CHAPTER 6

Convolutional Neural Network

Chapter 5 showed that incomplete training is the cause of the poor performance of the deep neural network and introduced how Deep Learning solved the problem. The importance of the deep neural network lies in the fact that it opened the door to the complicated non-linear model and systematic approach for the hierarchical processing of knowledge.

This chapter introduces the convolutional neural network (ConvNet), which is a deep neural network specialized for image recognition. This technique exemplifies how significant the improvement of the deep layers is for information (images) processing. Actually, ConvNet is an old technique, which was developed in the 1980s and 1990s.[1] However, it has been forgotten for a while, as it was impractical for real-world applications with complicated images. Since 2012 when it was dramatically revived[2], ConvNet has conquered most computer vision fields and is growing at a rapid pace.

Architecture of ConvNet

ConvNet is not just a deep neural network that has many hidden layers. It is a deep network that imitates how the visual cortex of the brain processes and recognizes images. Therefore, even the experts of neural networks often have a hard time understanding this concept on their first encounter. That is how much ConvNet differs in concept and operation from the previous neural networks. This section briefly introduces the fundamental architecture of ConvNet.

[1]LeCun, Y., et al., "Handwritten digit recognition with a back-propagation network," In *Proc. Advances in Neural Information Processing Systems*, 396–404 (1990).
[2]Krizhevsky, Alex, "ImageNet Classification with Deep Convolutional Neural Networks," 17 November 2013.

P. Kim, *MATLAB Deep Learning*, DOI 10.1007/978-1-4842-2845-6_6

Basically, image recognition is the classification. For example, recognizing whether the image of a picture is a cat or a dog is the same as classifying the image into a cat or dog class. The same thing applies to the letter recognition; recognizing the letter from an image is the same as classifying the image into one of the letter classes. Therefore, the output layer of the ConvNet generally employs the multiclass classification neural network.

However, directly using the original images for image recognition leads to poor results, regardless of the recognition method; the images should be processed to contrast the features. The examples in Chapter 4 used the original images and they worked well because they were simple black-and-white images. Otherwise, the recognition process would have ended up with very poor results. For this reason, various techniques for image feature extraction have been developed.[3]

Before ConvNet, the feature extractor has been designed by experts of specific areas. Therefore, it required a significant amount of cost and time while it yielded an inconsistent level of performance. These feature extractors were independent of Machine Learning. Figure 6-1 illustrates this process.

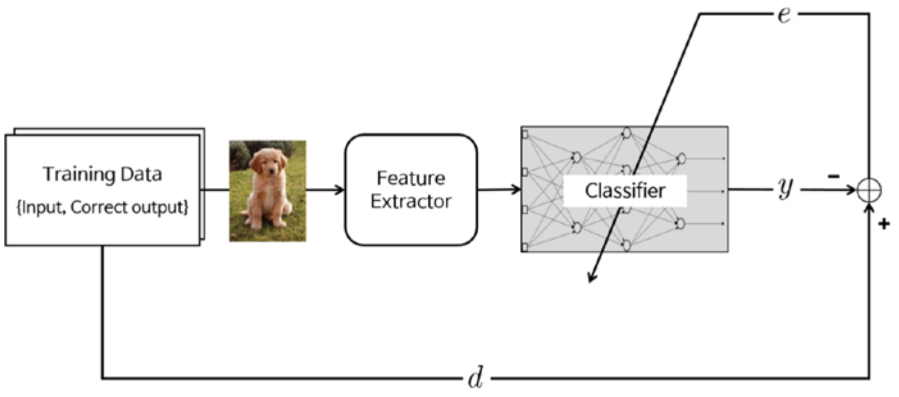


Figure 6-1. *Feature extractors used to be independent of Machine Learning*

ConvNet includes the feature extractor in the training process rather than designing it manually. The feature extractor of ConvNet is composed of special kinds of neural networks, of which the weights are determined via the training process. The fact that ConvNet turned the manual feature extraction design into the automated process is its primary feature and advantage. Figure 6-2 depicts the training concept of ConvNet.

[3]The representative methods include SIFT, HoG, Textons, Spin image, RIFT, and GLOH.

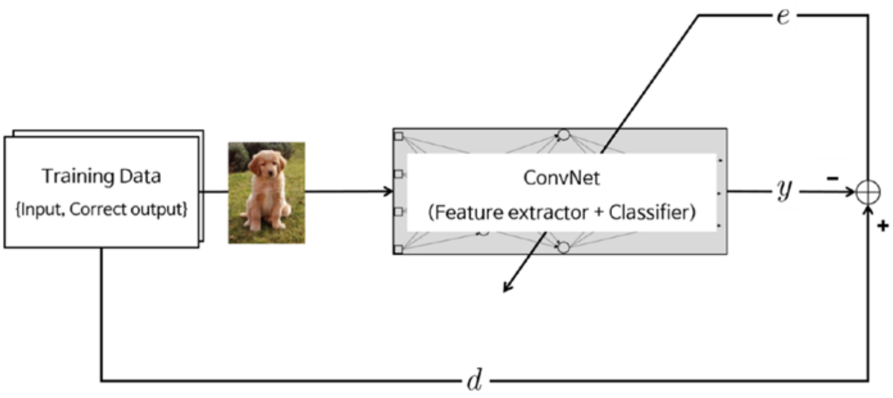


Figure 6-2. *ConvNet's feature extractor is composed of special kinds of neural networks*

ConvNet yields better image recognition when its feature extraction neural network is deeper (contains more layers), at the cost of difficulties in the training process, which had driven ConvNet to be impractical and forgotten for a while.

Let's go a bit deeper. ConvNet consists of a neural network that extracts features of the input image and another neural network that classifies the feature image. Figure 6-3 shows the typical architecture of ConvNet.

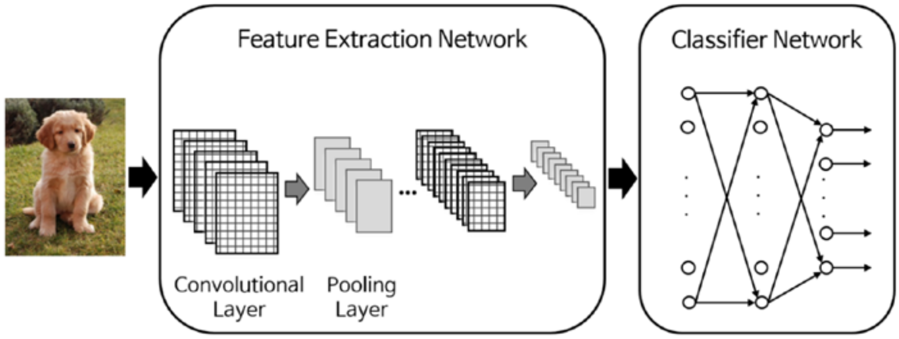


Figure 6-3. *Typical architecture of ConvNet*

The input image enters into the feature extraction network. The extracted feature signals enter the classification neural network. The classification neural network then operates based on the features of the image and generates the output. The classification techniques discussed in Chapter 4 apply here.

The feature extraction neural network consists of piles of the convolutional layer and pooling layer pairs. The convolution layer, as its name implies, converts the image using the convolution operation. It can be thought of as a collection of digital filters. The pooling layer combines the neighboring pixels into a single pixel. Therefore, the pooling layer reduces the dimension of the image. As the primary concern of ConvNet is the image; the operations of the convolution and pooling layers are conceptually in a two-dimensional plane. This is one of the differences between ConvNet and other neural networks.

In summary, ConvNet consists of the serial connection of the feature extraction network and the classification network. Through the training process, the weights of both layers are determined. The feature extraction layer has piled pairs of the convolution and pooling layers. The convolution layer converts the images via the convolution operation, and the pooling layer reduces the dimension of the image. The classification network usually employs the ordinary multiclass classification neural network.

Convolution Layer

This section explains how the convolution layer, which is one side of the feature extraction neural network, works. The pooling layer, the other side of the pair, is introduced in the next section.

The convolution layer generates new images called *feature maps*. The feature map accentuates the unique features of the original image. The convolution layer operates in a very different way compared to the other neural network layers. This layer does not employ connection weights and a weighted sum.[4] Instead, it contains filters[5] that convert images. We will call these filters *convolution filters*. The process of the inputting the image through the convolution filters yields the feature map.

Figure 6-4 shows the process of the convolution layer, where the circled * mark denotes the convolution operation, and the φ mark is the activation function. The square grayscale icons between these operators indicate the convolution filters. The convolution layer generates the same number of feature maps as the convolution filters. Therefore, for instance, if the convolution layer contains four filters, it will generate four feature maps.

[4]It is often explained using the local receptive filed and shared weights from the perspective of the ordinary neural network. However, they would not be helpful for beginners. This book does not insist its relationship with the ordinary neural network and explains it as a type of digital filter.

[5]Also called kernels.

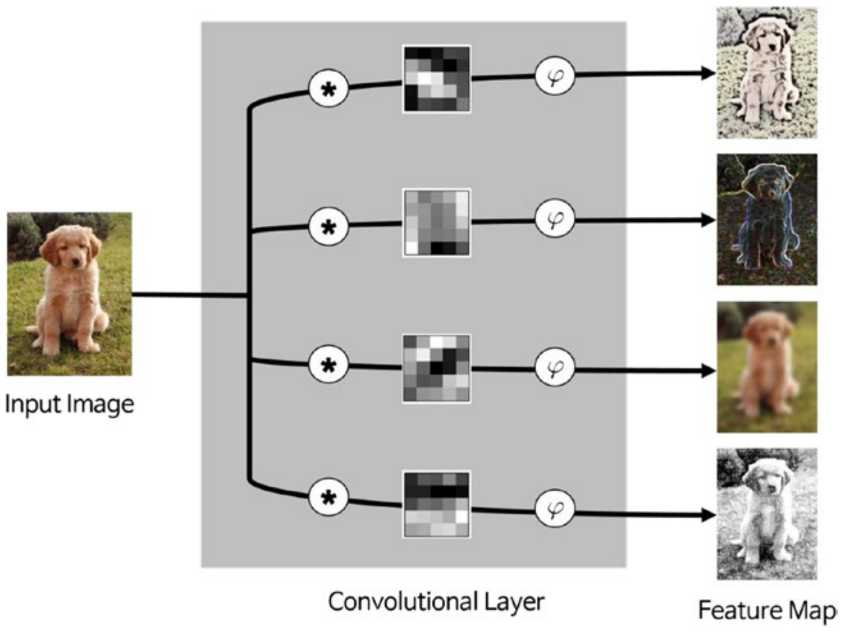
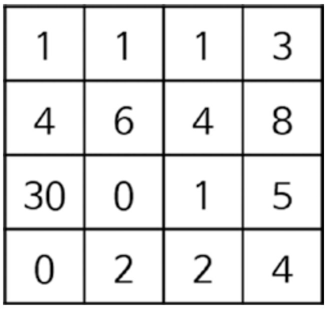


Figure 6-4. *The convolution layer process*

Let's further explore the details of the convolution filter. The filters of the convolution layer are two-dimensional matrices. They usually come in 5×5 or 3×3 matrices, and even 1×1 convolution filters have been used in recent applications. Figure 6-4 shows the values of the 5×5 filters in grayscale pixels. As addressed in the previous section, the values of the filter matrix are determined through the training process. Therefore, these values are continuously trained throughout the training process. This aspect is similar to the updating process of the connection weights of the ordinary neural network.

The *convolution* is a rather difficult operation to explain in text as it lies on the two-dimensional plane. However, its concept and calculation steps are simpler than they appear.[6] A simple example will help you understand how it works. Consider a 4×4 pixel image that is expressed as the matrix shown in Figure 6-5. We will generate a feature map via the convolution filter operation of this image.

[6]`deeplearning.stanford.edu/wiki/images/6/6c/Convolution_schematic.gif`



1	1	1	3
4	6	4	8
30	0	1	5
0	2	2	4

Figure 6-5. *Four-by-four pixel image*

We use the two convolution filters shown here. It should be noted that the filters of the actual ConvNet are determined through the training process and not by manual decision.

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

Let's start with the first filter. The convolution operation begins at the upper-left corner of the submatrix that is the same size as the convolution filter (see Figure 6-6).

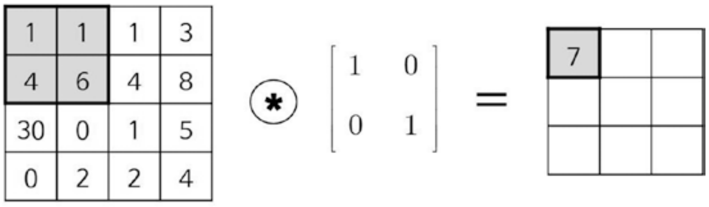


Figure 6-6. *The convolution operation starts at the upper-left corner*

The convolution operation is the sum of the products of the elements that are located on the same positions of the two matrices. The result of 7 in Figure 6-6 is calculated as:

$$(1\times1) + (1\times0) + (4\times0) + (6\times1) \quad = \quad 7$$

Another convolution operation is conducted for the next submatrix (see Figure 6-7).[7]

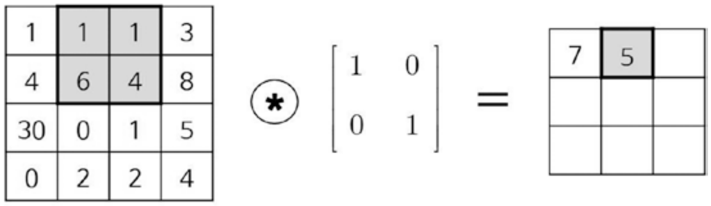


Figure 6-7. *The second convolution operation*

In the same manner, the third convolution operation is conducted, as shown in Figure 6-8.

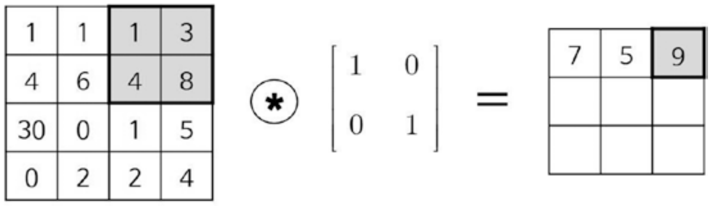


Figure 6-8. *The third convolution operation*

Once the top row is finished, the next row starts over from the left (see Figure 6-9).

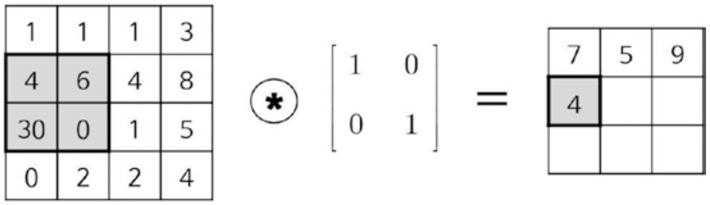


Figure 6-9. *The convolution operation starts over from the left*

[7]The designer decides how many elements to stride for each operation. It can be greater than one if the filter is larger.

It repeats the same process until the feature map of the given filter is produced, as shown in Figure 6-10.

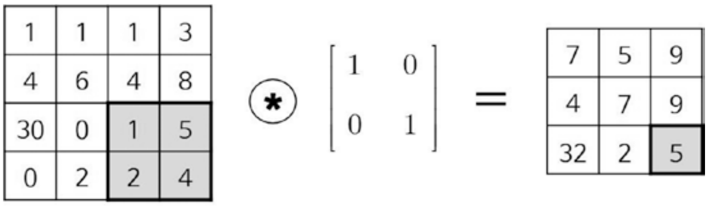


Figure 6-10. *The feature map of the given filter has been completed*

Now, take a closer look at the feature map. The element of (3, 1) of the map shows the greatest value. What happened to this cell? This value is the result of the convolution operation shown in Figure 6-11.

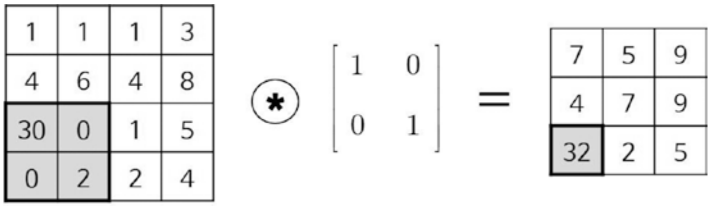


Figure 6-11. *The submatrix of the image matches the convolution filter*

It is noticeable from the figure that the submatrix of the image matches the convolution filter; both are diagonal matrices with significant numbers on the same cells. The convolution operation yields large values when the input matches the filter, as shown in Figure 6-12.

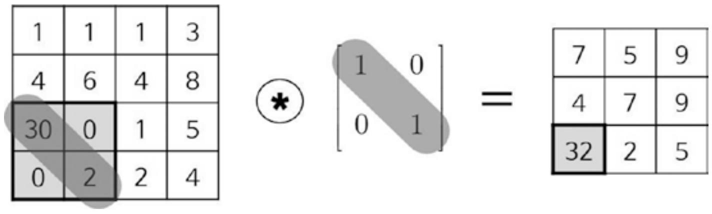


Figure 6-12. *The convolution operation yields large values when the input matches the filter*

In contrast, in the case shown in Figure 6-13, the same significant number of 30 does not affect the convolution result, which is only 4. This is because the image matrix does not match the filter; the significant elements of the image matrix are aligned in the wrong direction.

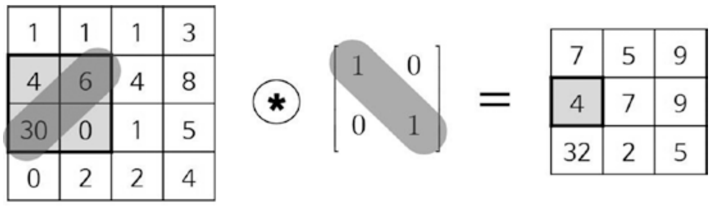


Figure 6-13. *When the image matrix does not match the filter, the significant elements are not aligned*

In the same manner, processing the second convolution filter produces the feature map shown in Figure 6-14.

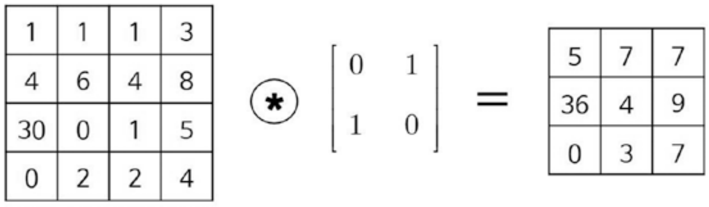


Figure 6-14. *The values depend on whether the image matrix matches the convolution filter*

Similarly to the first convolution operation, the values in the elements of this feature map depend on whether the image matrix matches the convolution filter or not.

In summary, the convolution layer operates the convolution filters on the input image and produces the feature maps. The features that are extracted in the convolution layer determined by the trained convolution filters. Therefore, the features that the convolution layer extracts vary depending on which convolution filter is used.

The feature map that the convolution filter creates is processed through the activation function before the layer yields the output. The activation function of the convolution layer is identical to that of the ordinary neural network. Although the ReLU function is used in most of the recent applications, the sigmoid function and the tanh function are often employed as well.[8]

Just for the reference, the moving average filter, which is widely used in the digital signal processing field, is a special type of convolution filter. If you are familiar with digital filters, relating them to this concept may allow you to better understand the ideas behind the convolution filter.

Pooling Layer

The *pooling layer* reduces the size of the image, as it combines neighboring pixels of a certain area of the image into a single representative value. Pooling is a typical technique that many other image processing schemes have already been employing.

In order to conduct the operations in the pooling layer, we should determine how to select the pooling pixels from the image and how to set the representative value. The neighboring pixels are usually selected from the square matrix, and the number of pixels that are combined differs from problem to problem. The representative value is usually set as the mean or maximum of the selected pixels.

The operation of the pooling layer is surprisingly simple. As it is a two-dimensional operation, and an explanation in text may lead to more confusion, let's go through an example. Consider the 4×4 pixel input image, which is expressed by the matrix shown in Figure 6-15.

1	1	1	3
4	6	4	8
30	0	1	5
0	2	2	4

Figure 6-15. *The four-by-four pixel input image*

We combine the pixels of the input image into a 2×2 matrix without overlapping the elements. Once the input image passes through the pooling layer, it shrinks into a 2×2 pixel image. Figure 6-16 shows the resultant cases of pooling using the mean pooling and max pooling.

[8]Sometimes the activation function is omitted depending on the problem.

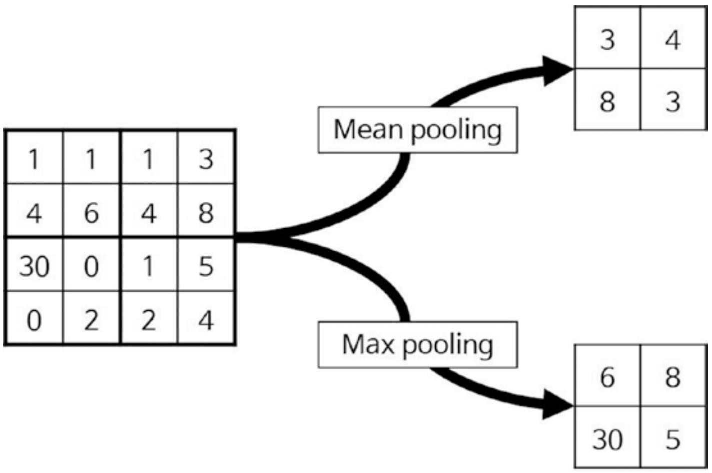


Figure 6-16. *The resultant cases of pooling using two different methods*

Actually, in a mathematical sense, 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 example provided in the next section will elaborate on this.

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. In addition, as the pooling process reduces the image size, it is highly beneficial for relieving the computational load and preventing overfitting.

Example: MNIST

We implement a neural network that takes the input image and recognizes the digit that it represents. The training data is the MNIST[9] database, which contains 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. Each digit image is a 28-by-28 pixel black-and-white image, as shown in Figure 6-17.

[9]Mixed National Institute of Standards and Technology.

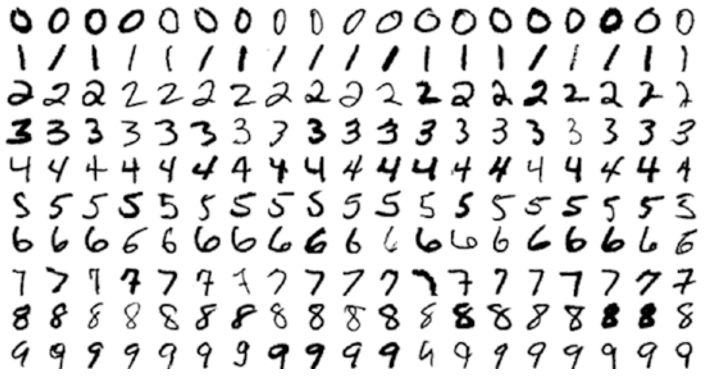


Figure 6-17. *A 28-by-28 pixel black-and-white image from the MNIST database*

Considering the training time, this example employs only 10,000 images with the training data and verification data in an 8:2 ratio. Therefore, we have 8,000 MNIST images for training and 2,000 images for validation of the performance of the neural network. As you may know well by now, the MNIST problem is caused by the multiclass classification of the 28×28 pixel image into one of the ten digit classes of 0-9.

Let's consider a ConvNet that recognizes the MNIST images. As the input is a 28×28 pixel black-and-white image, we allow 784(=28x28) input nodes. The feature extraction network contains a single convolution layer with 20 9×9 convolution filters. The output from the convolution layer passes through the `ReLU` function, followed by the pooling layer. The pooling layer employs the mean pooling process of two by two submatrices. The classification neural network consists of the single hidden layer and output layer. This hidden layer has 100 nodes that use the ReLU activation function. Since we have 10 classes to classify, the output layer is constructed with 10 nodes. We use the softmax activation function for the output nodes. The following table summarizes the example neural network.

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`

Figure 6-18 shows the architecture of this neural network. Although it has many layers, only three of them contain the weight matrices that require training; they are W_1, W_5, and W_o in the square blocks. W_5 and W_o contain the connection weights of the classification neural network, while W_1 is the convolution layer's weight, which is used by the convolution filters for image processing.

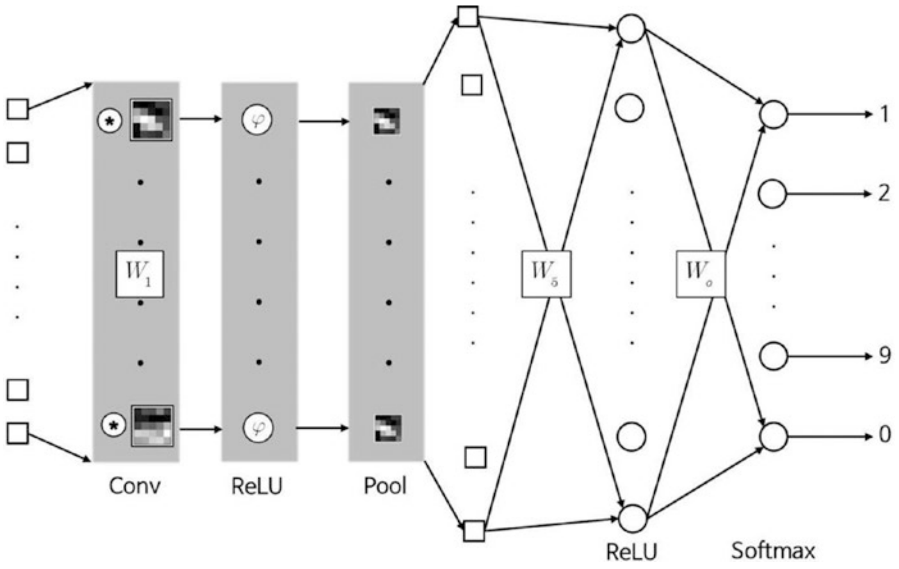


Figure 6-18. *The architecture of this neural network*

The input nodes between the pooling layer and the hidden layer, which are the square nodes left of the W_5 block, are the transformations of the two-dimensional image into a vector. As this layer does not involve any operations, these nodes are denoted as squares.

The function `MnistConv`, which trains the network using the back-propagation algorithm, takes the neural network's weights and training data and returns the trained weights.

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

where `W1`, `W5`, and `Wo` are the convolution filter matrix, pooling-hidden layer weight matrix, and hidden-output layer weight matrix, respectively. `X` and `D` are the input and correct output from the training data, respectively. The following listing shows the `MnistConv.m` file, which implements the `MnistConv` function.

```
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,           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);                 %

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

    % Backpropagation
    %
    e      = d - y;                   % Output layer
    delta  = e;
    e5     = Wo' * delta;            % Hidden(ReLU) layer
    delta5 = (y5 > 0) .* e5;

    e4     = W5' * delta5;           % Pooling layer

    e3     = reshape(e4, size(y3));

    e2 = zeros(size(y2));
    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;         % ReLU layer

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

    dW1 = dW1 + delta1_x;
    dW5 = dW5 + delta5*y4';
    dWo = dWo + delta *y5';
  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
```

This code appears to be rather more complex than the previous examples. Let's take a look at it part by part. The function `MnistConv` trains the network via the `minibatch` method, while the previous examples employed the SGD and `batch` methods. The `minibatch` portion of the code is extracted and shown in the following listing.

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

for batch = 1:length(blist)
  ...
  begin = blist(batch);
  for k = begin:begin+bsize-1
    ...
    dW1 = dW1 + delta2_x;
    dW5 = dW5 + delta5*y4';
    dWo = dWo + delta *y5';
  end
  dW1 = dW1 / bsize;
  dW5 = dW5 / bsize;
  dWo = dWo / bsize;
  ...
end
```

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. In this example, the variable `blist` stores the following values:

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

Once the starting point, `begin`, of the minibatch is found via `blist`, the weight update is calculated for every 100th data point. The 100 weight updates are summed and averaged, and the weights are adjusted. Repeating this process 80 times completes one epoch.

Another noticeable aspect of the function `MnistConv` is that it adjusts the weights using momentum. The variables `momentum1`, `momentum5`, and `momentumo` are used here. The following part of the code implements the momentum update:

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

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

momentumo = alpha*dWo + beta*momentumo;
Wo        = Wo + momentumo;
...
```

We have now captured the big picture of the code. Now, let's look at the learning rule, the most important part of the code. The process itself is not distinct from the previous ones, as ConvNet also employs back-propagation training. The first thing that must be obtained is the output of the network. The following listing shows the output calculation portion of the function `MnistConv`. It can be intuitively understood from the architecture of the neural network. The variable `y` of this code is the final output of the network.

```
...
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);                    %
...
```

Now that we have the output, the error can be calculated. As the network has 10 output nodes, the correct output should be in a 10×1 vector in order to calculate the error. However, the MNIST data gives the correct output as the respective digit. For example, if the input image indicates a 4, the correct output will be given as a 4. The following listing converts the numerical correct output into a 10×1 vector. Further explanation is omitted.

```
d = zeros(10, 1);
d(sub2ind(size(d), D(k), 1)) = 1;
```

The last part of the process is the back-propagation of the error. The following listing shows the back-propagation from the output layer to the subsequent layer to the pooling layer. As this example employs cross entropy and softmax functions, the output node delta is the same as that of the network output error. The next hidden layer employs the `ReLU` activation function. There is nothing particular there. The connecting layer between the hidden and pooling layers is just a rearrangement of the signal.

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

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

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

We have two more layers to go: the pooling and convolution layers. The following listing shows the back-propagation that passes through the pooling layer-ReLU-convolution layer. The explanation of this part is beyond the scope of this book. Just refer to the code when you need it in the future.

```
...
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
...
```

The following listing shows the function Conv, which the function MnistConv calls. This function takes the input image and the convolution filter matrix and returns the feature maps. This code is in the Conv.m file.

```
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
```

This code performs the convolution operation using conv2, a built-in two-dimensional convolution function of MATLAB. Further details of the function Conv are omitted, as it is beyond the scope of this book.

The function MnistConv also calls the function Pool, which is implemented in the following listing . This function takes the feature map and returns the image after the 2×2 mean pooling process. This function is in the Pool.m file.

```
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
```

There is something interesting about this code; it 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. The further details of the code are beyond the scope of this book.

The following listing shows the `TestMnistConv.m` file, which tests the function `MnistConv`.[10] This program calls the function `MnistConv` and trains the network three times. It provides the 2,000 test data points to the trained network and displays its accuracy. The test run of this example yielded an accuracy of 93% in 2 minutes and 30 seconds. Be advised that this program takes quite some time to run.

```
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);
```

[10] `loadMNISTImages` and `loadMNISTLabels` functions are from `github.com/amaas/stanford_dl_ex/tree/master/common`.

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

save('MnistConv.mat');

% 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);
```

This program is also very similar to the previous ones. The explanations regarding the similar parts will be omitted. The following listing shown is a new entry. It compares the network's output and the correct output and counts the matching cases. It converts the 10×1 vector output back into a digit so that it can be compared to the given correct output.

```
...
[~, i] = max(y)
if i == D(k)
```

```
  acc = acc + 1;
end
...
```

Lastly, let's investigate how the image is processed while it passes through the convolution layer and pooling layer. The original dimension of the MNIST image is 28×28. Once the image is processed with the 9×9 convolution filter, it becomes a 20×20 feature map.[11] As we have 20 convolution filters, the layer produces 20 feature maps. Through the 2×2 mean pooling process, the pooling layer shrinks each feature map to a 10×10 map. The process is illustrated in Figure 6-19.

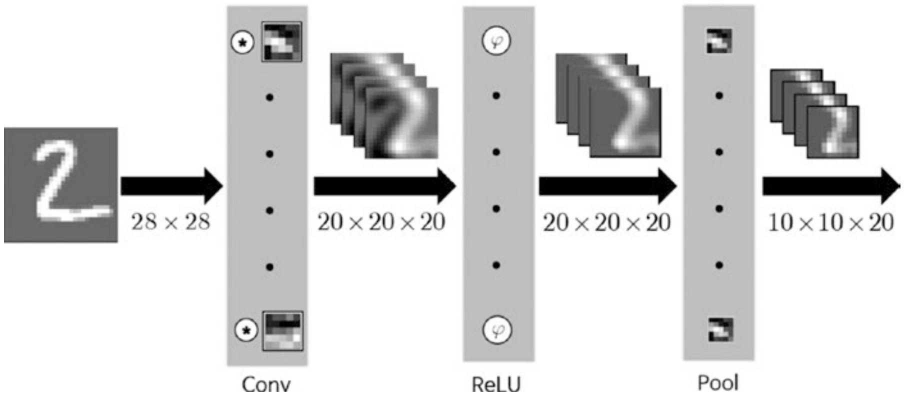


Figure 6-19. *How the image is processed while it passes through the convolution and pooling layers*

The final result after passing the convolution and pooling layers is as many smaller images as the number of the convolution filters; ConvNet converts the input image into the many small feature maps.

Now, we will see how the image actually evolves at each layer of ConvNet. By executing the TestMnistConv.m file, followed by the `PlotFeatures.m` file, the screen will display the five images. The following listing is in the `PlotFeatures.m` file.

[11]This size is valid only for this particular example. It varies depending on how the convolution filter is applied.

```
clear all

load('MnistConv.mat')

k  = 2;
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);                 %

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]')
```

The code enters the second image (**k = 2**) of the test data into the neural network and displays the results of all the steps. The display of the matrix on the screen is performed by the function `display_network`, which is originally from the same resource where the `loadMNISTImages` and `loadMNISTLabels` of the `TestMnistConv.m` file are from.

The first image that the screen shows is the following 28×28 input image of a 2, as shown in Figure 6-20.

Figure 6-20. *The first image shown*

Figure 6-21 is the second image of the screen, which consists of the 20 trained convolution filters. Each filter is pixel image and shows the element values as grayscale shades. The greater the value is, the brighter the shade becomes. These filters are what ConvNet determined to be the best features that could be extracted from the MNIST image. What do you think? Do you see some unique features of the digits?

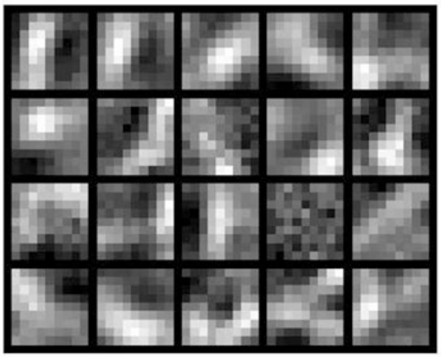


***Figure 6-21.** Image showing 20 trained convolution filters*

Figure 6-22 is the third image from the screen, which provides the results (**y1**) of the image processing of the convolution layer. This feature map consists of 20 20×20 pixel images. The various alterations of the input image due to the convolution filter can be noticeable from this figure.

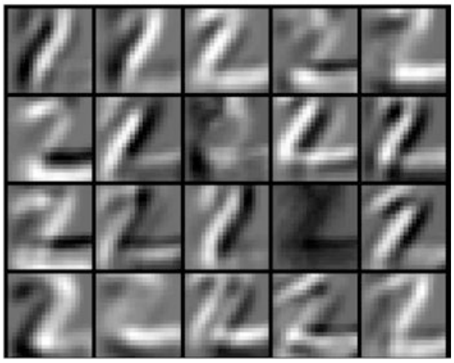


***Figure 6-22.** The results (y1) of the image processing of the convolution layer*

The fourth image shown in Figure 6-23 is what the `ReLU` function processed on the feature map from the convolution layer. The dark pixels of the previous image are removed, and the current images have mostly white pixels on the letter. This is a reasonable result when we consider the definition of the `ReLU` function.

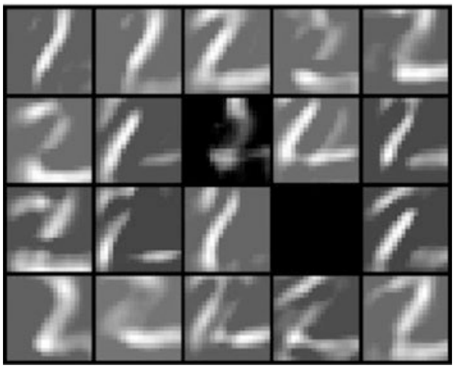


***Figure 6-23.** Image showing what the ReLU function processed on the feature map from the convolution layer*

Now, look at the Figure 6-22 again. It is noticeable that the image on third row fourth column contains a few bright pixels. After the `ReLU` operation, this image becomes completely dark. Actually, this is not a good sign because it fails to capture any feature of the input image of the `2`. It needs to be improved through more data and more training. However, the classification still functions, as the other parts of the feature map work properly.

Figure 6-24 shows the fifth result, which provides the images after the mean pooling process in which the ReLU layer produces. Each image inherits the shape of the previous image in a 10×10 pixel space, which is half the previous size. This shows how much the pooling layer can reduce the required resources.

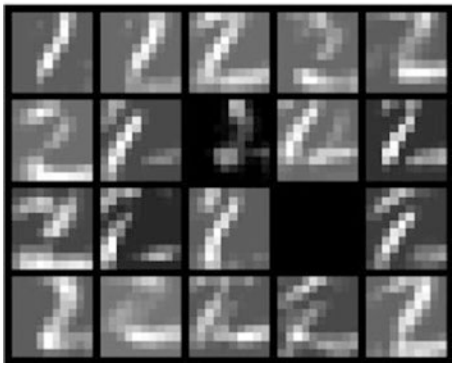


***Figure 6-24.** The images after the mean pooling process*

Figure 6-24 is the final result of the feature extraction neural network. These images are transformed into a one-dimensional vector and stored in the classification neural network.

This completes the explanation of the example code. Although only one pair of convolution and pooling layers is employed in the example; usually many of them are used in most practical applications. The more the small images that contain main features of the network, the better the recognition performance.

Summary

This chapter covered the following concepts:

- In order to improve the image recognition performance of Machine Learning, the feature map, which accentuates the unique features, should be provided rather than the original image. Traditionally, the feature extractor had been manually designed. ConvNet contains a special type of neural network for the feature extractor, of which the weights are determined via the training process.
- ConvNet consists of a feature extractor and classification neural network. Its deep layer architecture had been a barrier that made the training process difficult. However, since Deep Learning was introduced as the solution to this problem, the use of ConvNet has been growing rapidly.
- The feature extractor of ConvNet consists of alternating stacks of the convolution layer and the pooling layer. As ConvNet deals with two-dimensional images, most of its operations are conducted in a two-dimensional conceptual plane.
- Using the convolution filters, the convolution layer generates images that accentuate the features of the input image. The number of output images from this layer is the same as the number of convolution filters that the network contains. The convolution filter is actually nothing but a two-dimensional matrix.
- The pooling layer reduces the image size. It binds neighboring pixels and replaces them with a representative value. The representative value is either the maximum or mean value of the pixels.