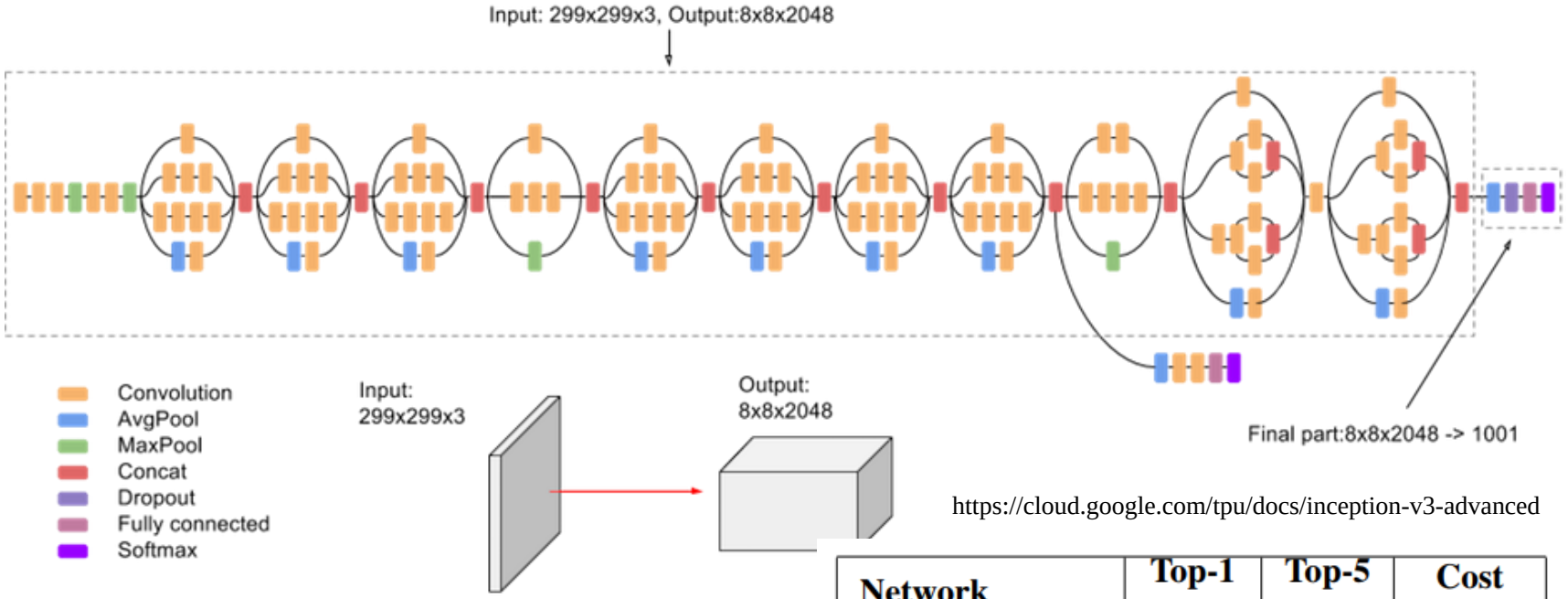
Model	top-5 classification error on ILSVRC-2012 (%)	
	validation set	test set
16-layer	7.5%	7.4%
19-layer	7.5%	7.3%
model fusion	7.1%	7.0%

- Small convolutional kernel is better (3x3)
- Deeper is better

http://www.robots.ox.ac.uk/~vgg/research/very_deep/
arXiv:1409.1556v6 [cs.CV] 10 Apr 2015

Deep Learning : InceptionV3 - GoogLeNet

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000



https://cloud.google.com/tpu/docs/inception-v3-advanced

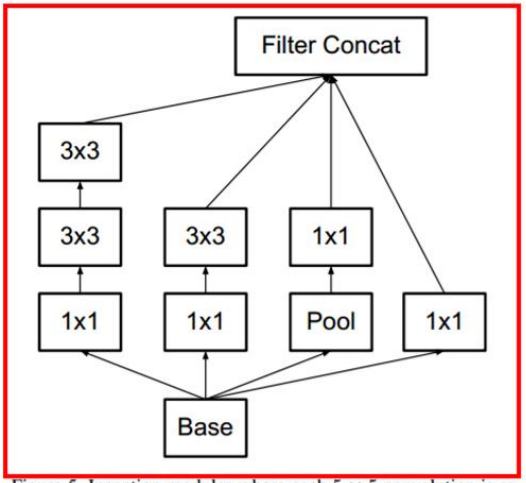


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2.

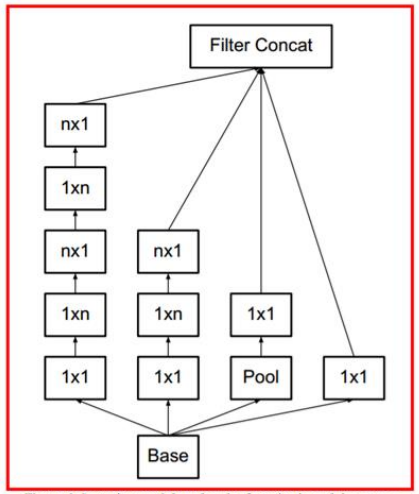


Figure 6. Inception modules after the factorization of the $n \times n$ convolutions. In our proposed architecture, we chose $n = 7$ for the 17×17 grid. (The filter sizes are picked using principle 3)

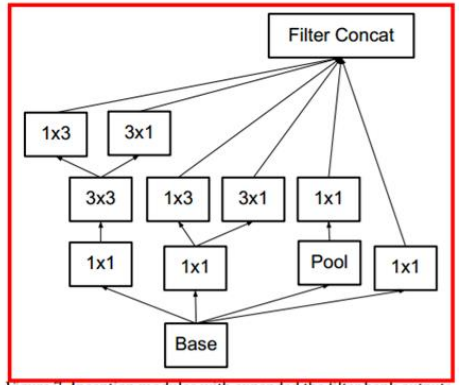
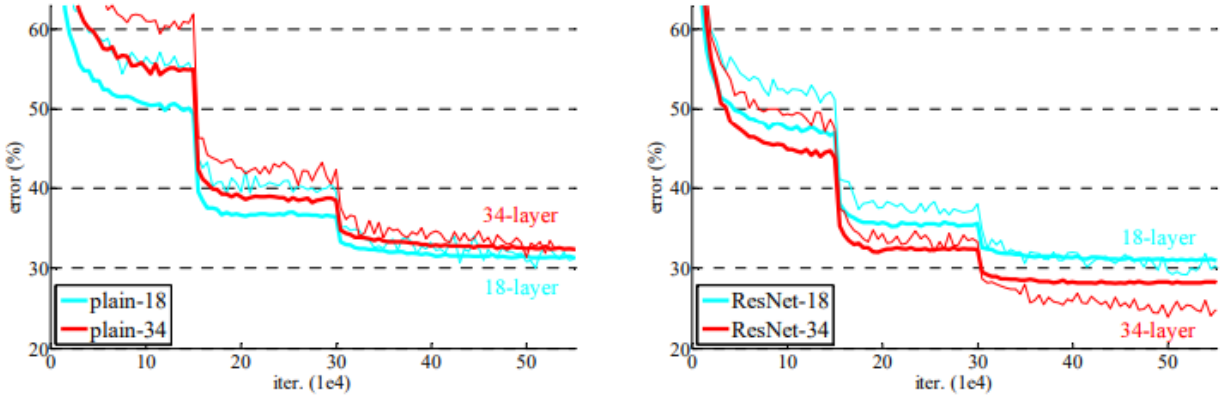


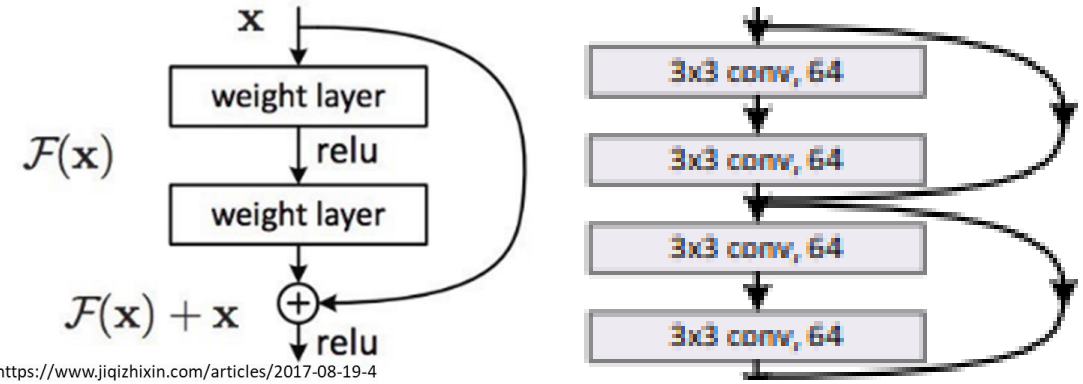
Figure 7. Inception modules with expanded the filter bank outputs. This architecture is used on the coarsest (8×8) grids to promote high dimensional representations, as suggested by principle 2 of Section 2. We are using this solution only on the coarsest grid, since that is the place where producing high dimensional sparse representation is the most critical as the ratio of local processing (by 1×1 convolutions) is increased compared to the spatial aggregation.

Network	Top-1 Error	Top-5 Error	Cost Bn Ops
GoogLeNet [20]	29%	9.2%	1.5
BN-GoogLeNet	26.8%	-	**1.5**
BN-Inception [7]	25.2%	7.8	2.0
Inception-v2	23.4%	-	3.8
Inception-v2 RMSProp	23.1%	6.3	3.8
Inception-v2 Label Smoothing	22.8%	6.1	3.8
Inception-v2 Factorized 7×7	21.6%	5.8	4.8
Inception-v2 BN-auxiliary	**21.2%**	**5.6%**	4.8

arXiv:1512.00567v1 [cs.CV] 2 Dec 2015

Introduction : CNN – ResNet, 2015





https://www.jiqizhixin.com/articles/2017-08-19-4

Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

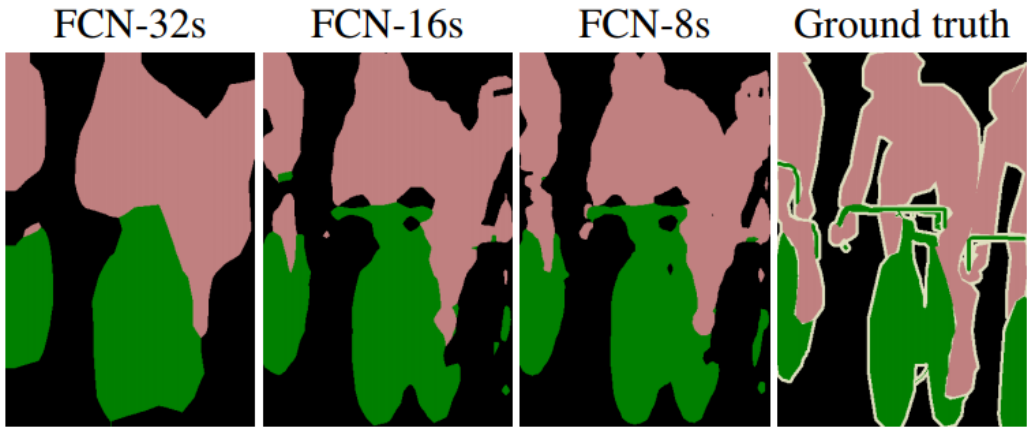
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

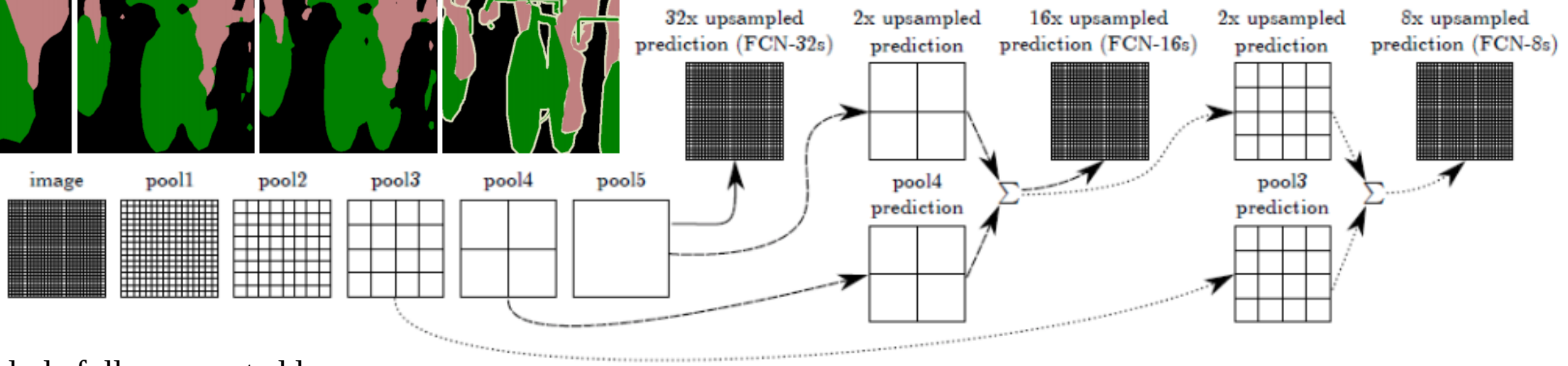
method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except † reported on the test set).

arXiv:1512.03385v1 [cs.CV] 10 Dec 2015

Deep Learning Model : FCN



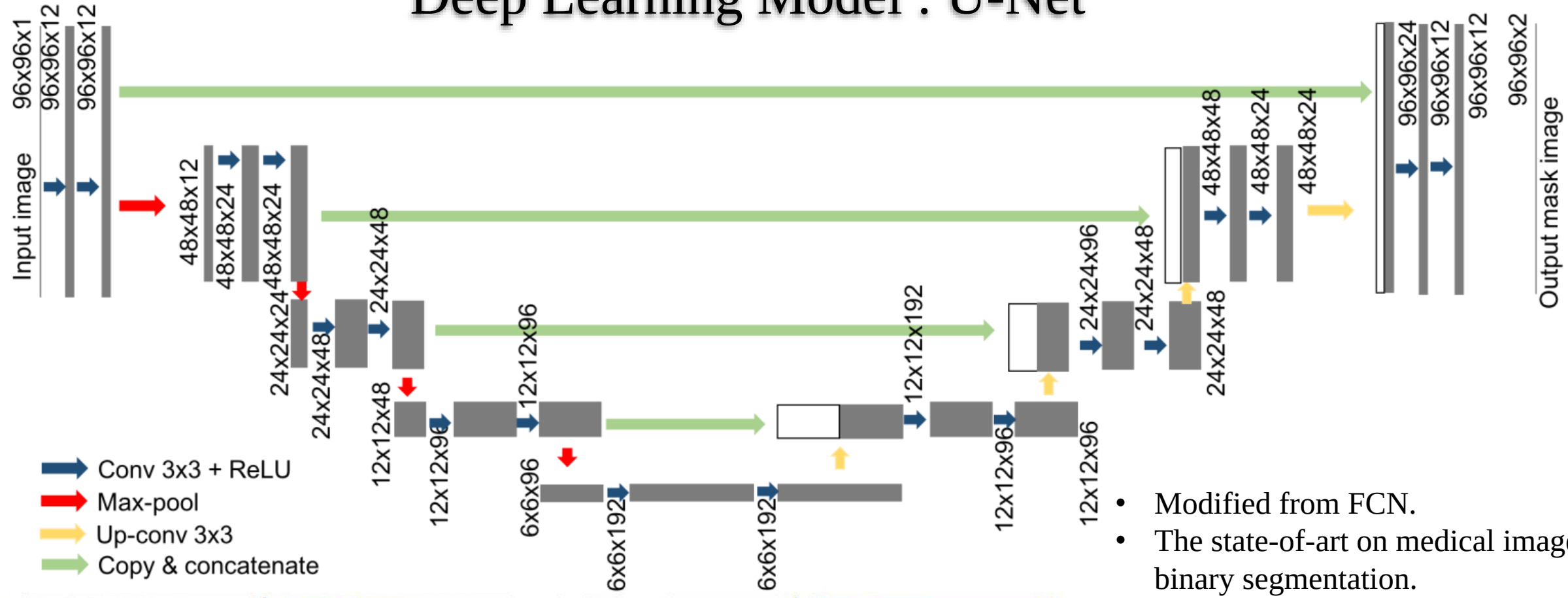


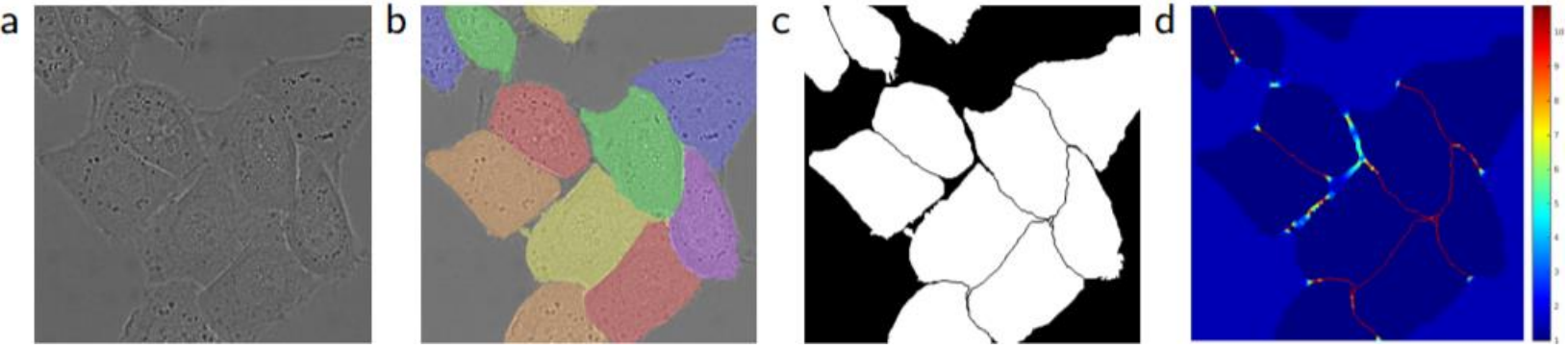
- Exclude fully connected layers (only used in classification)
- Convolution Transpose: for up-sampling
- Skip and Summation: fusion the predictions with the same size feature maps, retraining high-level semantic information.

Semantic Segmentation:
自動駕駛之道路、車輛、行人、號誌識別等

Jonathan Long ,Evan Shelhamer ,Trevor Darrell
"Fully Convolutional Networks for Semantic Segmentation."

Deep Learning Model : U-Net





- Modified from FCN.
- The state-of-art on medical images binary segmentation.
- The successive convolution block to obtain the more precise output.
- Concatenate the same size feature maps from encode part to decode part.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox "U-Net: Convolutional Networks for Biomedical Image Segmentation"

Image Analysis and Biometrics Lab @ IIIT Delhi

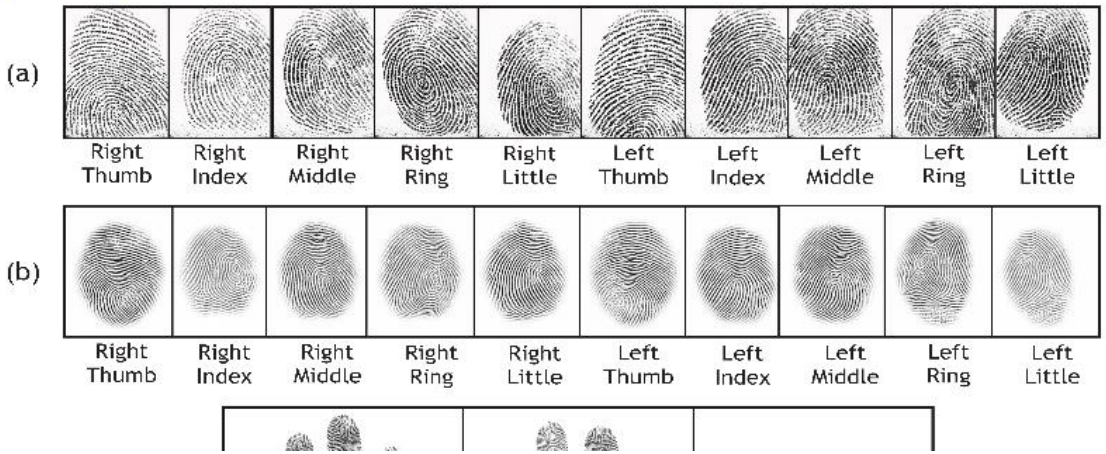
Home | People | Publications | Resources | Achievements | Updates

IIIT-D Multi-sensor Optical and Latent Fingerprint (MOLF)

IIIT-D MOLF database has 19,200 fingerprints collected using from 100 subjects using five different capture methods: (i) Lumidigm Venus IP65 Shell, (ii) Secugen Hamster-IV, (iii) CrossMatch L-Scan Patrol, (iv) Latent fingerprints, and (v) simultaneous latent fingerprints lifted using black powder dusting process. The details of the database are given below:

Subset	Fingerprint type	No. of Images	Image Size	Capture protocol	Comment
DB1	Multi-spectral live-scan dap	4000	352 × 544	100 users × 10 fingers × 2 sessions × 2 instances	Lumidigm Venus
DB2	Live-scan dap	4000	258 × 336	100 users × 10 fingers × 2 sessions × 2 instances	Secugen Hamster IV
DB3	Live-scan slap	1200	1600 × 1500	100 users × 3 slap prints × 2 sessions × 2 instances	CrossMatch L-Scan Patrol
DB3_A	Live-scan dap	4000	variable	100 users × 10 fingers × 2 sessions × 2 instances	Cropped prints from DB3
DB4	Latent	4400	variable	100 users × 2 hands × 2 sessions × 11 instances	Latent fingerprints, cropped from simultaneous prints
DB5	Simultaneous latent	1600	1924 × 1232	100 users × 2 hands × 2 sessions × 4 instances	Simultaneous impression with annotated ROI, core points and minutiae

Sample Images



http://www.iab-rubric.org/resources/molf.html