Ch 07 합성곱 신경망(CNN)

이번 장의 주제는 합성곱 신경망(convolutional neural network, CNN)입니다.

7.1 전체 구조

우선 CNN의 네트워크 구조를 살펴보도록 하겠습니다. 기존에 Affine, ReLU 계층으로 연결된 계층에 합성곱(Conv), 풀링(Pooling) 계층이 추가되어 ‘Affine-ReLU’ 연결에서 ‘Conv-ReLU-(Pooling)’ 흐름으로 바뀝니다.

7.2 합성곱 계층

CNN에서는 패딩(padding), 스트라이드(stride)등 CNN 고유의 용어가 등장합니다. 각 계층 사이에는 3차원 데이터같이 입체적인 데이터가 흐른다는 점에서 완전연결 신경망과 다릅니다.

7.2.1 완전연결 계층의 문제점

지금까지는 Affine 계층을 이용하여 신경망을 구성했습니다. 이런 완전연결 계층의 문제점을 무엇일까요? ‘데이터의 형상이 무시’된다는 것입니다. MNIST 데이터셋을 예로 들면 지금까지 형상이 (1, 28, 28)인 데이터를 1줄로 세운 784개의 데이터를 입력했습니다.

이미지는 3차원 형상이며, 이 형상에는 공간적 정보가 담겨 있습니다. 예를 들어 공간적으로 가까운 픽셀은 값이 비슷하거나, RGB의 각 채널은 서로 밀접하게 관련되어 있거나, 거리가 먼 픽셀 간에는 별 연관이 없는 등, 3차원 속에서 의미를 갖는 본질적인 패턴이 숨어 있을 것입니다. 그러나 Affine layer은 형상을 무시하기 때문에 이런 정보를 살릴 수 없습니다.

이런 Affine layer와 달리 Conv layer는 형상을 유지합니다. 이미지를 3차원 데이터로 입력 받으면, 다음 계층에도 3차원 데이터로 전달합니다. 이런 성질 때문에 CNN은 형상을 가진 데이터를 이해할 (가능성이 있는) 것입니다.

CNN에서 Conv layer의 입출력 데이터를 특징 맵(feature map)이라고 하고, 입력 데이터를 입력 특징 맵(input feature map), 출력 데이터를 출력 특징 맵(output feature map)이라고 합니다. 이 책에서는 ‘입출력 데이터’와 ‘특징 맵’을 같은 의미로 사용한다고 합니다.

|  |  |
| --- | --- |
| 15 | 6 |
| 6 | 15 |

7.2.2 합성곱 연산

Conv layer 에선 합성곱 연산을 처리합니다. 합성곱 연산은 이미지 처리에서 말하는 필터 연산에 해당합니다. 예를 들어 설명하겠습니다.

∙

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 3 | 0 | 1 |

|  |  |  |
| --- | --- | --- |
| 2 | 0 | 1 |
| 0 | 1 | 2 |
| 1 | 0 | 2 |

필터

입력 데이터

합성곱 연산은 입력 데이터에 필터를 적용합니다. 연산 순서는 다음과 같습니다.

2

1

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 3 | 0 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 3 | 0 | 1 |

4

3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 3 | 0 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 3 | 0 | 1 |

1, 2, 3, 4 순서대로 윈도우(window)를 일정 간격으로 이동해가며 입력 데이터에 적용하여 입력과 필터에 대응하는 원소끼리 곱한 후 그 총합을 구합니다(이 계산을 단일 곱셈-누산(fused multiply-add, FMA)이라 합니다). 그리고 그 결과를 출력의 해당 장소에 저장합니다. CNN에서는 필터의 매개변수가 그동안의 ‘가중치’에 해당합니다. 편향은 스칼라 값으로 모든 원소에 더해집니다.

7.2.3 패딩

합성곱 연산을 수행하기 전에 입력 데이터 주변을 특정 값(0)으로 채우기도 합니다. 이를 패딩(padding)이라 하며, 출력 크기를 조정할 목적으로 사용합니다. 합성곱 연산을 되풀이할 수록 데이터의 형상이 줄어들기 때문에 주로 이런 사태를 막기 위해 사용됩니다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 1 | 2 | 3 | 0 |  |
|  | 0 | 1 | 2 | 3 |  |
|  | 3 | 0 | 1 | 2 |  |
|  | 2 | 3 | 0 | 1 |  |
|  |  |  |  |  |  |

전 합성곱 예의 입력 데이터에 패딩 1을 한 모습은 위와 같습니다. (4, 4) ∙ (3, 3) 합성곱 계산이 (6, 6) ∙ (3, 3)이 되어 출력이 (2, 2)에서 (4, 4)로 바뀝니다.

7.2.4 스트라이드

필터를 적용하는 위치의 간격을 스트라이드라고 합니다. 이 전까지는 모두 스트라이드가 1인 경우였고, 스트라이드를 2로 하면 필터를 적용하는 윈도우가 두 칸 씩 이동합니다. 스트라이드를 키우면 출력 크기는 작아집니다.

입력 크기(H, W), 필터 크기(FH, FW), 패딩(P), 스트라이드(S)에 따른 출력 크기(OH, OW)는 다음 식으로 계산할 수 있습니다.

위 식은 정수로 딱 떨어져야 한다는 것에 주의하도록 합시다.

7.2.5 3차원 데이터의 합성곱 연산

지금까지는 2차원 데이터를 살펴보았습니다. 이미지 데이터는 세로, 가로에 채널까지 더해진 3차원 데이터이므로 이번 절에는 채널을 고려한 3차원 데이터를 다루는 합성곱 연산을 살펴보도록 하겠습니다.

채널 3, 세로, 가로 4 인 데이터 (3, 4, 4)가 있고 채널 3 세로, 가로 3 인 필터(3, 3, 3)이 있을 때의 합성곱은 채널마다 합성곱을 수행한 후에 하나로 합칩니다. 고로 결과는 (1, 2, 2)가 됩니다. 3차원 합성곱에서 주의해야할 점은 입력 데이터와 필터의 채널이 같아야 한다는 것입니다.

7.2.6 블록으로 생각하기

W

C

OW

OH

∙

FW

FH

C

H

3차원 합성곱을 블록을 생각하면 위의 그림과 같이 표현할 수 있습니다. 3차원 데이터를 다차원 배열로 나타낼 때 (채널(channel), 높이(height), 너비(width)) 순서로 쓰면, 입력데이터는 (C, H, W), 필터는 (C, FH, FW), 출력데이터는(OH, OW)로 쓸 수 있습니다. 이 경우에 출력데이터는 한 장의 특징 맵입니다. 그렇다면 출력으로 다수의 채널을 보내려면 어떻게 해야 할까요?

필터의 개수를 늘리면 됩니다. 필터를 FN개로 늘리면 출력데이터는 (FN, OH, OW)인 블록으로 완성됩니다.

이런 이유로 합성곱 연산에서 필터의 가중치 데이터는 필터의 개수를 포함한 4차원 데이터이며 (출력 채널 수, 입력 채널 수, 높이, 너비) 순으로 씁니다.

합성곱 연산에도 편향은 더해집니다. 편향은 채널 하나에 값 하나로 구성되어 (FN, 1, 1)의 형상을 갖습니다. (FN, OH, OW)의 블록과는 브로드캐스트를 통해 (FN, OH, OW) 블록의 대응 채널의 원소 모두에 더해집니다.

7.2.7 배치 처리

신경망 처리에서 입력 데이터는 배치라는 한 덩이로 묶여 처리되었습니다. 이 방식을 통해 처리 효율을 높이고, 미니배치 방식의 학습도 지원하였습니다.

합성곱 연산에 배치 처리를 하고자 합니다. 이를 위해 데이터의 차원을 하나 늘려 4차원의 데이터로 표현합니다. (데이터 수, 채널 수, 높이, 너비) 순으로 저장합니다.

배치 처리 시의 데이터 흐름을 보면 각 데이터의 선두에 배치용 차원을 추가했기 때문에 4차원 형상을 가진 채 계층을 타고 흐릅니다.

(N, C, H, W) \* (FN, C, FH, FW) -> (N, FN, OH, OW) + (FN, 1, 1) -> (N, FN, OH, OW)

신경망에 4차원 데이터가 하나 흐를 때마다 데이터 N개에 대한 합성곱 연산이 이뤄지는 것을 볼 수 있습니다. 이는 N회 분의 처리를 한 번에 수행하는 것입니다.

7.3 풀링 계층

풀링은 세로, 가로 방향의 공간을 줄이는 연산입니다. 다음 그림과 같이 적용됩니다.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 1 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 4 | 0 | 1 |

|  |  |
| --- | --- |
| 2 |  |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 1 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 4 | 0 | 1 |

|  |  |
| --- | --- |
| 2 | 3 |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 1 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 4 | 0 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 1 | 0 |
| 0 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |
| 2 | 4 | 0 | 1 |

|  |  |
| --- | --- |
| 2 | 3 |
| 4 |  |

|  |  |
| --- | --- |
| 2 | 3 |
| 4 | 2 |

2x2 최대 풀링(max pooling)을 스트라이드 2로 처리하는 순서입니다. 최대 풀링 외에도 평균 풀링(average pooling)등이 있습니다. 이미지 인식 분야에서는 주로 최대 풀링을 사용합니다.

7.3.1 풀링 계층의 특징

풀링 계층은 다음의 특징을 가집니다.

1. 학습해야 할 매개변수가 없다

풀링 계층은 합성곱 계층과 달리 학습해야 할 매개변수가 없습니다. 풀링은 대상 영역에서 최댓값이나 평균을 취하는 명확한 처리이므로 특별히 학습할 것이 없습니다.

2. 채널 수가 변하지 않는다.

풀링 연산은 입력 데이터의 채널 수 그대로 출력 데이터로 보냅니다. 채널마다 독립적으로 계산하기 때문입니다.

3. 입력의 변화에 영향을 적게 받는다(강건하다).

입력 데이터가 조금 변해도 풀링의 결과는 잘 변하지 않습니다. 입력 데이터의 차이를 풀링이 흡수해 사라지는 경우도 있습니다.

7.4 합성곱/풀링 계층 구현하기

이번 절에서는 전에 신경망을 구현했던 것처럼 클래스에 forward와 backward 메서드를 추가하여 구현해보도록 하겠습니다.

7.4.1 4차원 배열

CNN에서 계층 사이를 흐르는 데이터는 4차원입니다. 연산을 편히 하기 위해 im2col이라는 ‘트릭’을 사용하도록 하겠습니다.

7.4.2 im2col로 데이터 전개하기

im2col은 입력 데이터를 가중치 계산하기 좋게 펼치는 함수입니다. 배치 안의 데이터 수까지 포함한 4차원 입력 데이터에 im2col을 적용하면 2차원 행렬로 바뀝니다. 입력 데이터에서 필터를 적용하는 영역을 한 줄로 늘어놓습니다. 이 전개를 필터를 적용하는 모든 영역에서 수행하는 게 im2col입니다.

적용할 때 영역이 겹치는 경우가 대부분이기 때문에 전개한 후의 원소 수가 원래 블록의 원소 수보다 많아져 메모리를 더 많이 소비하는 단점이 있습니다. 하지만 행렬 계산에 고도로 최적화 된 선형 대수 라이브러리를 사용하여 큰 행렬 곱셈을 빠르게 계산할 수 있기 때문에 효율을 높일 수 있습니다.

im2col로 입력 데이터를 전개하고 필터를 1열로 전개해, 두 행렬의 곱을 계산하는 것이 합성곱 계층에서 하는 일입니다. 이 방식으로 출력되는 결과는 2차원 행렬이기 때문에 4차원으로 변형(reshape)해야 합니다.

7.4.3 합성곱 계층 구현하기

책에서 미리 제공하는 im2col 함수를 사용하여 합성곱 계층을 구현해보겠습니다. 함수의 인터페이스는 다음과 같습니다.

im2col(input\_data, filter\_h, filter\_w, stride=1, pad=0)

그러면 실제로 im2col 함수를 사용해 합성곱 계층을 구현해봅시다.

class Convolution:  
 def \_\_init\_\_(self, W, b, stride=1, pad=0):  
 self.W = W  
 self.b = b  
 self.stride = stride  
 self.pad = pad  
  
 # backward 시 사용  
 self.x = None  
 self.col = None  
 self.col\_W = None  
  
 # 가중치와 편향 매개변수의 기울기  
 self.dW = None  
 self.db = None  
  
 def forawrd(self, x):  
 FN, C, FH, FW = self.W.shape  
 N, C, H, W = x.shape  
 out\_h = 1 + int((H + 2\*self.pad - FH) / self.stride)  
 out\_w = 1 + int((H + 2\*self.pad - FW) / self.stride)  
  
 col = im2col(x, FH, FW, self.stride, self.pad)  
 col\_W = self.W.reshape(FN, -1).T  
  
 out = np.dot(col, col\_W) + self.b  
 out = out.reshape(N, out\_h, out\_w, -1).transpose(0, 3, 1, 2)  
  
 self.x = x  
 self.col = col  
 self.col\_W = col\_W

return out  
  
 def backward(self, dout):  
 FN, C, FH, FW = self.W.shape  
 dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN)  
  
 self.db = np.sum(dout, axis=0)  
 self.dW = np.dot(self.col.T, dout)  
 self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)  
  
 dcol = np.dot(dout, self.col\_W.T)  
 dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)  
  
 return dx

순전파 구현에서 주의할 점은 데이터의 형상입니다. im2col 함수를 통해 4차원 데이터를 2차원 데이터로 전개하기 때문에, 필터도 reshape를 통해 맞춰주어야 합니다. 그리고 마지막에 출력 데이터를 적절한 형상으로 바꿔줘야 합니다. reshape을 통해 (N, out\_h, out\_w, -1)의 형상으로 바뀌기 때문에 transpose 함수를 사용해 (N, C, H, W)의 순서로 다시 바꿔줘야 합니다.

합성곱 계층의 역전파는 데이터의 형태를 제외하고는 Affine 계층과 다른 것이 없습니다. 주의할 것은 col2im 함수를 통해 im2col을 역으로 수행해야 합니다.

7.4.4 풀링 계층 구현하기

1

2

0

1

3

0

2

4

1

0

3

2

4

2

0

1

1

2

3

0

0

1

2

4

1

0

4

2

3

2

0

1

위와 같은 방식으로 전개한 후 행별 최댓값을 구하고 적절한 형상으로 reshape하면 됩니다.

class Pooling:  
 def \_\_init\_\_(self, pool\_h, pool\_w, stride=1, pad=0):  
 self.pool\_h = pool\_h  
 self.pool\_w = pool\_w  
 self.stride = stride  
 self.pad = pad  
  
 self.x = None  
 self.arg\_max = None  
  
 def forward(self, x):  
 N, C, H, W = x.shape  
 out\_h = int(1 + (H - self.pool\_h) / self.stride)  
 out\_W = int(1 + (W - self.pool\_w) / self.stride)  
  
 col = im2col(x, self.pool\_h, self.pool\_w, self.stride, self.pad)  
 col = col.reshape(-1, self.pool\_h \* self.pool\_w)  
  
 arg\_max = np.argmax(col, axis=1)  
 out = np.max(col, axis=1)  
 out = out.reshape(N, out\_h, out\_W, C).transpose(0, 3, 1, 2)  
  
 self.x = x  
 self.arg\_max = arg\_max  
  
 return out  
  
 def backward(self, dout):  
 dout = dout.transpose(0, 2, 3, 1)  
  
 pool\_size = self.pool\_h \* self.pool\_w  
 dmax = np.zeros((dout.size, pool\_size))  
 dmax[np.arange(self.arg\_max.size), self.arg\_max.flatten()] = dout.flatten()  
 dmax = dmax.reshape(dout.shape + (pool\_size,))  
  
 dcol = dmax.reshape(dmax.shape[0] \* dmax.shape[1] \* dmax.shape[2], -1)  
 dx = col2im(dcol, self.x.shape, self.pool\_h, self.pool\_w, self.stride, self.pad)  
  
 return dx

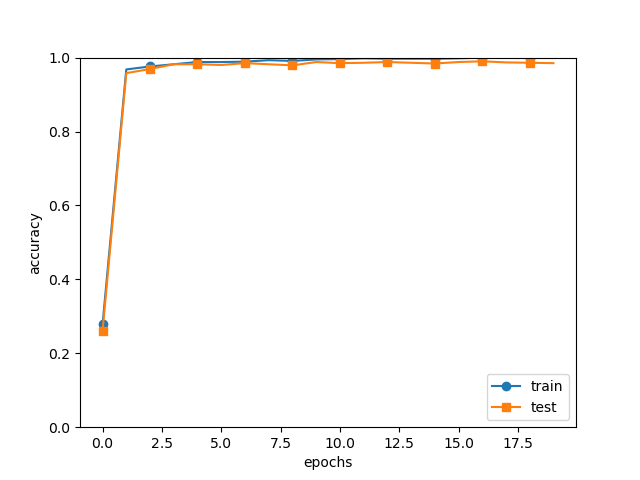
7.5 CNN 구현해보기

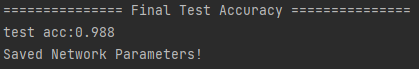
이번에 구현해볼 CNN 네트워크는 ‘Convolution-ReLU-Pooling-Affine-ReLU-Affine-Softmax’순으로 흐릅니다.

import sys, os  
sys.path.append(os.pardir)  
import pickle  
import numpy as np  
from collections import OrderedDict  
from common.layers import \*  
from common.gradient import numerical\_gradient  
  
  
class SimpleConvNet:  
 def \_\_init\_\_(self, input\_dim=(1, 28, 28),  
 conv\_param={'filter\_num': 30, 'filter\_size': 5, 'pad': 0, 'stride': 1},  
 hidden\_size=100, output\_size=10, weight\_init\_std=0.01  
 ):  
 filter\_num = conv\_param['filter\_num']  
 filter\_size = conv\_param['filter\_size']  
 filter\_pad = conv\_param['pad']  
 filter\_stride = conv\_param['stride']  
 input\_size = input\_dim[1]  
 conv\_output\_size = (input\_size - filter\_size + 2\*filter\_pad) / filter\_stride + 1  
 pool\_output\_size = int(filter\_num \* (conv\_output\_size/2) \* (conv\_output\_size/2))  
  
 # 가중치 초기화  
 self.params = {}  
 self.params['W1'] = weight\_init\_std \* \  
 np.random.randn(filter\_num, input\_dim[0], filter\_size, filter\_size)  
 self.params['b1'] = np.zeros(filter\_num)  
 self.params['W2'] = weight\_init\_std \* \  
 np.random.randn(pool\_output\_size, hidden\_size)  
 self.params['b2'] = np.zeros(hidden\_size)  
 self.params['W3'] = weight\_init\_std \* \  
 np.random.randn(hidden\_size, output\_size)  
 self.params['b3'] = np.zeros(output\_size)  
  
 # 계층 생성  
 self.layers = OrderedDict()  
 self.layers['Conv1'] = Convolution(self.params['W1'], self.params['b1'],  
 conv\_param['stride'], conv\_param['pad'])  
 self.layers['Relu1'] = Relu()  
 self.layers['Pool1'] = Pooling(pool\_h=2, pool\_w=2, stride=2)  
 self.layers['Affine1'] = Affine(self.params['W2'], self.params['b2'])  
 self.layers['Relu2'] = Relu()  
 self.layers['Affine2'] = Affine(self.params['W3'], self.params['b3'])  
  
 self.last\_layer = SoftmaxWithLoss()  
  
 def predict(self, x):  
 for layer in self.layers.values():  
 x = layer.forward()  
  
 return x  
  
 def loss(self, x, t):  
 y = self.predict(x)  
 return self.last\_layer.forward(y, t)  
  
 def accuracy(self, x, t, batch\_size=100):  
 if t.ndim != 1 : t = np.argmax(t, axis=1)  
  
 acc = 0.0  
  
 for i in range(int(x.shape[0] / batch\_size)):  
 tx = x[i\*batch\_size:(i+1)\*batch\_size]  
 tt = t[i\*batch\_size:(i+1)\*batch\_size]  
 y = self.predict(tx)  
 y = np.argmax(y, axis=1)  
 acc += np.sum(y==tt)  
  
 return acc / x.shape[0]  
  
 def numerical\_gradient(self, x, t):  
 loss\_w = lambda w: self.loss(x, t)  
  
 grads = {}  
 for idx in (1, 2, 3):  
 grads['W' + str(idx)] = numerical\_gradient(loss\_w, self.params['W' + str(idx)])  
 grads['b' + str(idx)] = numerical\_gradient(loss\_w, self.params['b' + str(idx)])  
  
 return grads  
  
 def gradient(self, x, t):  
 # forward  
 self.loss(x, t)  
  
 # backward  
 dout = 1  
 dout = self.last\_layer.backward(dout)  
  
 layers = list(self.layers.values())  
 layers.reverse()  
 for layer in layers:  
 dout = layer.backward(dout)  
  
 grads = {}  
 grads['W1'], grads['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db  
 grads['W2'], grads['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db  
 grads['W3'], grads['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db  
  
 return grads  
  
 def save\_params(self, file\_name='params.pkl'):  
 params = {}  
 for key, val in self.params.items():  
 params[key] = val  
 with open(file\_name, 'wb') as f:  
 pickle.dump(params, f)  
  
 def load\_params(self, file\_name='params.pkl'):  
 with open(file\_name, 'rb') as f:  
 params = pickle.load(f)  
 for key, val in params.items():  
 self.params[key] = val  
  
 for i, key in enumerate(['Conv1', 'Affine1', 'Affine2']):  
 self.layers[key].W = self.params['W' + str(i+1)]  
 self.layers[key].b = self.params['b' + str(i+1)]

CNN을 구현해보았습니다. 이제 MNIST 데이터셋을 학습해볼 차례입니다.

# coding: utf-8  
import sys, os  
  
sys.path.append(os.pardir)   
import numpy as np  
import matplotlib.pyplot as plt  
from dataset.mnist import load\_mnist  
from simple\_convnet\_trial import SimpleConvNet  
from common.trainer import Trainer  
  
# 데이터 읽기  
(x\_train, t\_train), (x\_test, t\_test) = load\_mnist(flatten=False)  
  
max\_epochs = 20  
  
network = SimpleConvNet(input\_dim=(1, 28, 28),  
 conv\_param={'filter\_num': 30, 'filter\_size': 5, 'pad': 0, 'stride': 1},  
 hidden\_size=100, output\_size=10, weight\_init\_std=0.01)  
  
trainer = Trainer(network, x\_train, t\_train, x\_test, t\_test,  
 epochs=max\_epochs, mini\_batch\_size=100,  
 optimizer='Adam', optimizer\_param={'lr': 0.001},  
 evaluate\_sample\_num\_per\_epoch=1000)  
trainer.train()  
  
# 매개변수 보존  
network.save\_params("params.pkl")  
print("Saved Network Parameters!")  
  
# 그래프 그리기  
markers = {'train': 'o', 'test': 's'}  
x = np.arange(max\_epochs)  
plt.plot(x, trainer.train\_acc\_list, marker='o', label='train', markevery=2)  
plt.plot(x, trainer.test\_acc\_list, marker='s', label='test', markevery=2)  
plt.xlabel("epochs")  
plt.ylabel("accuracy")  
plt.ylim(0, 1.0)  
plt.legend(loc='lower right')  
plt.show()



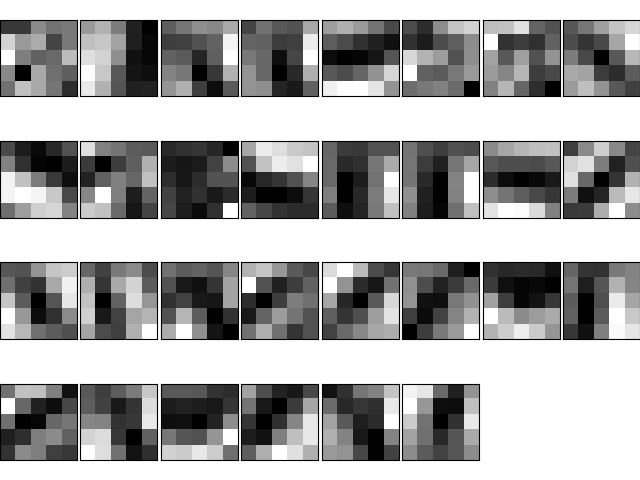
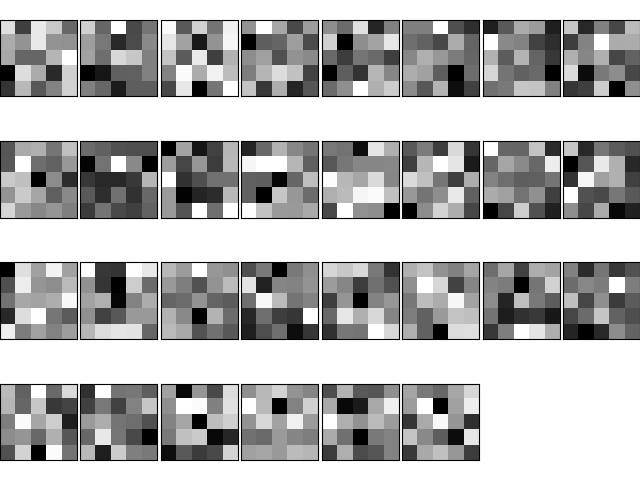


학습과 시험 두 데이터 모두 98%가 넘는 정확도를 보이고 있습니다.

7.6 CNN 시각화하기

7.6.1 1번째 층의 가중치 시각화하기

1번째 층의 가중치의 형상은 (30, 1, 5, 5)로 필터의 크기가 5x5, 채널이 1개입니다. 채널이 1개라는말은 회색조 이미지로 시각화 할 수 있다는 뜻입니다. 다음은 책의 예제 소스를 사용해 출력한 필터의 이미지입니다. 왼쪽이 학습 전, 오른쪽이 학습 후입니다.



학습 전 필터는 무작위로 초기화 되었기 때문에 특정한 패턴을 가지고 있지 않습니다. 반면에 학습 후 필터는 규칙성(가로, 세로, 대각선 등)을 가진 이미지가 되었습니다. 필터는 학습을 통해 이미지의 에지(색상이 바뀐 경계선)와 블롭(국소적으로 덩어리진 영역)에 반응합니다. 예를 들어 왼쪽 절반이 검은색이고 오른쪽 절반이 흰색인 필터는 이미지에서 세로 방향의 에지에 반응하는 필터입니다. 합성곱 계층의 필터는 에지나 블롭 등의 원시적인 정보를 추출해 뒤 계층에 전달하는 것이 CNN에서 일어나는 일입니다.

7.6.2 층 깊이에 따른 추출 정보 변화

1층이 아닌 다른 층계에선 어떤 정보가 추출될까요? 딥러닝 시각화에 관한 연구에 따르면, 추출되는 정보는 더 추상화된다고 합니다. 1번째 층은 에지와 블롭, 3번째 층은 텍츠쳐, 5번째 층은 사물의 일부, 마지막 계층은 사물의 클래스에 뉴런이 반응한다고 합니다. 즉, 층이 깊어질수록 뉴런이 반응하는 대상이 단순한 모양에서 ‘고급’ 정보로 변화합니다.

7.7 대표적인 CNN

7.7.1 LeNet

LeNet은 1998년에 제안된 손 글씨 숫자를 인식하는 네트워크입니다. 합성곱과 풀링계층을 반복하고, 마지막으로 완전연결 계층을 거쳐 결과를 출력합니다.

현재의 CNN과 비교해보면 활성화 함수를 sigmoid 함수를 사용한다는 것, 서브샘플링을 하여 중간에 데이터의 크기를 줄이는 것 정도가 다릅니다. 20년 전에 제안된 CNN이라는 점을 생각하면 대단합니다.

7.7.2 AlexNet

AlexNet은 딥러닝 열풍을 일으키는 데 큰 역할을 했습니다. 이 신경망은 활성화 함수로 ReLU 함수를 이용하고, LRN(Local Response Normalization)이라는 국소적 정규화를 실시하는 계층을 이용하고, 드롭 아웃을 사용한다는 것을 제외하면 LeNet과 크게 다르지 않습니다.

대량의 데이터를 누구나 얻을 수 있고, 병렬 계산에 특화된 GPU가 보급되면서 진보를 이뤘기 때문에 딥러닝은 큰 발전을 할 수 있었습니다.