Convolutional Neural Network

Convolutional Neural Network

- **Introduction of CNN**
- **Why CNN?**
- **Structure of CNN**
- **Code of CNN**

Introduction of CNN

Introduction of CNN

- 이상적인 머신 러닝
  - 학습 데이터(training data)만 적절히 많이 넣어주면 알아서 해주는 것

- 현실적인 머신 러닝
  - 기존 지식(prior knowledge)를 이용해 네트워크의 구조를 특수한 형태로 변형

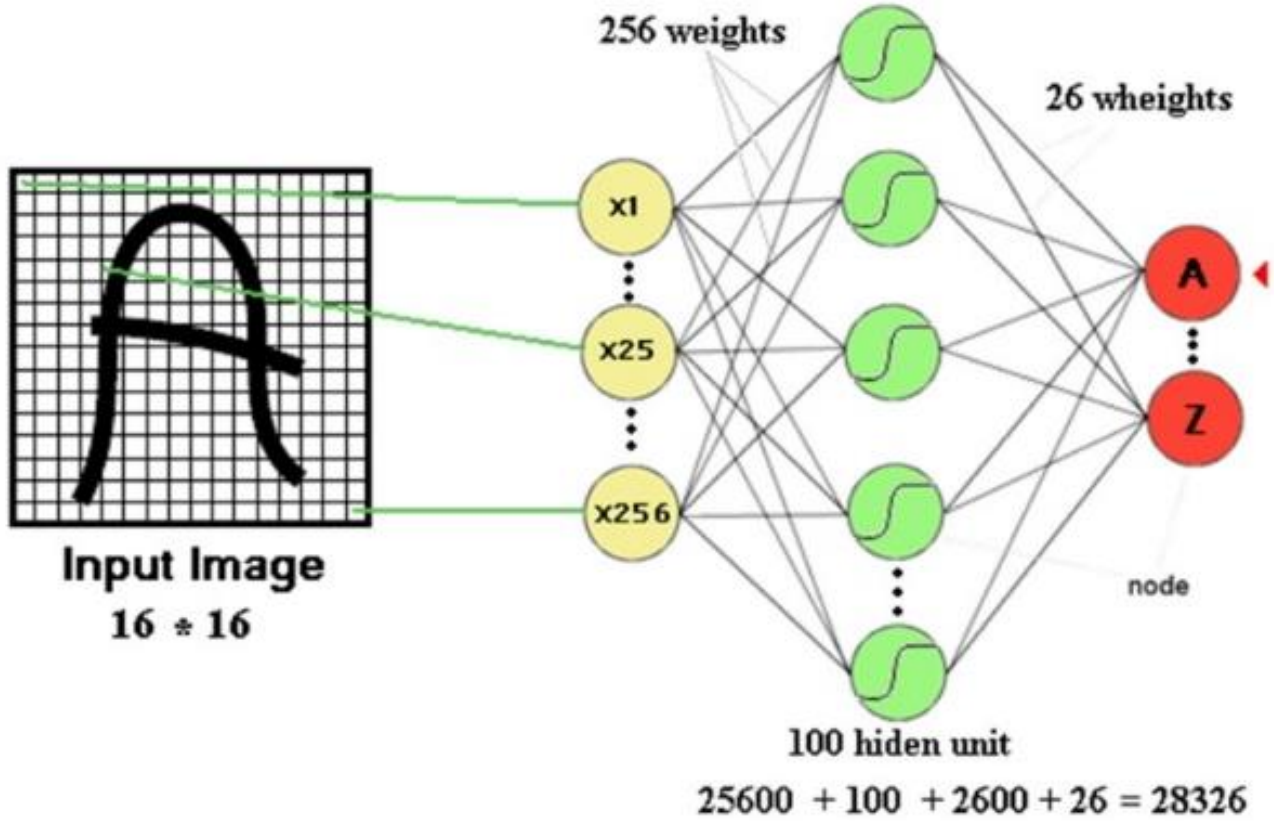
=> 2D 이미지가 갖는 특성을 최대한 살릴 수 있는 방법은???

Introduction of CNN

- 기존 fully-connected Neural Network의 한계

- 파라미터의 개수
  - Weight in 1-layer : 256*100=25,600
  - Weight in 2-layer : 100*26 = 2,600
  - Bias in 1-layer : 100
  - Bias in 2-layer : 26

- 16*16의 이미지이면 눈으로 물체의 이미지를 제대로 보기 힘들 정도로 작은 이미지

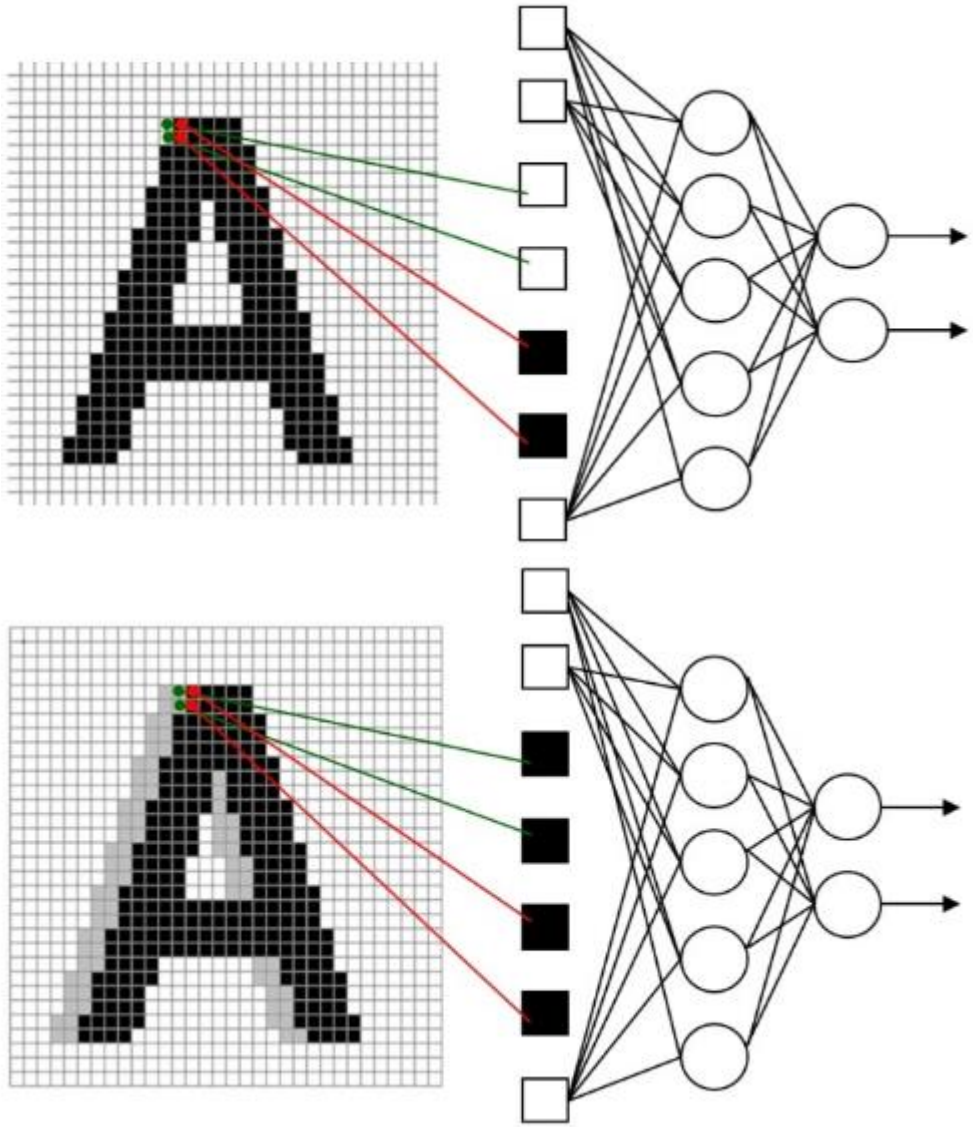


Introduction of CNN

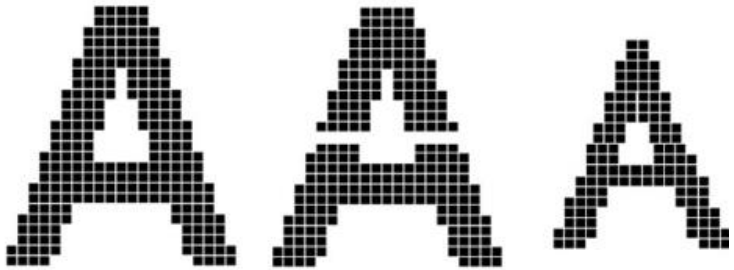
- 기존 fully-connected Neural Network의 한계

- 데이터 처리의 문제
  - 글자의 경우 단지 2픽셀 값만 달라지거나 이동하더라도 새로운 학습 데이터로 처리를 해줘야 함
  - 글자의 변형(이동, 축소/확대, 왜곡)이 조금만 생기더라도 그에 대한 새로운 학습 데이터를 넣어주지 않으면 좋은 결과를 기대하기 어렵다.

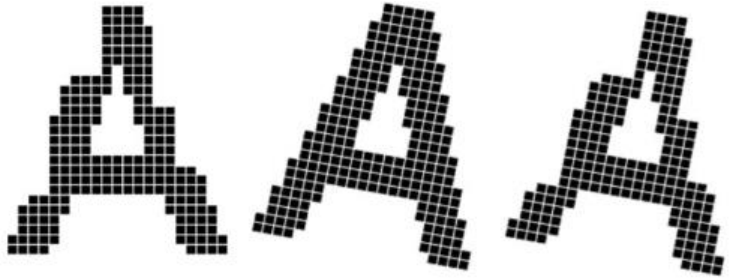
이동



축소/확대



왜곡



Introduction of CNN

- 기존 fully-connected Neural Network의 한계

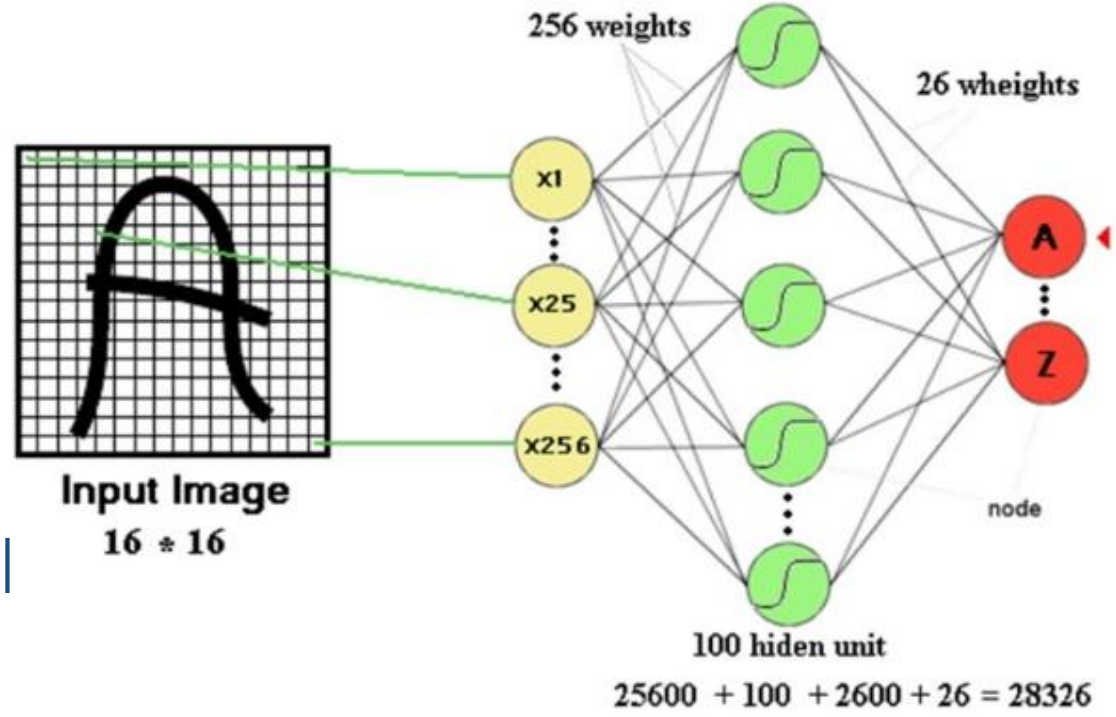
- 이미지 분류 작업

- 3가지 측면의 문제점
  - 학습 시간

⇒엄청나게 많은 학습데이터가 필요하고, 엄청나게 많은 시간이 소모된다.

  - 망의 크기
  - 변수의 개수

$=>256^{16*16} = 256^{256}$어마어마한 패턴이 나오므로 이것을 학습하기 위해선 거대한 신경망의 크기가 필요



Introduction of CNN

- 기존 fully-connected Neural Network의 한계를 해결 할 방법
- 2D이미지의 특성을 최대한 살릴 수 있는 방법

=> Convolution 구조!

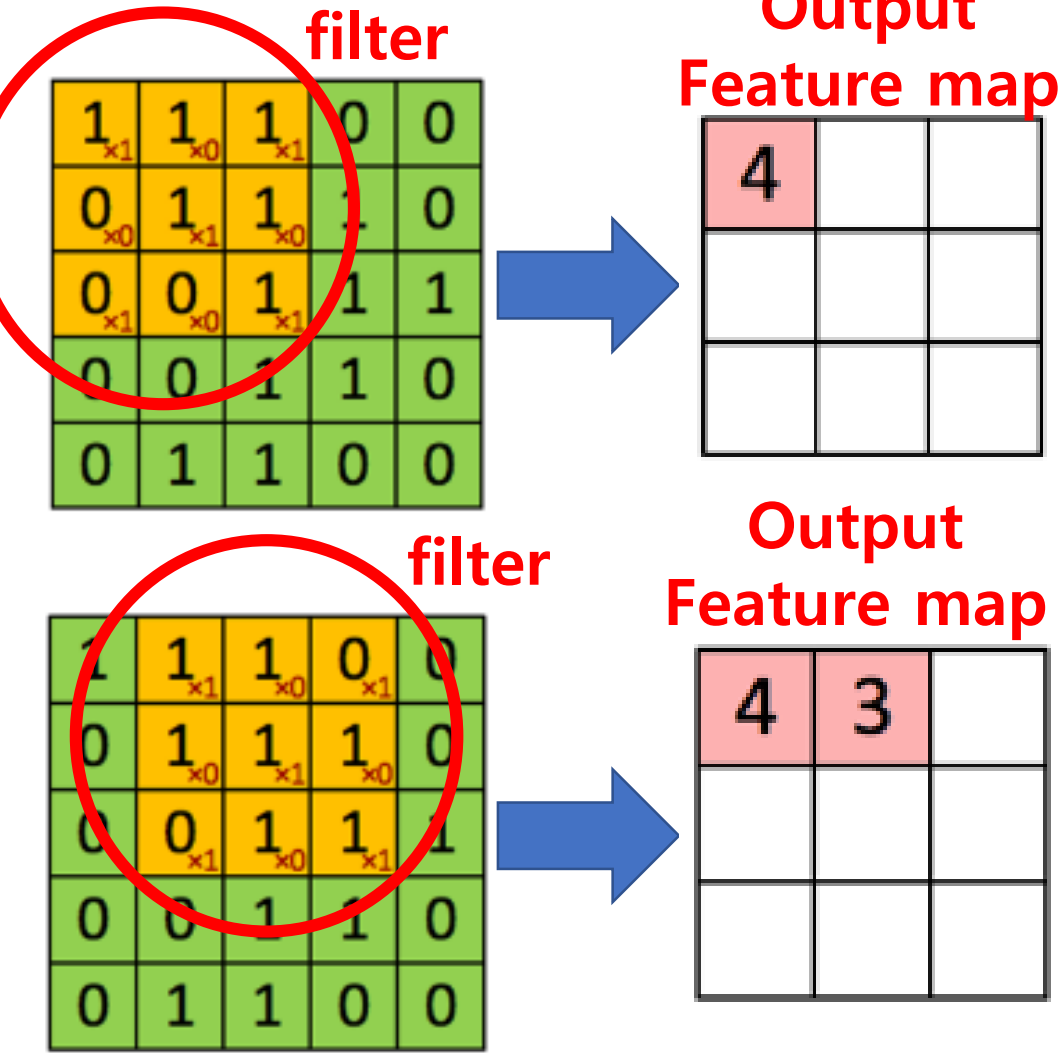
Why CNN?

Why CNN ?

- Convolution 이란?
- 영상 처리 분야에서 convolution은 주로 filter 연산에 사용이 됨
- 영상에서 특정 feature들을 추출하기 위한 필터를 구현 할 때 사용
- 3x3이나 그 이상의 mask를 영상 전체에 대해 반복적으로 수행하면 mask값에 따라 적정한 결과를 얻을 수 있다.

Why CNN ?

- Convolution 이란?



여러 특징을 추출 가능
특정 특징을 부각 시켜 줌

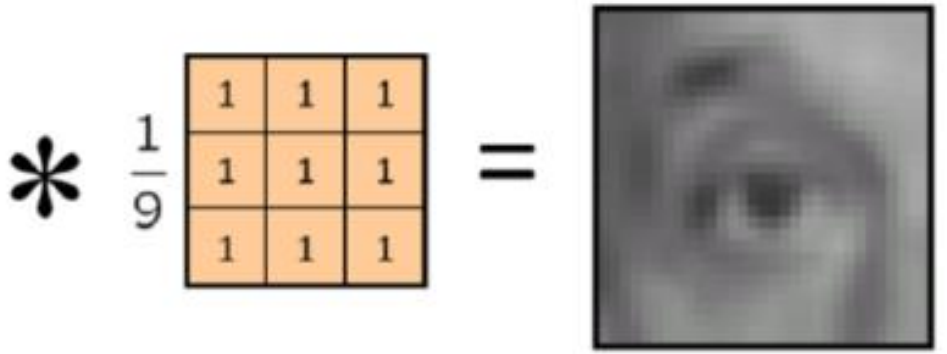
Why CNN ?

- Convolution 을 Neural Network에 적용하면

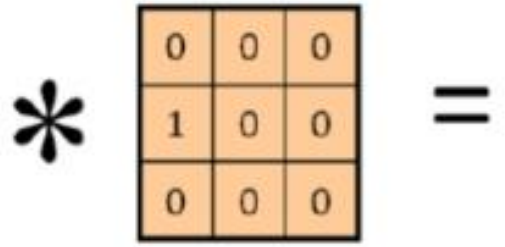
- **Locality (Local Connectivity)**
  - 여러 개의 filter를 convolution 연산을 통해 적용하면 다양한 local 특징들을 추출 해 낼 수 있다.

- **Shared Weights**
  - convolution 개념을 도입하면 weight를 공유하게 되어, 변수의 수를 획기적으로 줄일 수 있다.

Original



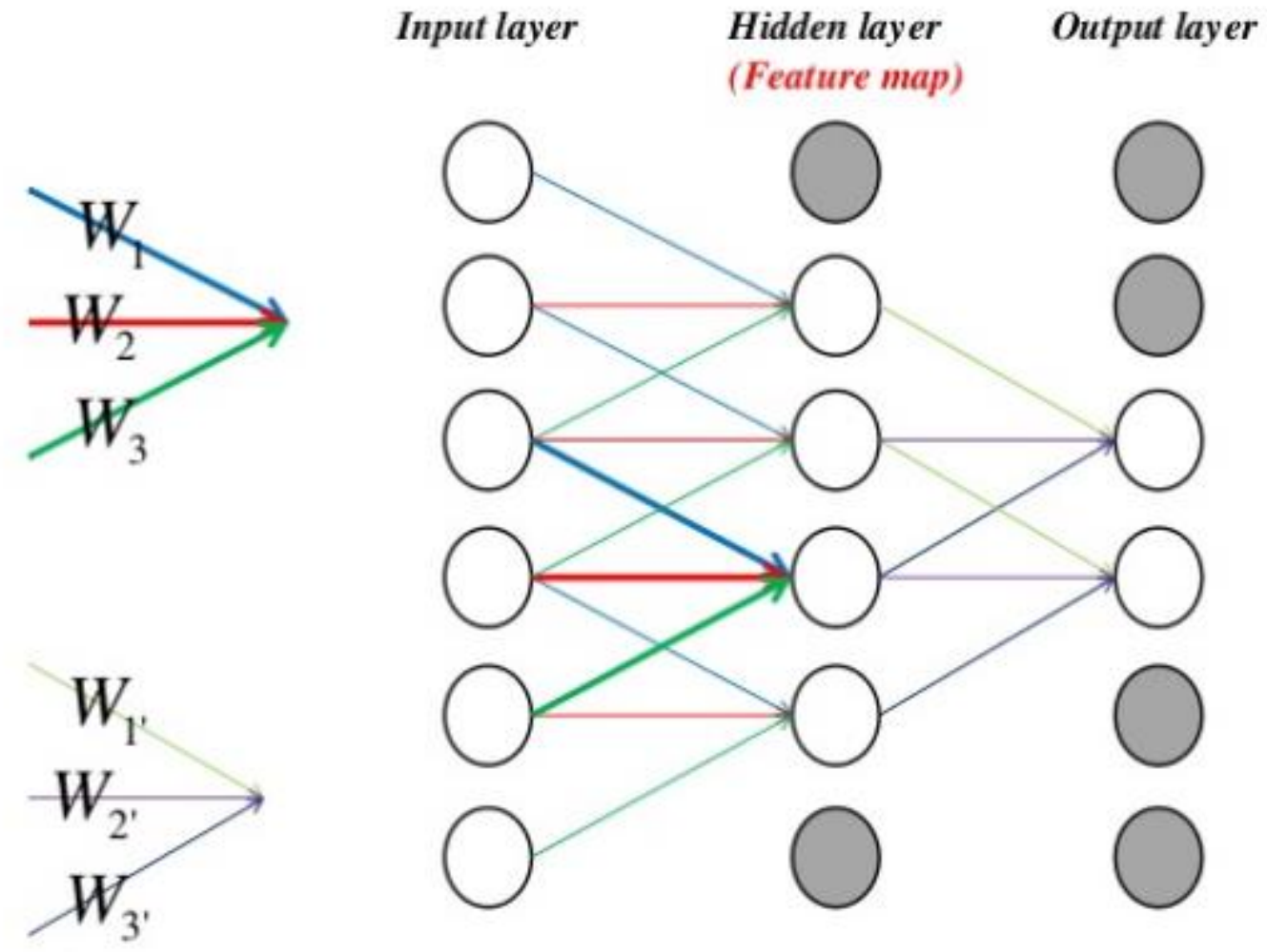
Blur (with a mean filter)



Shifted left
By 1 pixel

Why CNN ?

- Convolution Neural Network를 통해 파라미터 개수가 줄어든다.
- Shared Weights



Why CNN ?

- Convolution Neural Network를 통해 파라미터 개수가 줄어든다.
- Shared Weights

Input 노드 번호

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

*

Filter1

W1	W2	W3
W4	W5	W6
W7	W8	W9

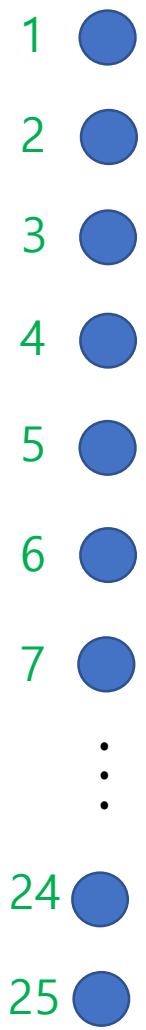
Filter2

W1	W2	W3
W4	W5	W6
W7	W8	W9

=

Output 노드 번호

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



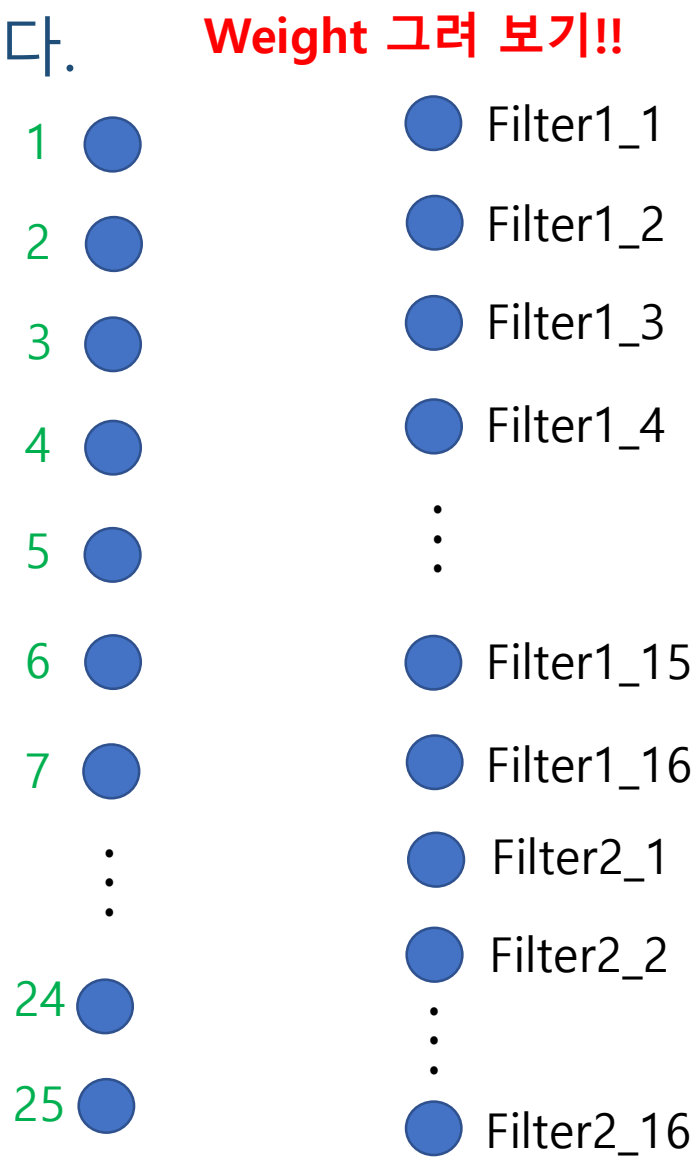
Weight 그려 보기!!

Why CNN ?

- Convolution Neural Network를 통해 파라미터 개수가 줄어든다.
- Shared Weights

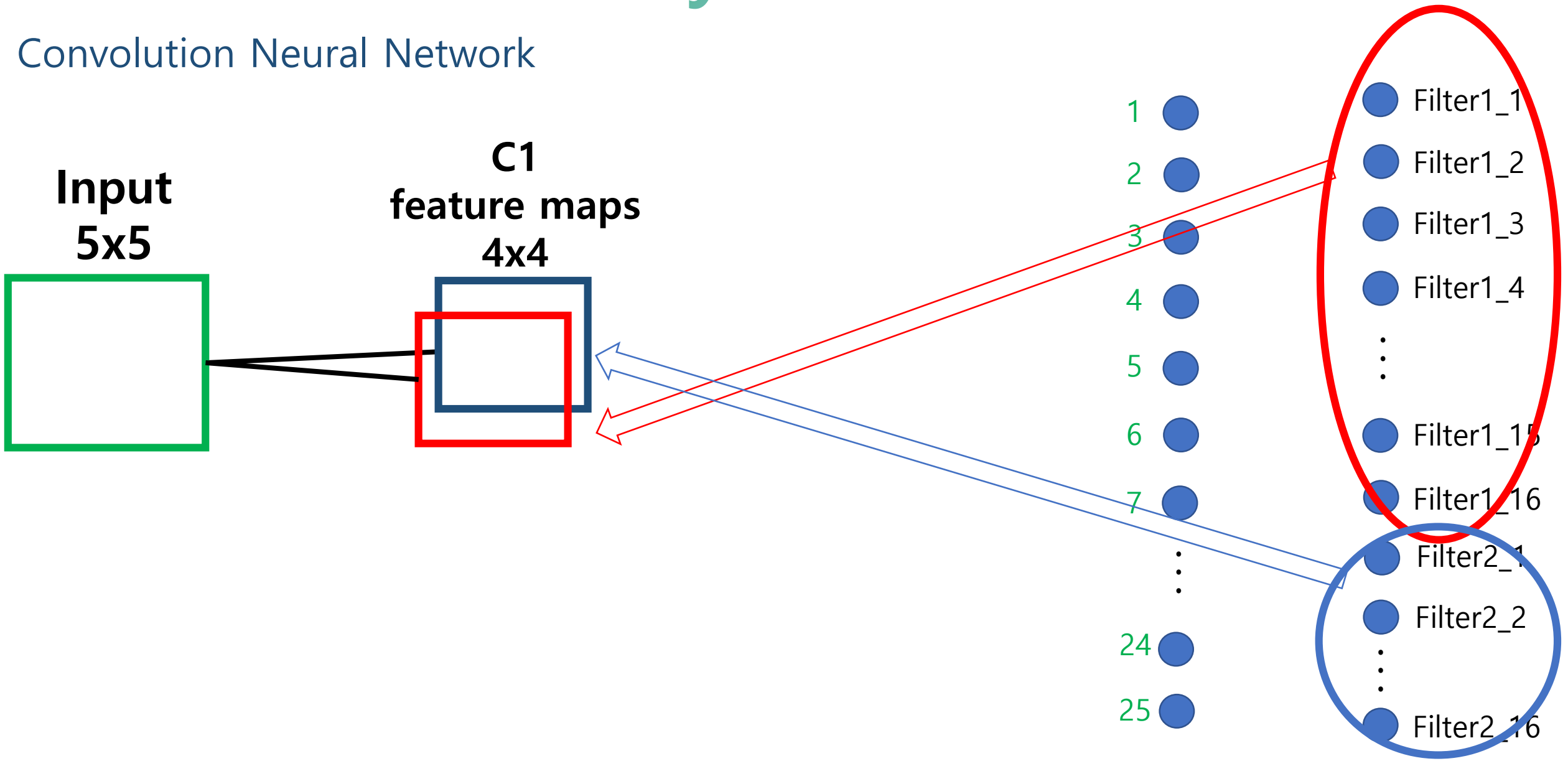
Fully-connected neural network일 때 weight 개수는?

Convolution neural network일 때 weight 개수는?



Why CNN ?

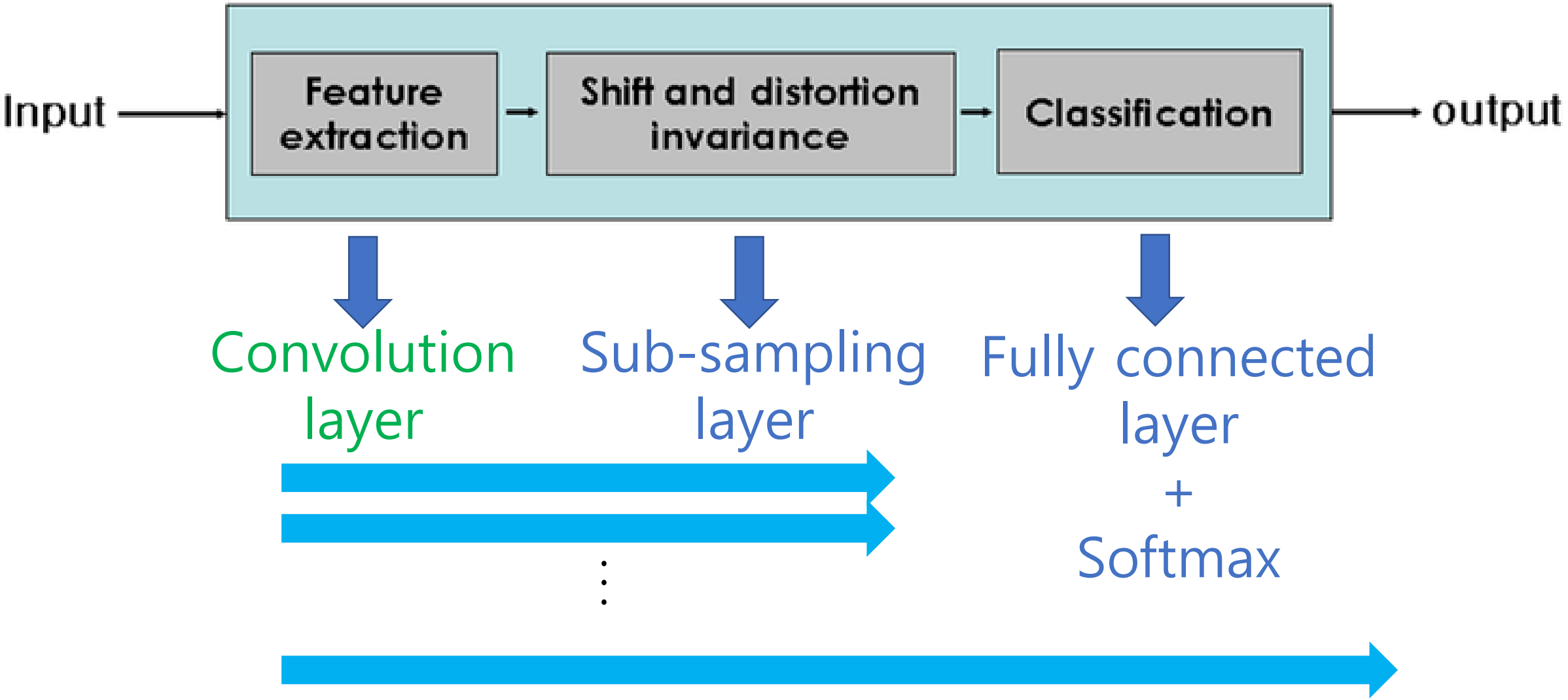
- Convolution Neural Network



Structure of CNN

Structure of CNN

- Convolution Neural Network



Structure of CNN

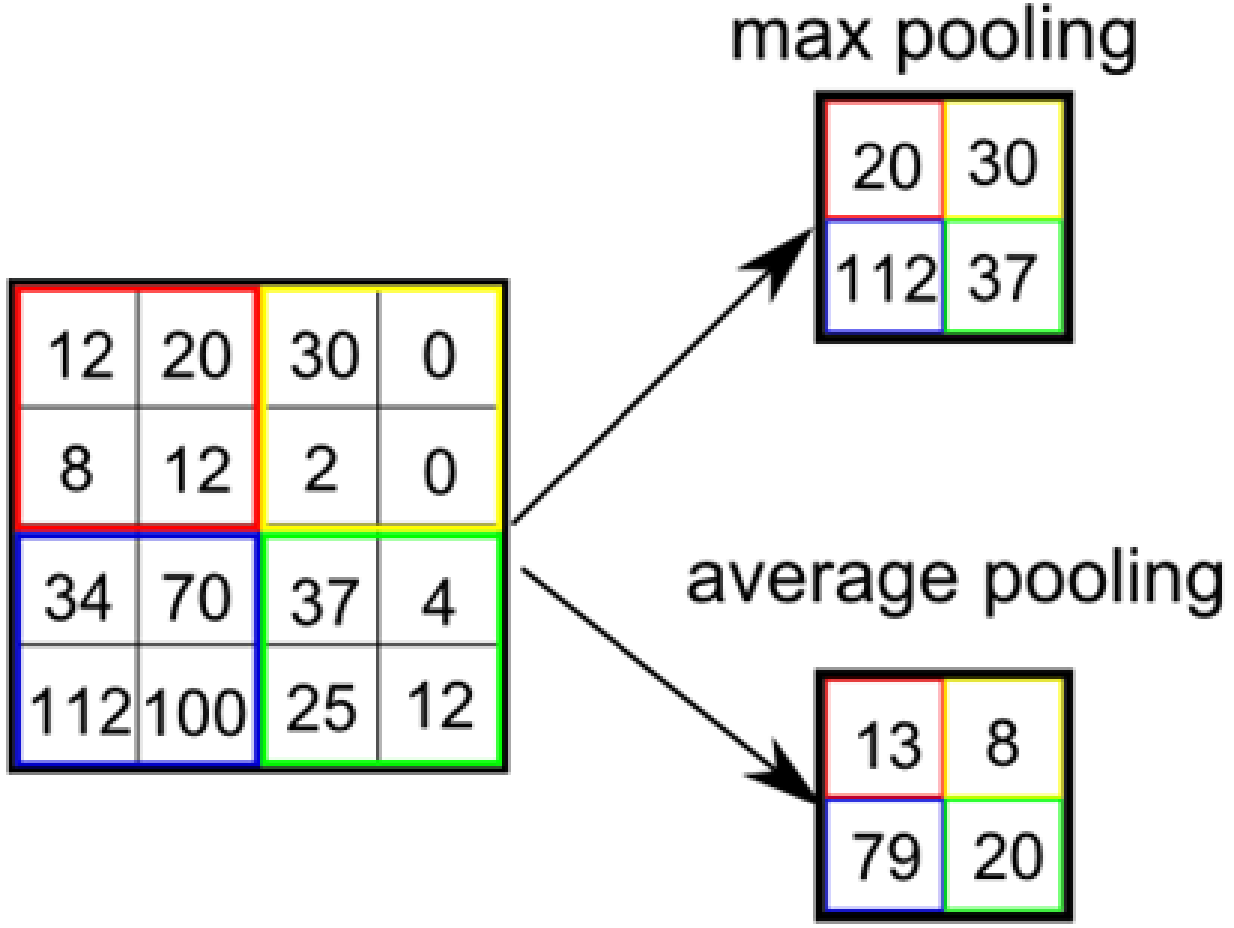
- Sub-sampling layer

- **신경세포학적으로 살펴보면 통상적으로 강한 신호만 전달하고 나머지는 무시**

- **이와 비슷한 과정을 CNN에서는 max pooling 과정을 거친다**

- **좀 더 강하고 global한 특징을 추출**

- **연산 량을 더 줄이는 역할을 함**



Structure of CNN

- Fully-connected layer + softmax

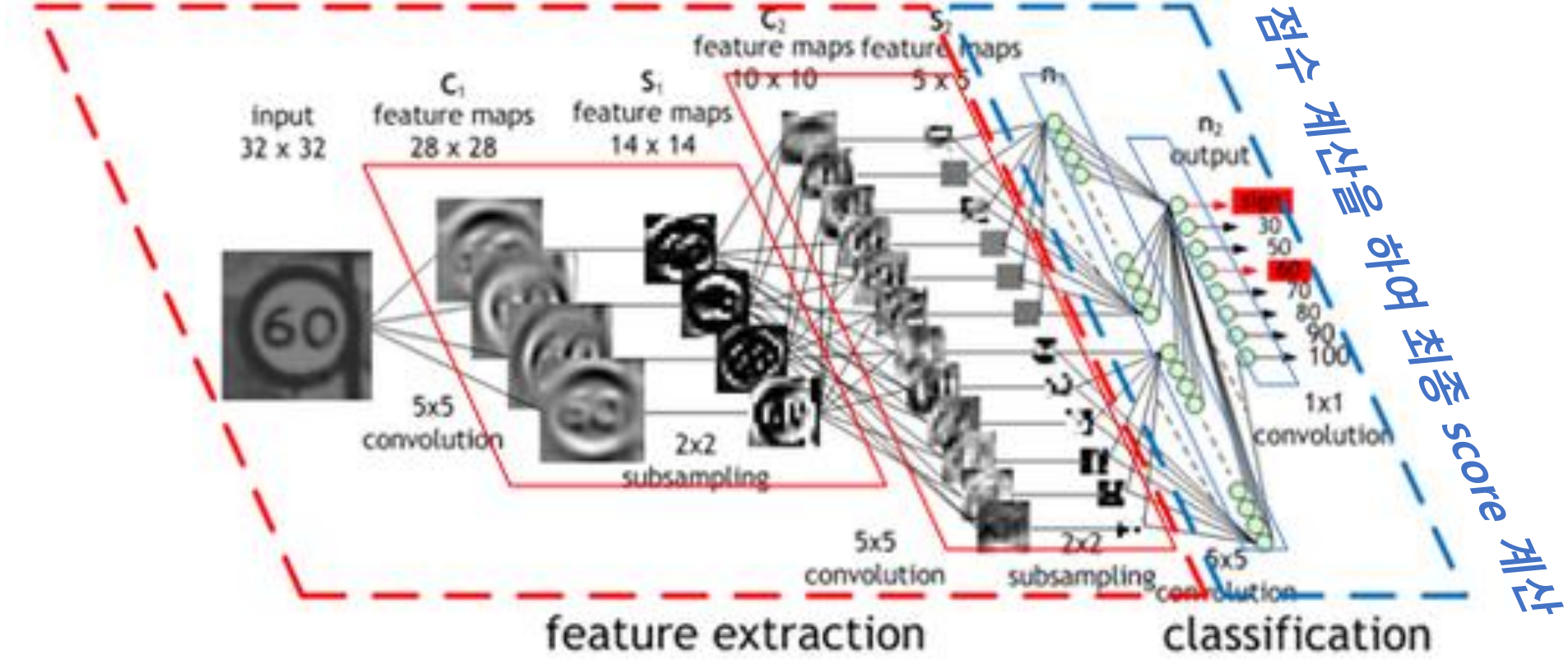
- **Convolution layer와 Sub-sampling layer를 반복하면 결국 feature의 크기가 작아지면서 전체를 대표할 수 있는 강인한 특징들만 남게 된다.**

- **이렇게 얻어진 global한 특징은 fully connected network의 입력으로 연결되어 마지막 클래스별 score계산을 통해 최종 결과를 낸다.**

Structure of CNN

Convolution과 sub-sampling를 반복 하여 점차 global한 특징을 생성

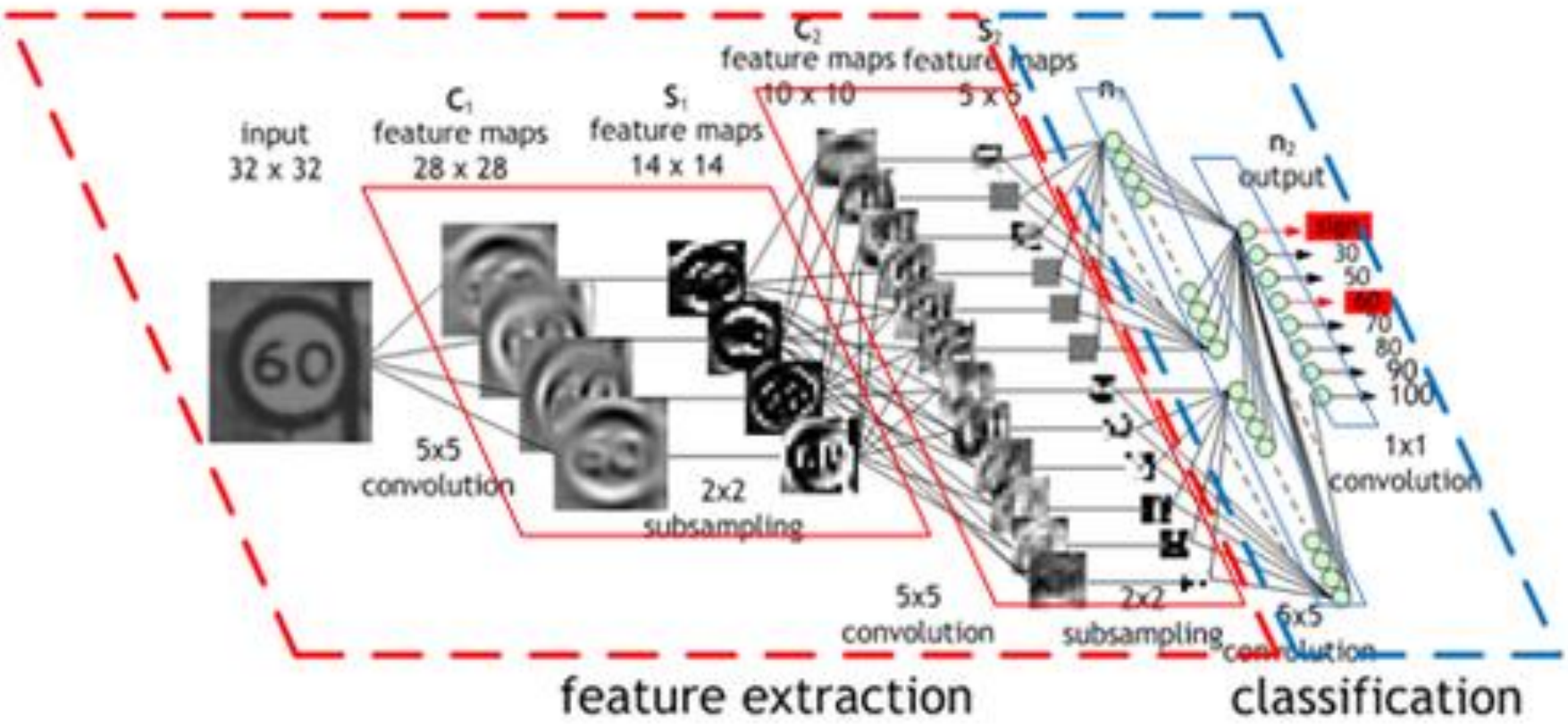
강인한 특징만 남음



점수 계산을 하여 최종 score 계산

Structure of CNN

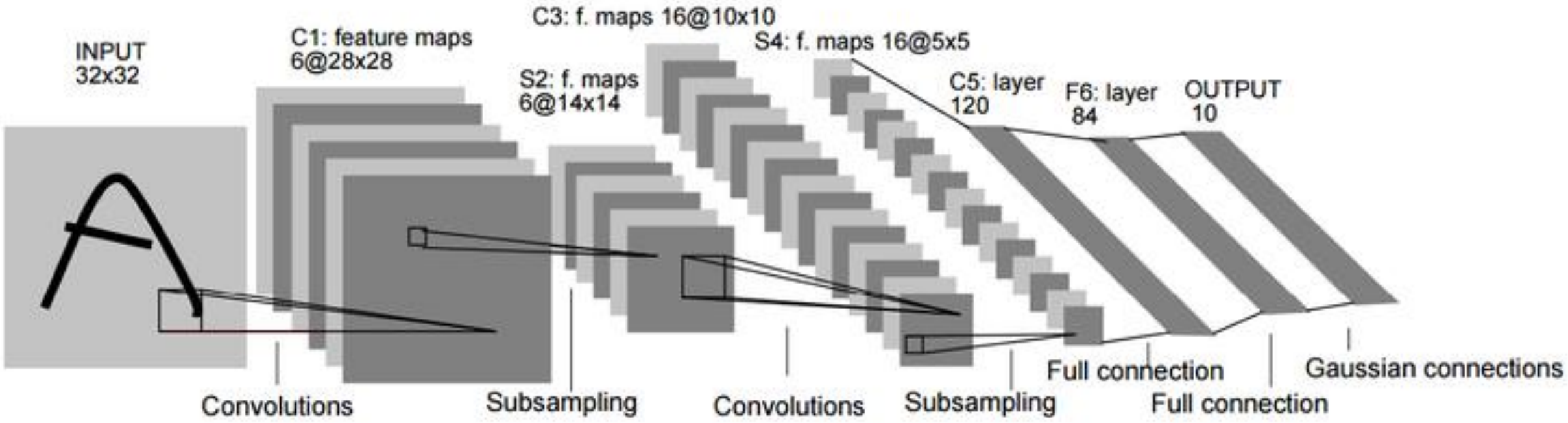
이미지 해상도
Filter 개수
특징 세기



Structure of CNN

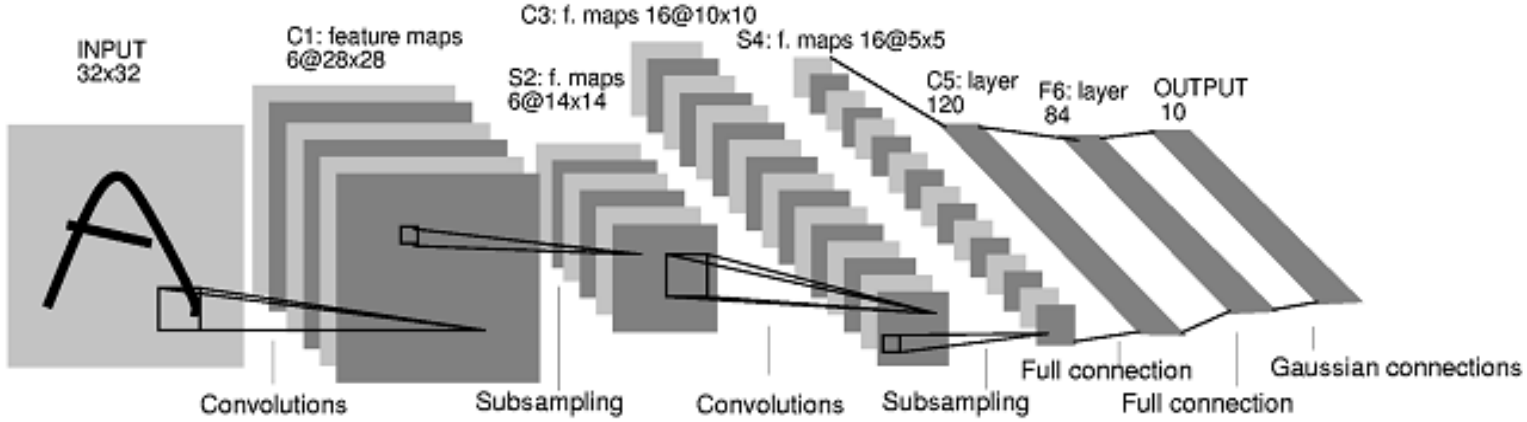
- Lecun – “Gradient-based learning applied to document recognition”

- Lecun은 CNN의 개념을 처음으로 만든 사람, 한동안 정체기에 빠진 신경망 연구의 돌파구 역할을 함



Structure of CNN

문자,숫자 구분을 위한 CNN Architecture



input : 32 x 32 (for NMIST)

C1 : convolution layer (6 kernal size 5 x 5 x 6)

S2 : subsampling layer (6 kernal size 2 x 2, average, not overlaping)

C3 : convolution layer (16 kernal size 5 x 5 x 16)

S4 : subsampling layer (scale size 2 x 2, average, not overlaping)

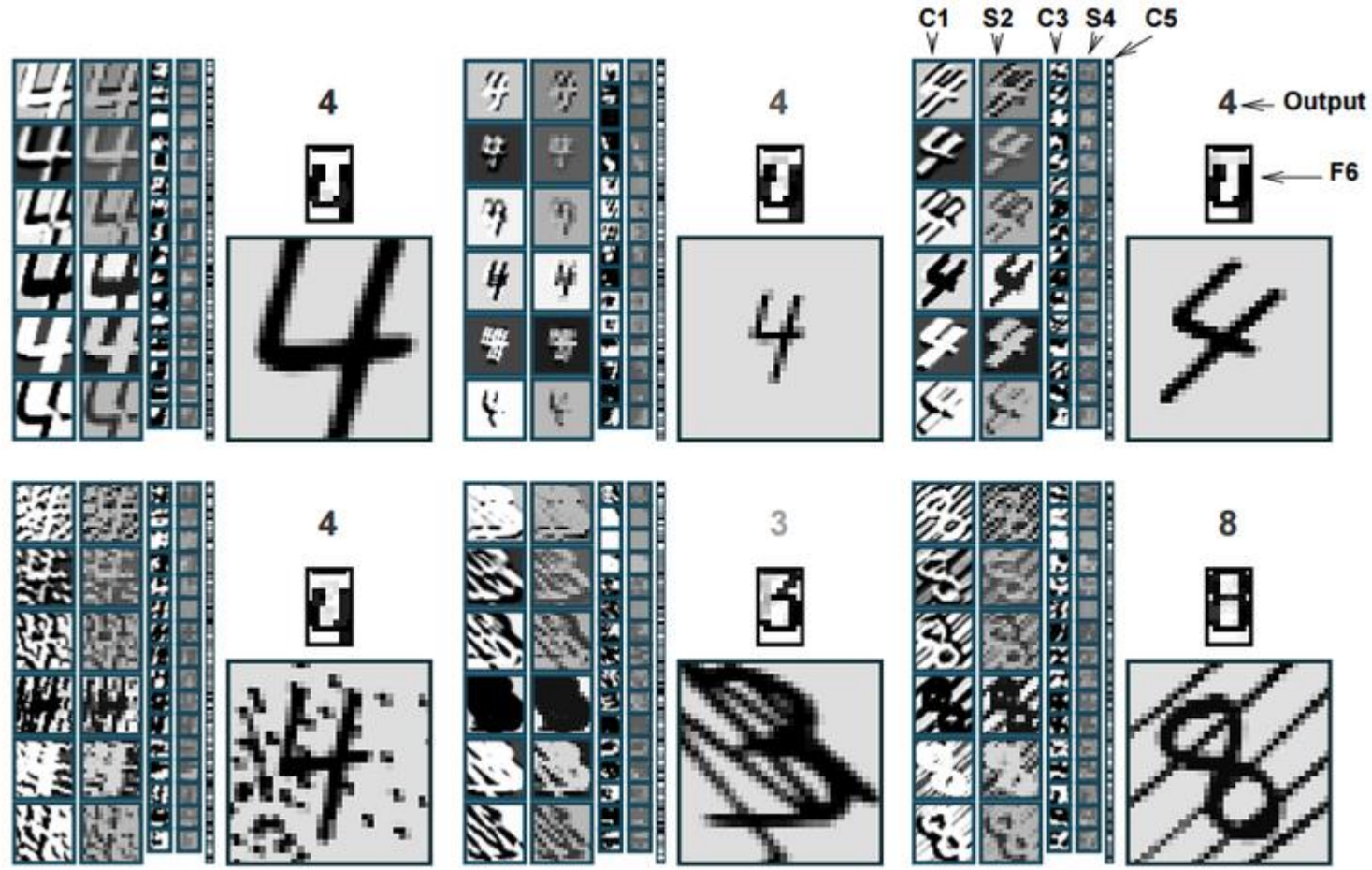
C5 , F6 : fully connected layer (for classification)

activation function : sigmoid (only apply to convolution layer)

Gradient-Based learning Applied to Document Recogntition (Yann LeCun 1990)

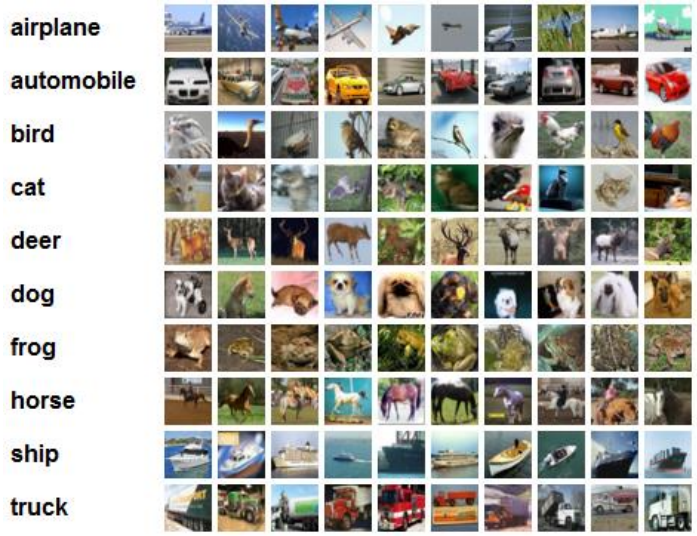
Structure of CNN

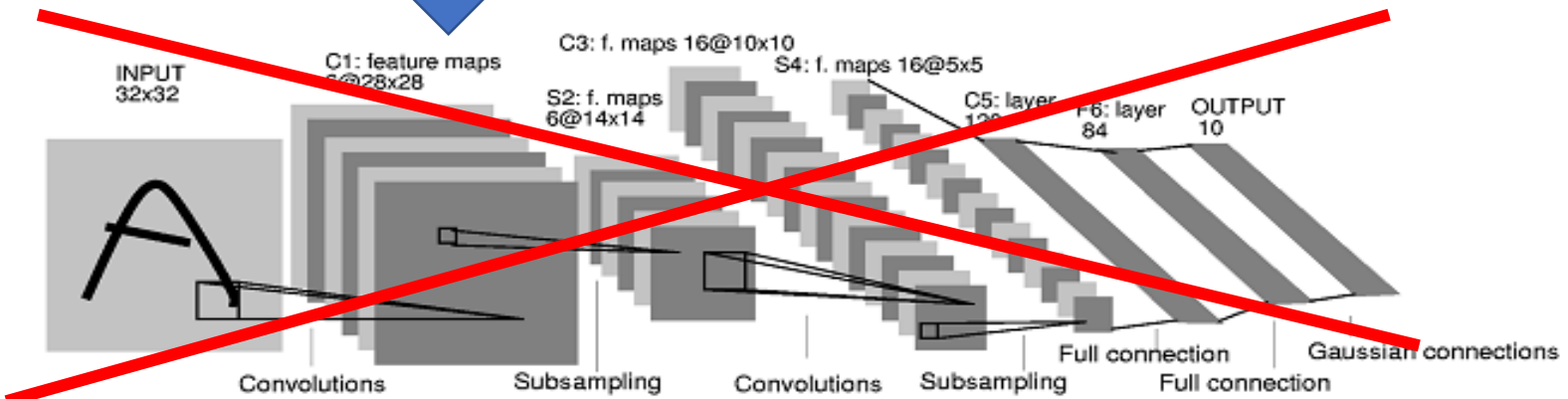
- Lecun – “Gradient-based learning applied to document recognition”
- 여러 feature들을 종합하며, 잡음이 낀 이미지에 대해서도 강인한 성능을 보임



Structure of CNN

- Lecun – “Gradient-based learning applied to document recognition”
- 문제점

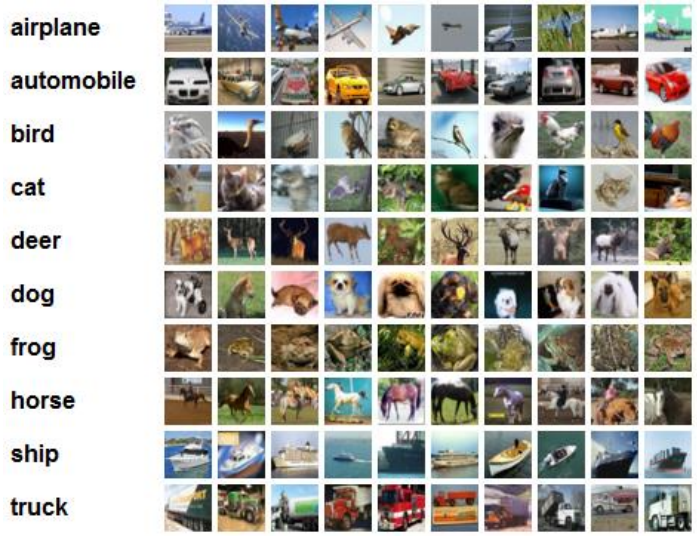


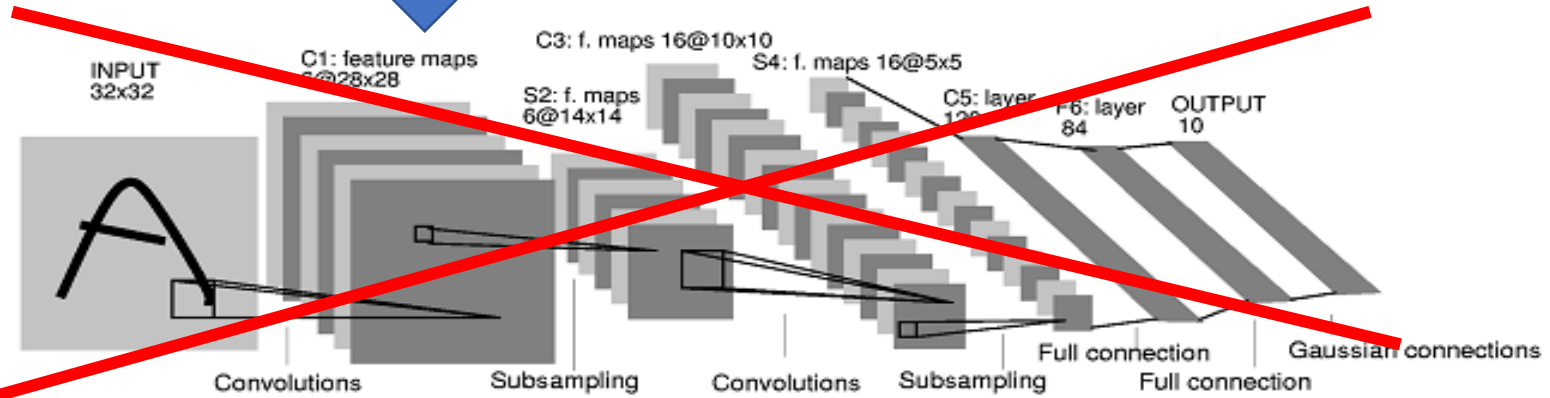


- **기존 Lenet-5의 문제점**
  ⇒ 파라미터 수가 적음
  ⇒ 파라미터 수를 늘릴 경우

Structure of CNN

- Lecun – “Gradient-based learning applied to document recognition”
- 문제점

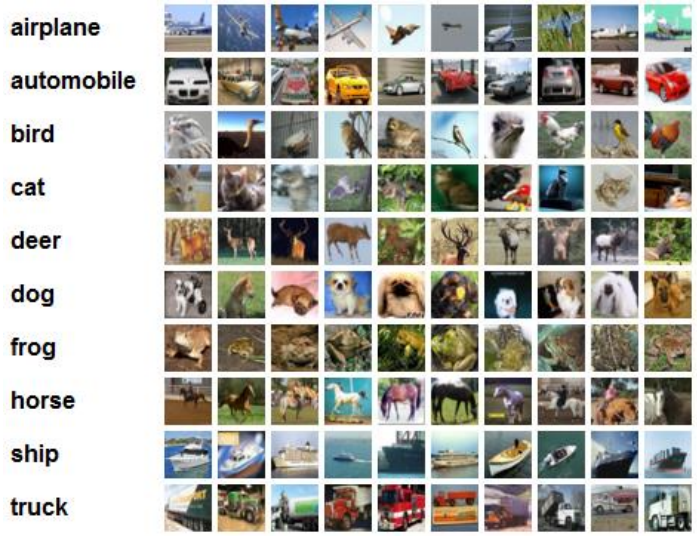


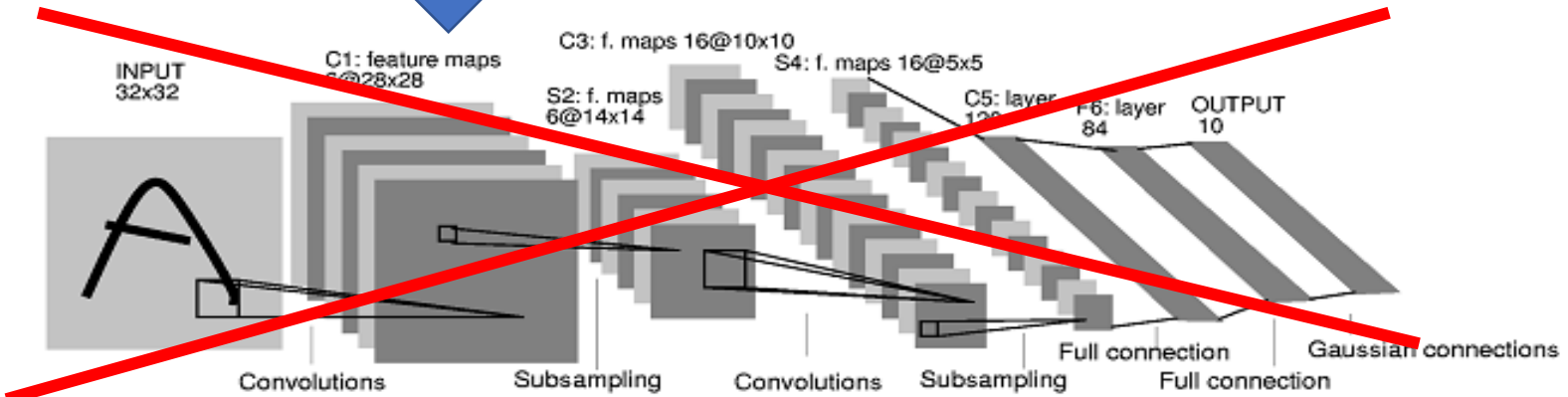


- **파라미터가 적은 문제**
  => 실제 물체의 경우 문자와 달리 복잡성이 훨씬 높게 때문에 기존의 Lenet-5 모델의 적은 수의 파라미터로는 실제 물체 이미지의 정보를 담을 수 없음

Structure of CNN

- Lecun – “Gradient-based learning applied to document recognition”
- 문제점





- **Activation Function 문제**
  => 기존 Sigmoid Function의 경우 네트워크의 층수를 늘리면 Vanishing Gradient 현상이 발생함

Structure of CNN

LeNet (1990, Yann LeCun , just for NMIST)

이후 개선된 네트워크 (이 후 네트워크 부터 이미지 구분 가능)

AlexNet (2012 , Krizhevesky-Hinton , winning in ILSVRC 2012)

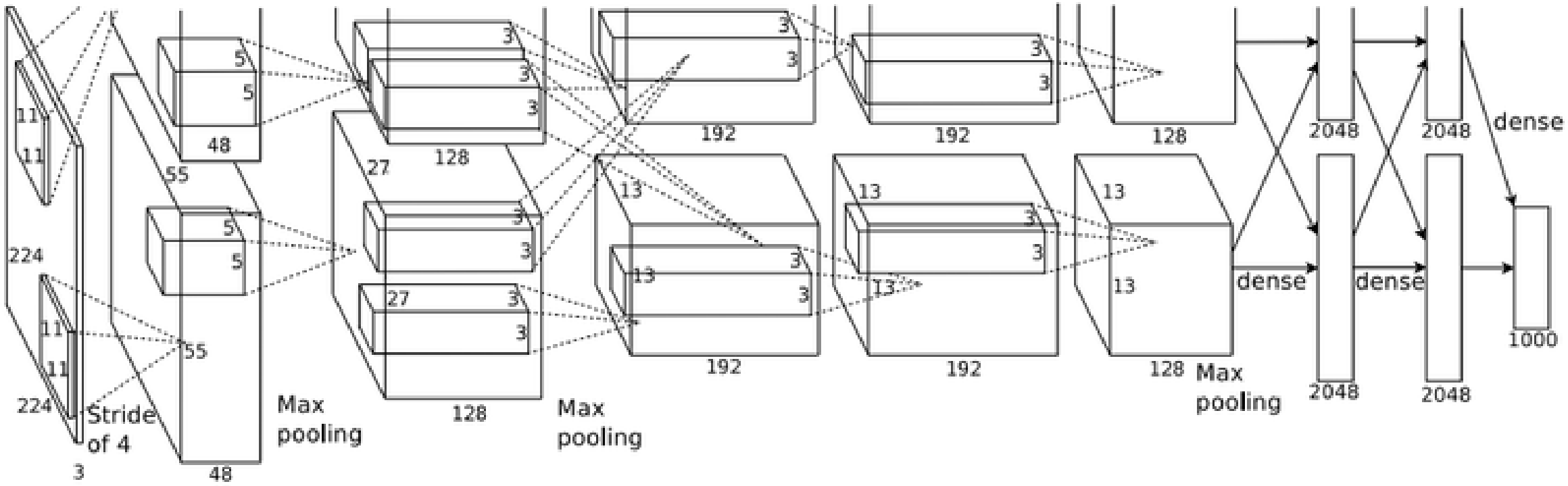
ZF Net (2013, Matthew Zeiler-Rob Fergus , winning in ILSVRC 2013) => modifying AlexNet's hyper-parameter and increasing the size of convolution layer

GoogLeNet (2014, Szegedy in Google, winning in ILSVRC 2014)

ResNet (2015, Kaimming He in microsoft, winning in ILSVRC 2015) => first introducing residual framework

Structure of CNN

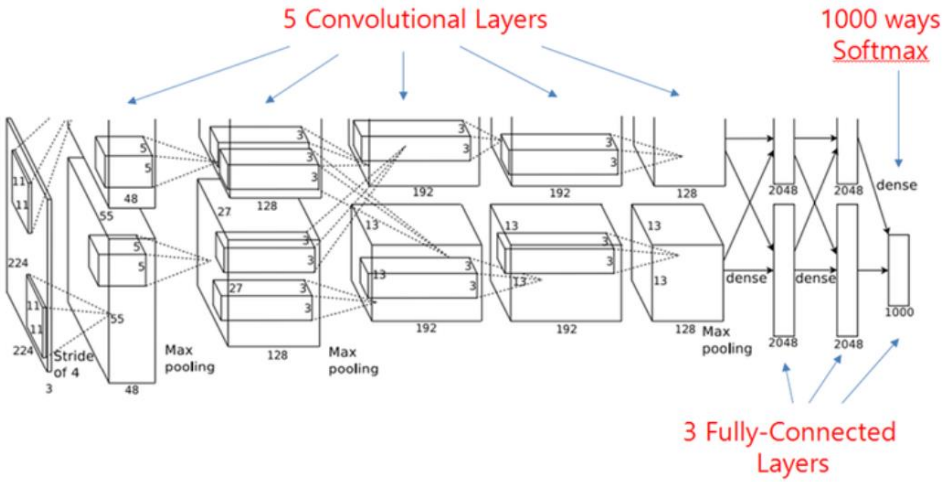
- Krizhevsky – “ImageNet classification with deep convolution neural network”
- 숫자가 아닌 실제 이미지를 구분하는 데 있어서 압도적인 결과를 냄
- 4차혁명의 시초



Structure of CNN

AlexNet (2012 , Krizhevesky-Hinton , winning in ILSVRC 2012)

- **AlexNet 구조**

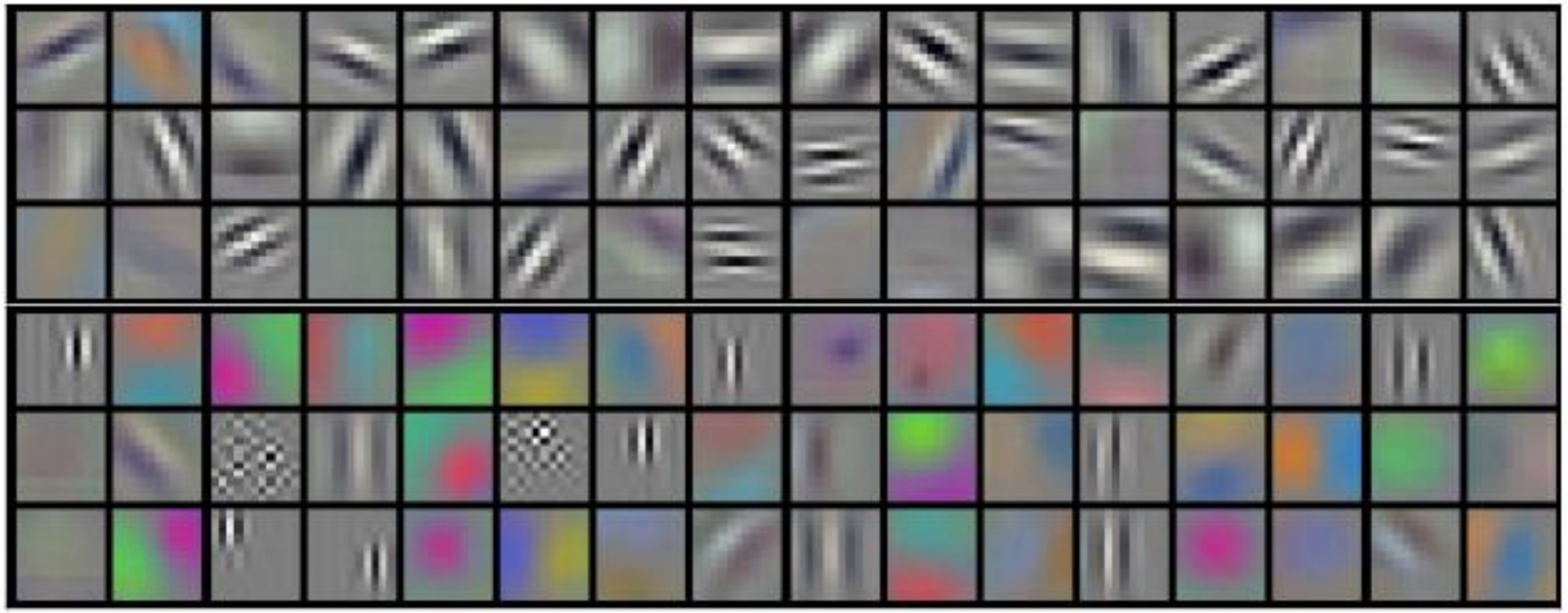


input :224 x 224 x 3 (RGB)
Layer 1 : convolution layer (96 kernels of size 11 x 11 x 3 with a stride of 4 pixel
Layer 2 : subsampling layer (256 kernels of size 5 x 5 x 48 , overlapping max pooling)
(respons-normalized -> max pooling)
Layer 3 : subsampling layer (384 kernels of size 3 x 3 x 256 ,overlapping max pooling)
(respons-normalized -> max pooling)
Layer 4 : convolution layer (384 kernels of size 3 x 3 x 192)
Layer 5 : convolution layer (256 kernels of size 3 x 3 x 192)
Layer 6 , 7 :fully connected layer

Activation function : Rectified Linear Units is applied to all layers.
Drop out is applied to all fully connected layers

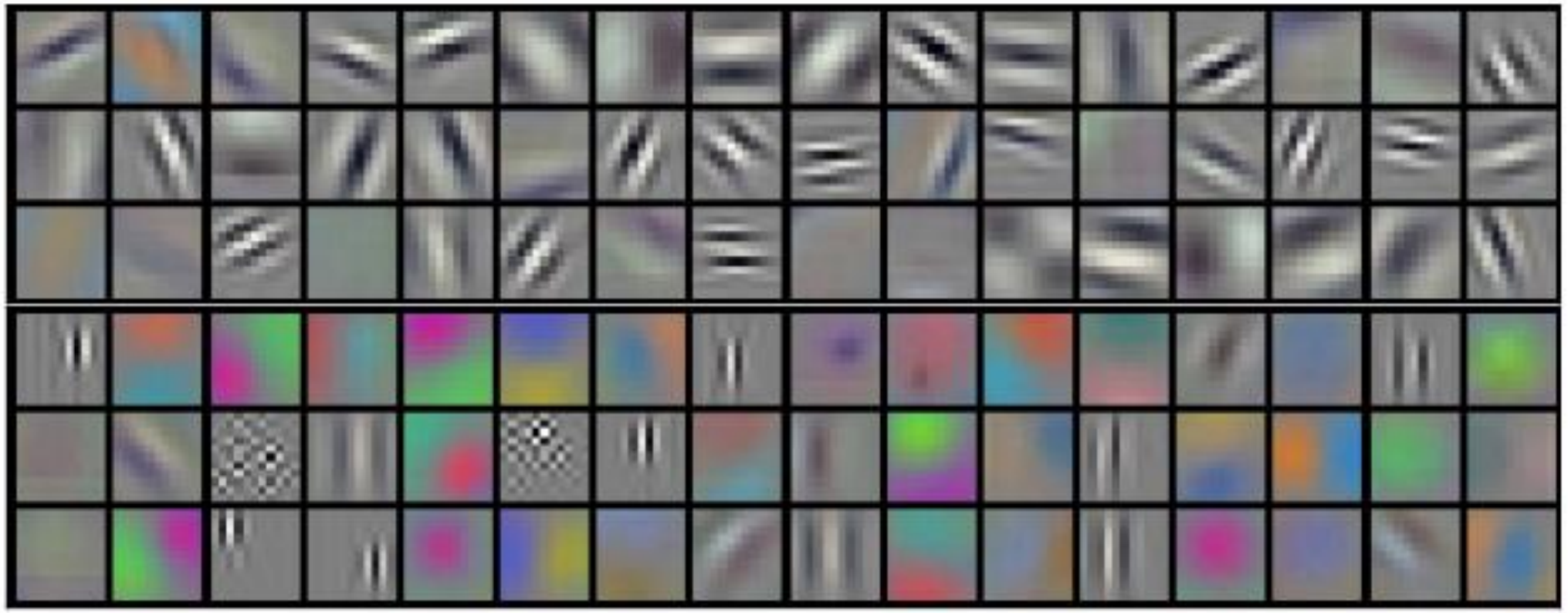
Structure of CNN

- Krizhevsky – “ImageNet classification with deep convolution neural network”
- 이런 학습된 kernel을 갖는 여러 단계의 convolution 중첩을 통해 영상을 1000개의 class로 분류할 수 있는 특징을 추출할 수 있게 되었다.



Structure of CNN

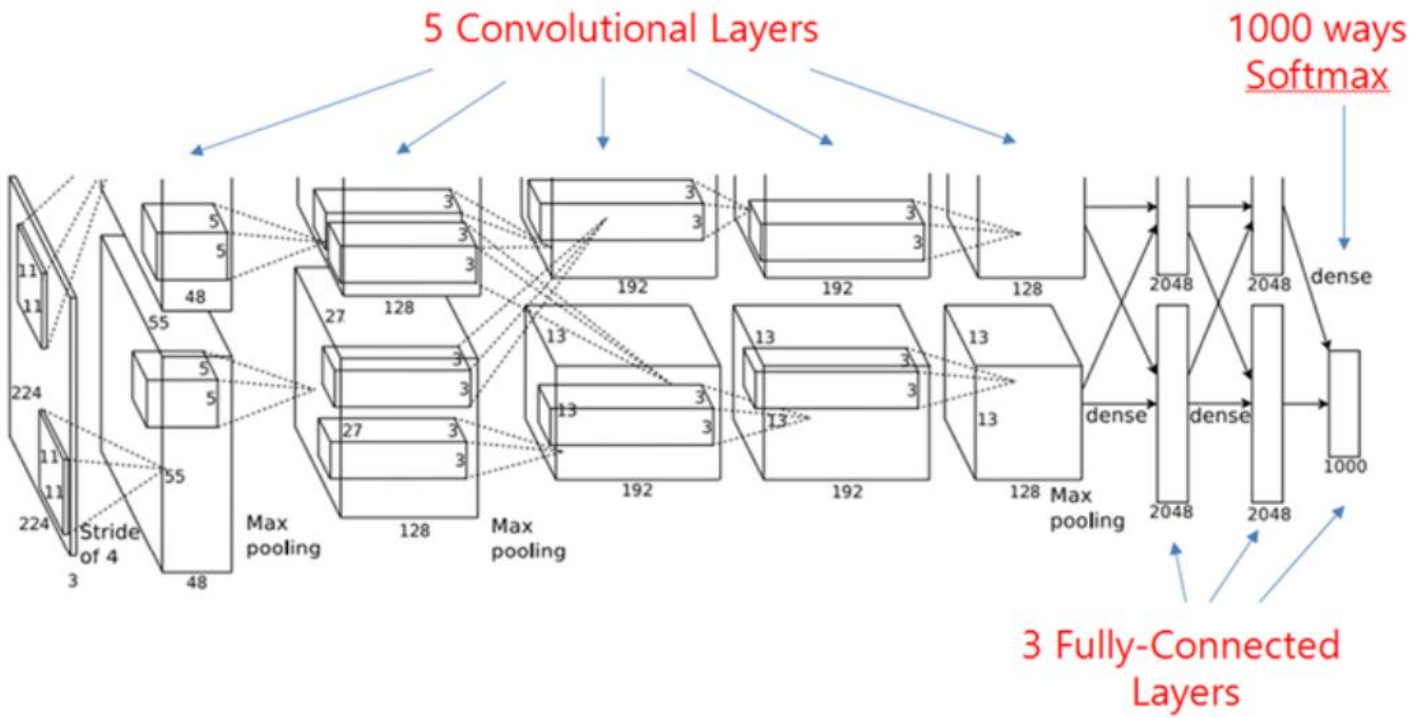
- Krizhevsky – “ImageNet classification with deep convolution neural network”
- 일반 영상처리의 convolution의 filter는 값이 고정되어 있지만
- CNN은 학습을 통해 계수 값을 스스로 정하는 것



Structure of CNN

AlexNet (2012 , Krizhevesky-Hinton , winning in ILSVRC 2012)

- 마지막 Classification 층에 **softmax**를 사용
- 숫자와 달리 이미지는 RGB 3개의 채널을 가짐
- Activation Function : Sigmoid => **Relu (Vanishing 문제 해결)**
- Overlapped pooling 을 새로 도입
- **Over-fitting** 문제 해결



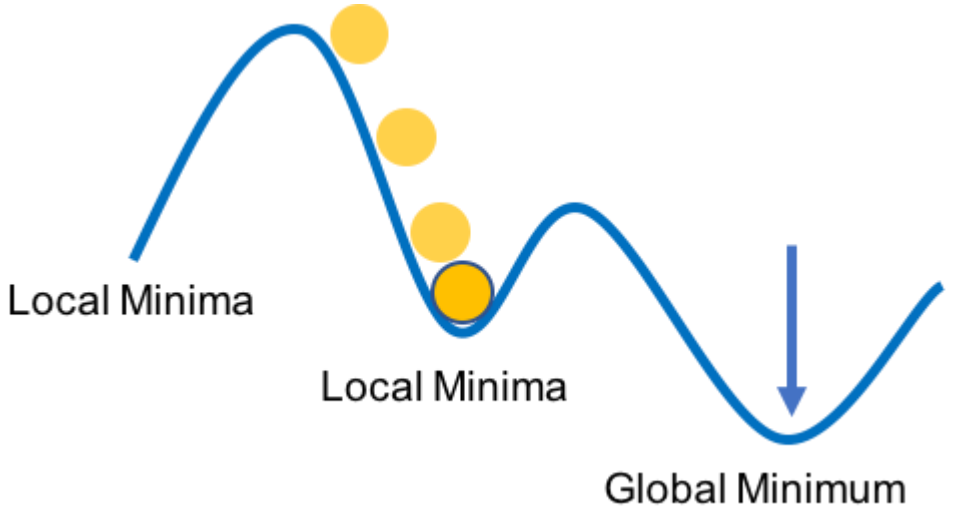
Structure of CNN

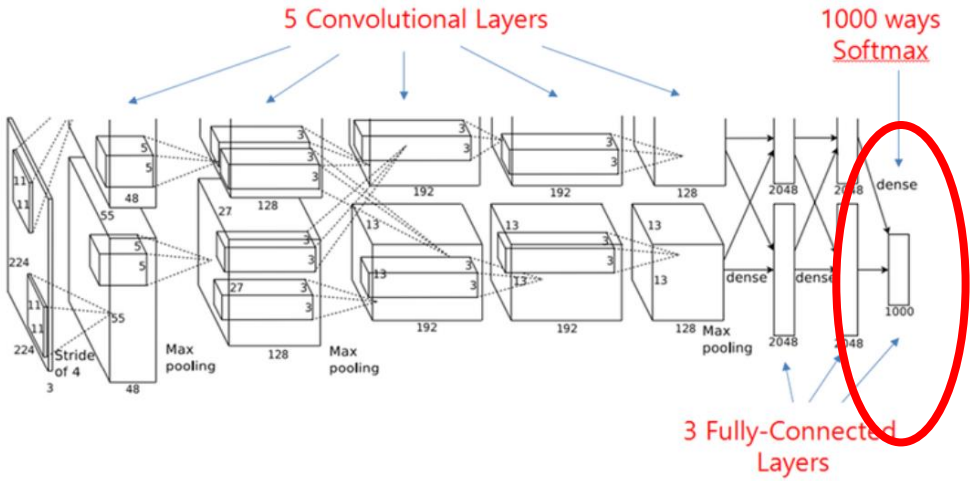
AlexNet (2012 , Krizhevesky-Hinton , winning in ILSVRC 2012)

- **마지막 Classification 층에 softmax를 사용**
  => Loss 함수에서 Mean Square Error를 쓰면 Global Minima로 가기 힘듬(최적화가 힘듬) => Cross Entropy 식을 사용

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

$$H(p,q) = -\sum_x p(x)\,\log q(x)$$



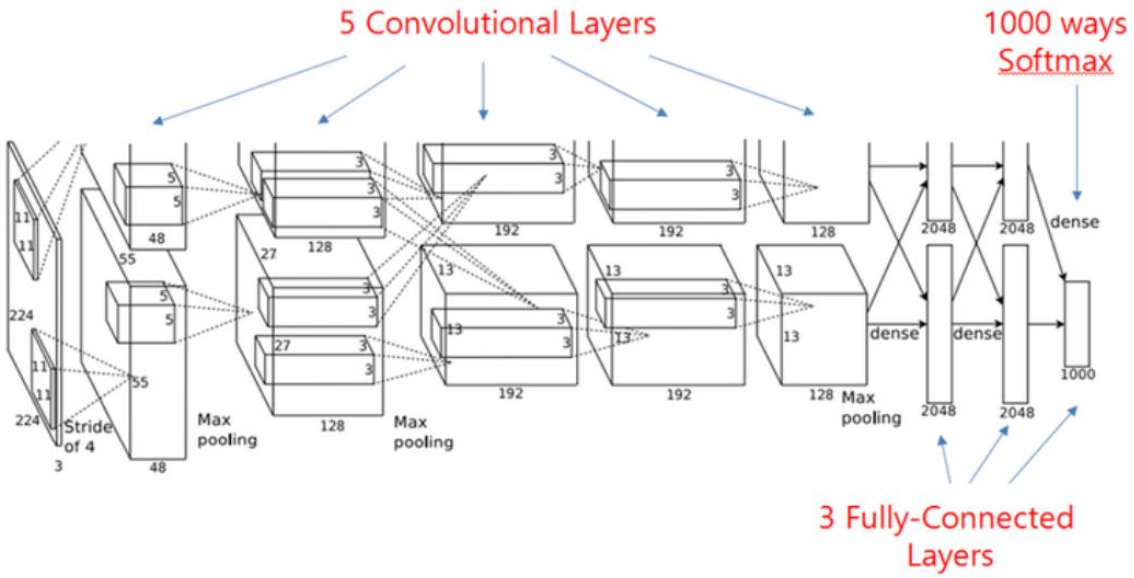


Structure of CNN

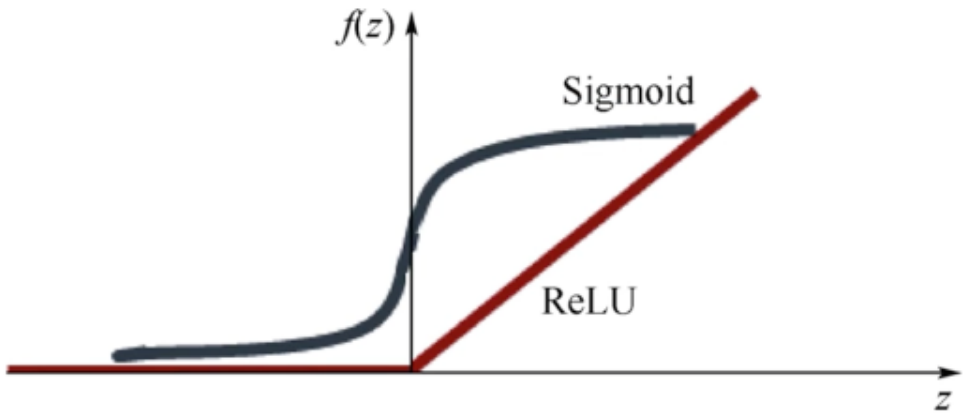
AlexNet (2012 , Krizhevesky-Hinton , winning in ILSVRC 2012)

- **Activation Function : Sigmoid => Relu (Vanishing 문제 해결)**

- **Relu**



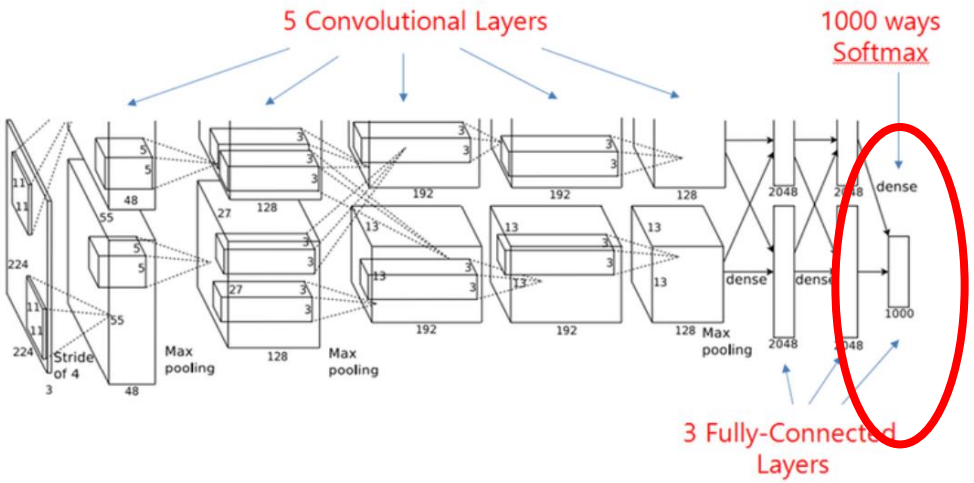
~~Sigmoid!~~

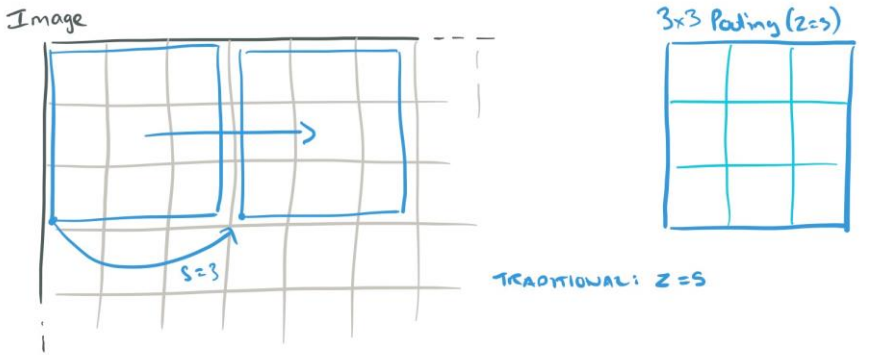


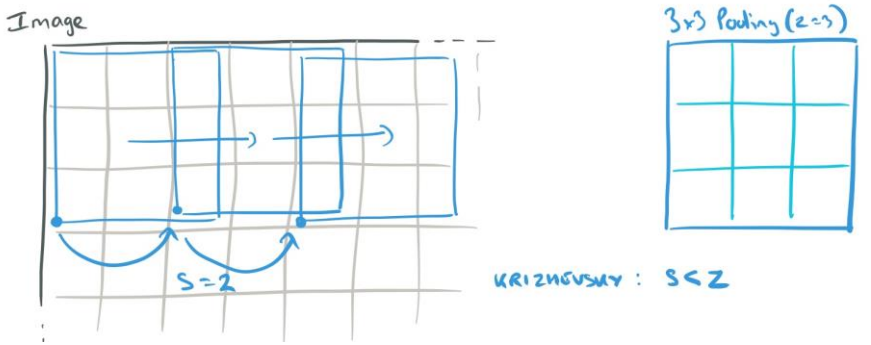
Structure of CNN

AlexNet (2012 , Krizhevesky-Hinton , winning in ILSVRC 2012)

- **Overlapped pooling 을 새로 도입(최근에 쓰지 않음)**



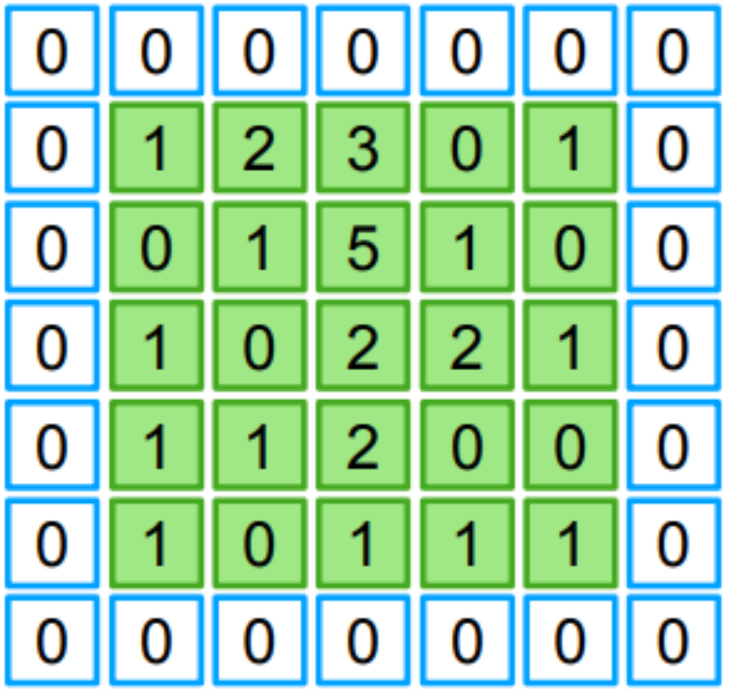




Structure of CNN

Stride and Padding

- stride : filter를 한번에 얼마나 이동 할 것인가
- padding : zero-padding



input + padding

Code of CNN

Code of CNN

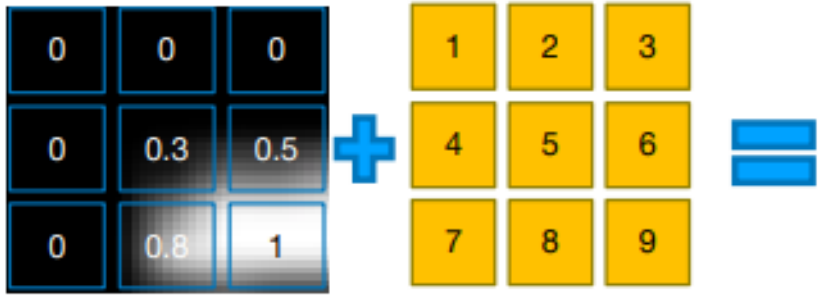
Pytorch nn.Conv2d

CLASS `torch.nn.Conv2d(`*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True*`)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$



???

ex) 입력 채널 1 /출력채널 1 / 커널 크기3x3

conv = nn.Conv2d(1,1,3)

Code of CNN

입력의 형태

- input type : torch.Tensor
- input shape : (N x C x H x W)

  (batch_size, channel, height, width)

Code of CNN

Convolution의 output 크기

$$Output\ size = \frac{input\ size - filter\ size + (2 * padding)}{Stride} + 1$$

예제 1)
input image size : 227 x 227
filter size = 11x11
stride = 4
padding = 0
output image size = ?

예제 2)
input image size : 64 x 64
filter size = 7x7
stride = 2
padding = 0
output image size = ?

예제 3)
input image size : 32 x 32
filter size = 5x5
stride = 1
padding = 2
output image size = ?

예제 4)
input image size : 32 x 64
filter size = 5x5
stride = 1
padding = 0
output image size = ?

예제 5)
input image size : 64 x 32
filter size = 3x3
stride = 1
padding = 1
output image size = ?

Code of CNN

Neuron과 Convolution

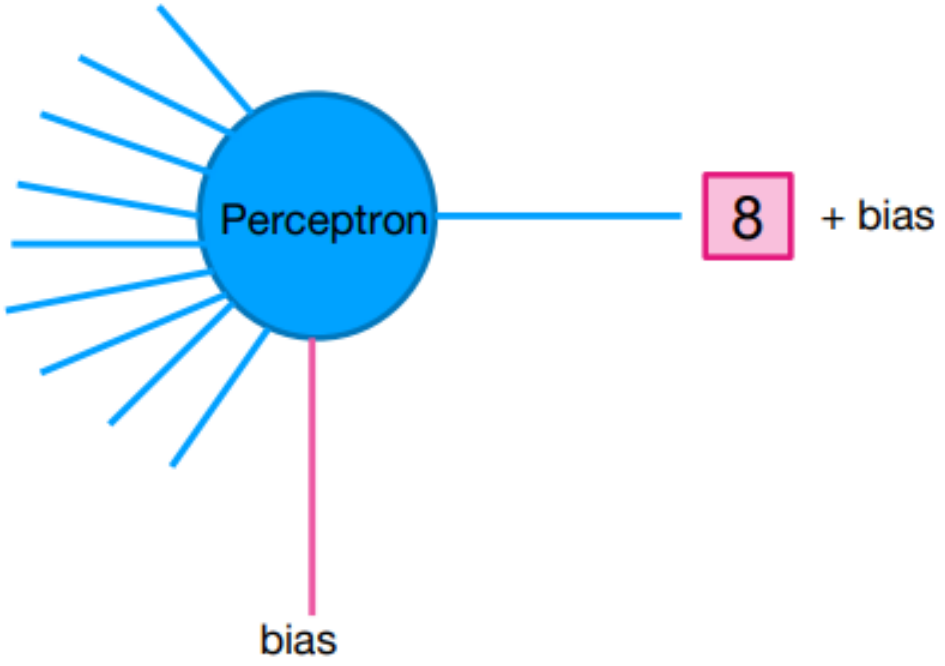
- Perceptron과 Convolution

Convolution filter(=kernel)

1	0	1
0	1	0
1	0	1

1	2	3
0	1	5
1	0	2

1	2	3	0	1
0	1	5	1	0
1	0	2	2	1
1	1	2	0	0
1	0	1	1	1



Code of CNN

직접 계산

- nn.Conv2d에 입력
- filter size 변경 (size = 1x1, 3x3, 5x5)
- bias
- stride
- padding

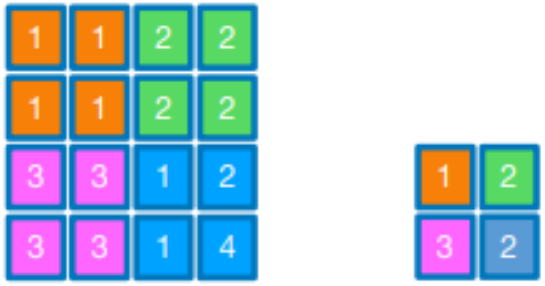
Code of CNN

Pooling

- Max Pooling

max pooling

- Average Pooling



Average pooling

Code of CNN

MaxPool2d

MaxPool2d

CLASS `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)` [SOURCE]

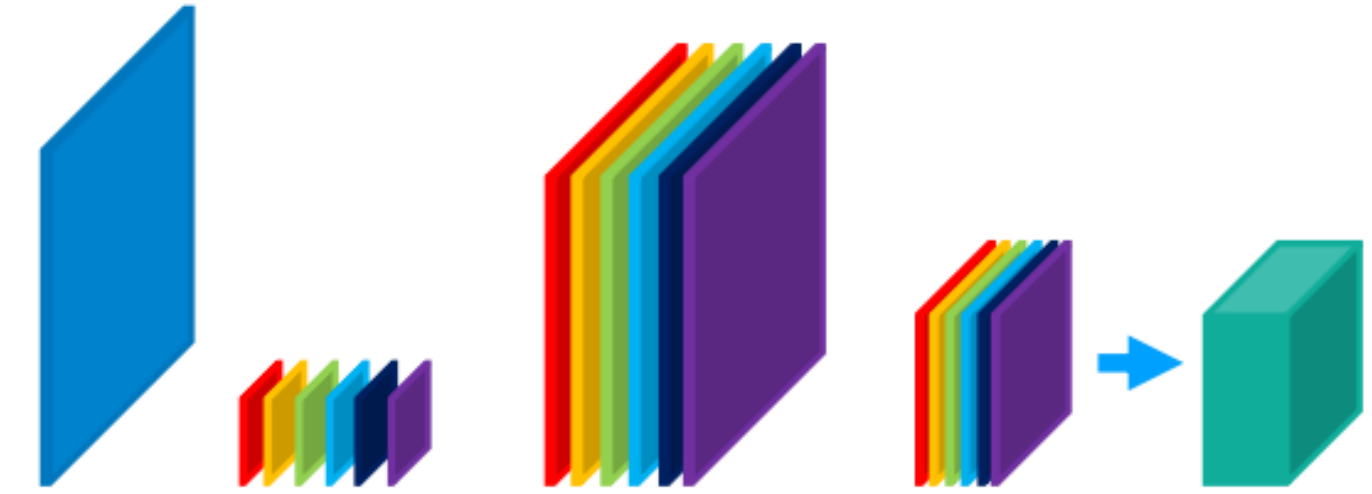
Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

Code of CNN

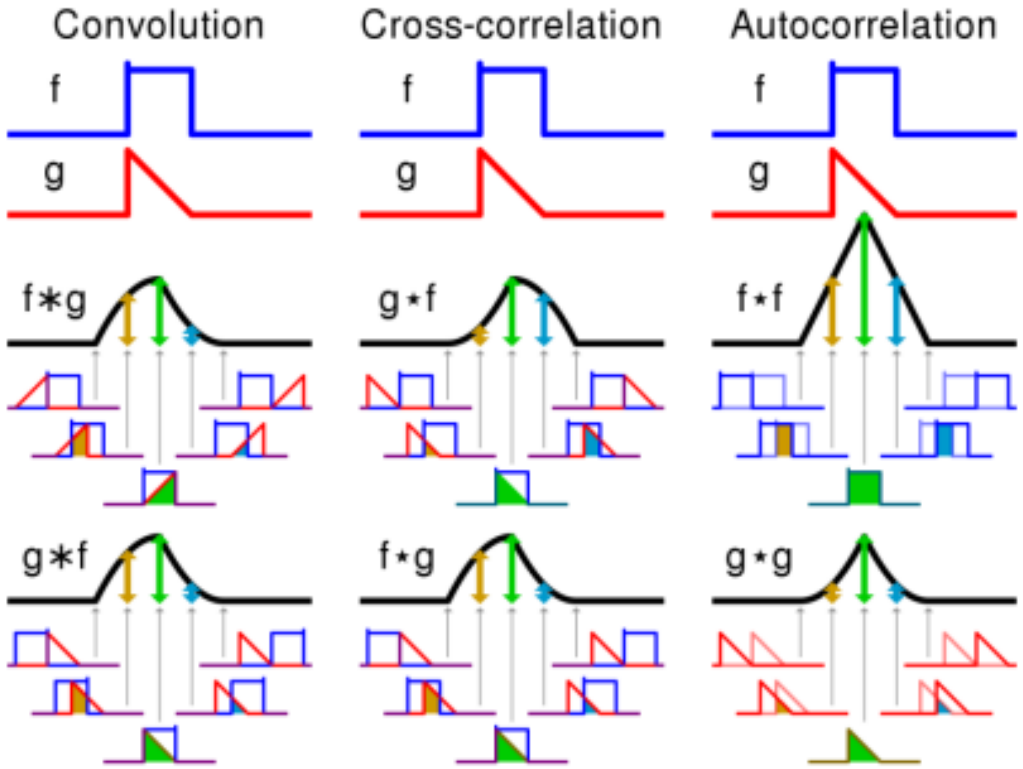
CNN implementation



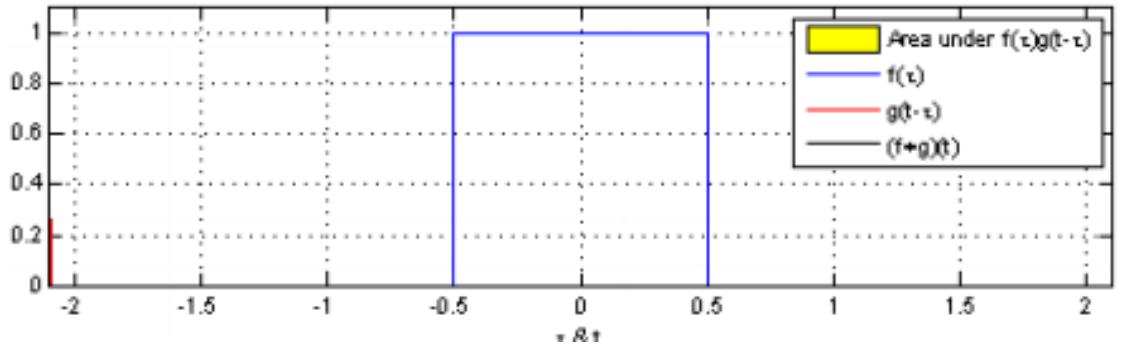

```
input = torch.Tensor(1,1,28,28)
conv1=nn.Conv2d(1,5,5)
pool = nn.MaxPool2d(2)
out = conv1(input)
out2 =pool(out)
out.size()
out2.size()
```

Code of CNN

What is Convolution?



$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t - \tau) g(\tau)\, d\tau.$$



Code of CNN

학습 단계(code 기준)

1. 라이브러리 가져오고 (torch, torchvision, matplotlib 같은것들)
2. GPU 사용 설정 하고 random value를 위한 seed 설정!
3. 학습에 사용되는 parameter 설정!(learning_rate, training_epochs, batch_size, etc)
4. 데이터셋을 가져오고 (학습에 쓰기 편하게) loader 만들기
5. 학습 모델 만들기(class CNN(torch.nn.Module))
6. Loss function (Criterion)을 선택하고 최적화 도구 선택(optimizer)
7. 모델 학습 및 loss check(Criterion의 output)
8. 학습된 모델의 성능을 확인한다.

Code of CNN

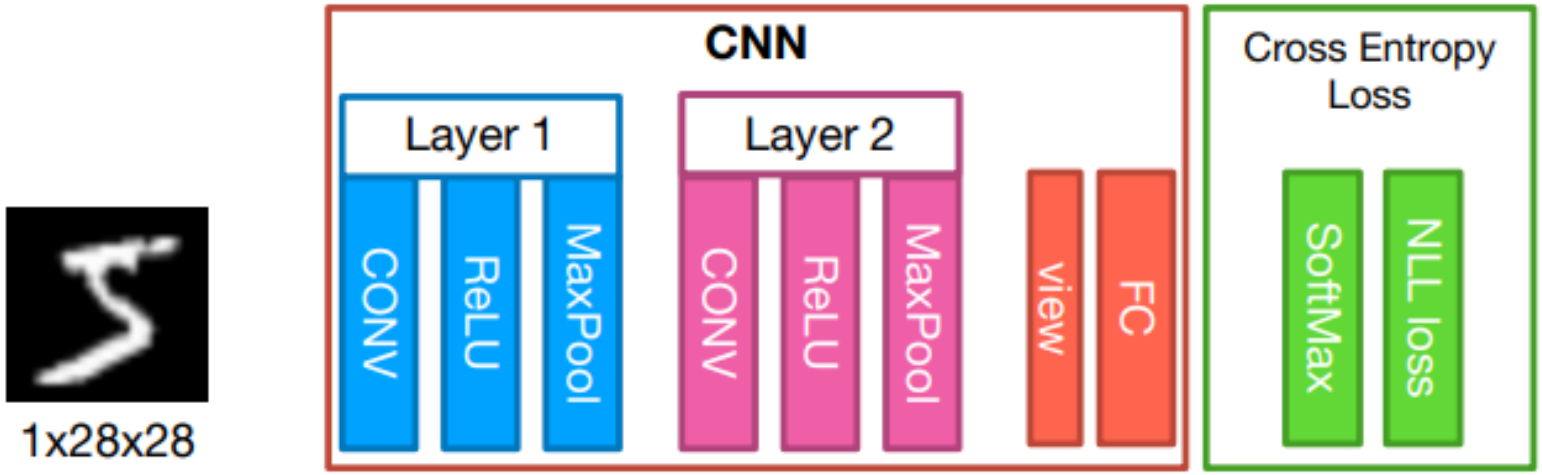
```python
# Lab 11 MNIST and Convolutional Neural Network
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torch.nn.init

# device = 'cuda' if torch.cuda.is_available() else 'cpu'
device = 'cpu'
# for reproducibility
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)

# parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100

# MNIST dataset
mnist_train = dsets.MNIST(root='MNIST_data/',
                          train=True,
                          transform=transforms.ToTensor(),
                          download=True)

mnist_test = dsets.MNIST(root='MNIST_data/',
                         train=False,
                         transform=transforms.ToTensor(),
                         download=True)

# dataset loader
data_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True)

# CNN Model (2 conv layers)
class CNN(torch.nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        # L1 ImgIn shape=(?, 28, 28, 1)
        #    Conv     -> (?, 28, 28, 32)
        #    Pool     -> (?, 14, 14, 32)
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))
        # L2 ImgIn shape=(?, 14, 14, 32)
        #    Conv      ->(?, 14, 14, 64)
        #    Pool      ->(?, 7, 7, 64)
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))
        # Final FC 7x7x64 inputs -> 10 outputs
        self.fc = torch.nn.Linear(7 * 7 * 64, 10, bias=True)
        torch.nn.init.xavier_uniform_(self.fc.weight)

```

Code of CNN

```
    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)   # Flatten them for FC
        out = self.fc(out)
        return out


# instantiate CNN model
model = CNN().to(device)

# define cost/loss & optimizer
criterion = torch.nn.CrossEntropyLoss().to(device)    # Softmax is internally computed.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# train my model
total_batch = len(data_loader)
print('Learning started. It takes sometime.')
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:
        # image is already size of (28x28), no reshape
        # label is not one-hot encoded
        X = X.to(device)
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = model(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('[Epoch: {:>4}] cost = {:>.9}'.format(epoch + 1, avg_cost))

print('Learning Finished!')

# Test model and check accuracy
with torch.no_grad():
    X_test = mnist_test.test_data.view(len(mnist_test), 1, 28, 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = model(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print('Accuracy:', accuracy.item())
```

Code of CNN

우리가 만들 CNN 구조



(Layer 1) Convolution layer = (in_c=1, out_c=32,kernel_size =3, stride=1,padding=1)
(Layer 1) MaxPool layer = (kernel_size=2, stride =2)

(Layer 2) Convolution layer = (in_c=32, out_c=64, kernel_size =3, stride=1,padding=1)
(Layer 2) MaxPool layer = (kernel_size=2, stride =2)

view => (batch_size x [7,7,64] => batch_size x [3136])
Fully_Connect layer => (input=3136, output = 10)

Code of CNN

우리가 확인한 결과

- 더 깊게 레이어를 쌓으면 어떻게 될까?

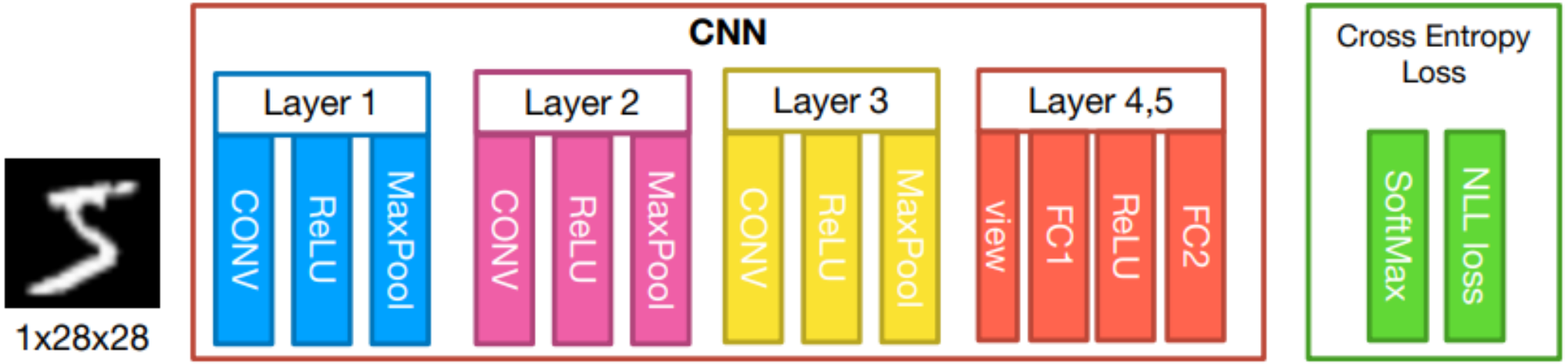
- 더 잘 되지 않을까?

  - Vanishing/Exploding Gradient 문제: CNN에서 파라미터 update를 할 때, gradient값이 너무 큰 값이나 작은 값으로 포화되어 더 이상 움직이지 않아 학습의 효과가 없어지거나 학습 속도가 아주 느려지는 문제가 있다. 망이 깊어질수록 이 문제는 점점 더 심각해지며, 이 문제를 피하기 위해 batch normalization, 파라미터의 초기값 설정 방법 개선 등의 기법들이 적용되고 있지만, layer 개수가 일정 수를 넘어가게 되면 여전히 골치거리가 된다.

  - 더 어려워지는 학습 방법: 망이 깊어지게 되면, 파라미터의 수가 비례적으로 늘어나게 되어 overfitting의 문제가 아닐지라도 오히려 에러가 커지는 상황이 발생한다.

Code of CNN

오늘 만들 CNN 구조 확인!



(Layer 1) Convolution layer = (in_c=1, out_c=32,kernel_size =3, stride=1,padding=1)
(Layer 1) MaxPool layer = (kernel_size=2, stride =2)

(Layer 2) Convolution layer = (in_c=32, out_c=64,kernel_size =3, stride=1,padding=1)
(Layer 2) MaxPool layer = (kernel_size=2, stride =2)

(Layer 3) Convolution layer = (in_c=64, out_c=128,kernel_size =3, stride=1,padding=1)
(Layer 3) MaxPool layer = (kernel_size=2, stride =2)

(Layer 4) Fully Connected layer =(input=4*4*128, output = 625)
(Layer 5) Fully Connected layer =(input=625, output = 10)

Code of CNN

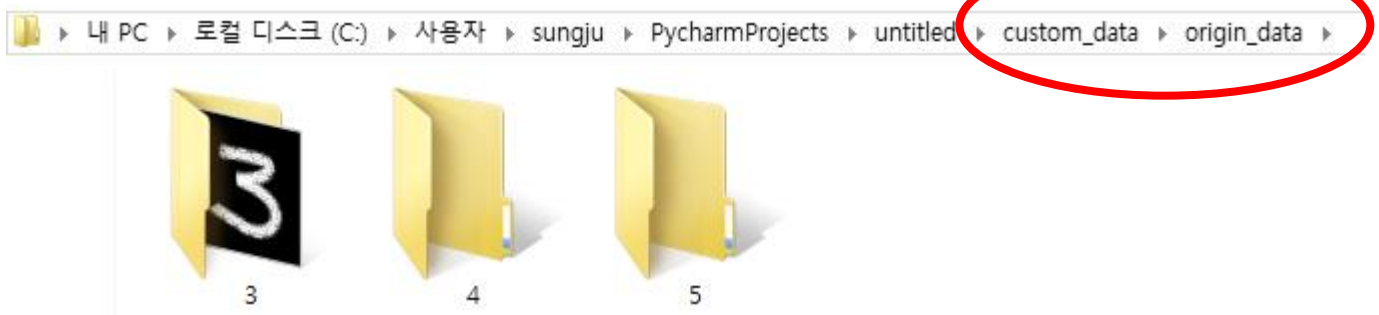
나만의 데이터 만들기

```python
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self):
        self.x_data = [[73, 80, 75],
                       [93, 88, 93],
                       [89, 91, 90],
                       [96, 98, 100],
                       [73, 66, 70]]
        self.y_data = [[152], [185], [180], [196], [142]]

    def __len__(self):
        return len(self.x_data)

    def __getitem__(self, idx):
        x = torch.FloatTensor(self.x_data[idx])
        y = torch.FloatTensor(self.y_data[idx])

        return x, y

dataset = CustomDataset()
```

대체 가능!

```python
trans=torchvision.transforms.Compose([
    transforms.Resize((64,128)),
    transforms.ToTensor()
])
test_data = torchvision.datasets.ImageFolder(root='./custom_data/test_data', transform=trans)
```

Input 사이즈



Root 경로 안에 클래스별로 폴더안에 담아두면 저절로 만들어짐

Code of CNN

나만의 데이터 만들기

```
import torch
import torch.nn as nn
import torch.nn.functional as F

import torch.optim as optim
from torch.utils.data import DataLoader

import torchvision
import torchvision.transforms as transforms
```

```
trans=torchvision.transforms.Compose([
    transforms.Resize((64,128)),
    transforms.ToTensor()
])
test_data = torchvision.datasets.ImageFolder(root='./custom_data/test_data', transform=trans)
```

```
test_set = DataLoader(dataset = test_data, batch_size = len(test_data))
```

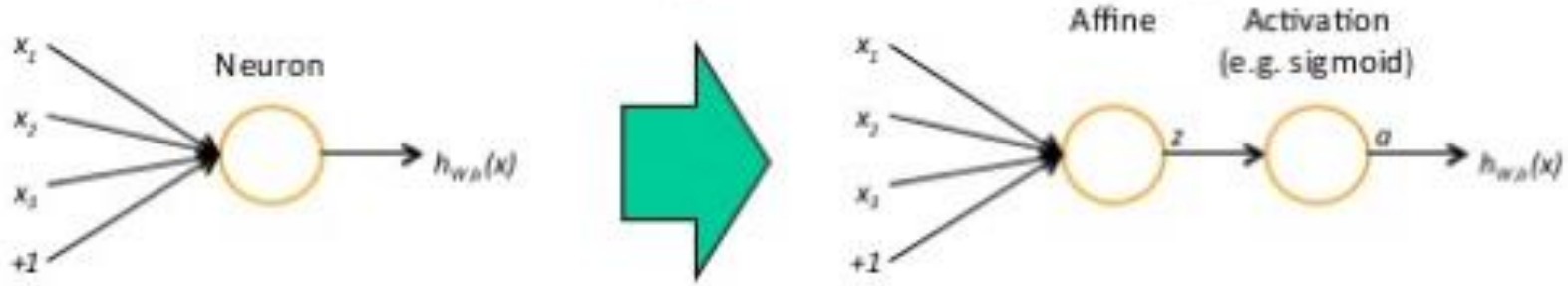
```
with torch.no_grad():
    for num, data in enumerate(test_set):
        imgs, label = data
        imgs = imgs.to(device)
        label = label.to(device)

        prediction = net(imgs)

        correct_prediction = torch.argmax(prediction, 1) == label

        accuracy = correct_prediction.float().mean()
        print('Accuracy:', accuracy.item())
```

Neural Network as a Composite Function

A neural network is decomposed into a composite function where each function element corresponds to a differentiable operation.

- Single neuron (the simplest neural network) example

  A single neuron is decomposed into a composite function of an affine function element parameterized by W and b and an activation function element f which we choose to be the sigmoid function.



$$h_{W,b}(x) = f\left(W^T x + b\right) = \text{sigmoid}\left(\text{affine}_{W,b}(x)\right) = \left(\text{sigmoid} \circ \text{affine}_{W,b}\right)(x)$$

Derivatives of both affine and sigmoid function elements w.r.t. both inputs and parameters are known.
Note that sigmoid function does not have neither parameters nor derivatives parameters.

Sigmoid function is applied element-wise. '•' denotes Hadamard product, or element-wise product.

$$\frac{\partial a}{\partial z} = a \bullet (1-a) \text{ where } a = h_{w,b}(x) = \text{sigmoid}(z) = \frac{1}{1+\exp(-z)}$$

$$\frac{\partial z}{\partial x} = W,\ \frac{\partial z}{\partial W} = x,\ \frac{\partial z}{\partial b} = I \text{ where } z = \text{affine}_{W,b}(x) = W^T x + b, \text{ and } I \text{ is identity matrix}$$

Chain Rule of Error Signals and Gradients

Error signals are defined as the derivatives of any cost function *J* which we choose to be the square error. Error signals are computed (propagated backward) by the chain rule of derivative and useful for computing the gradient of the cost function.

- Single neuron example

  Suppose we have *m* labeled training examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$. Square error cost function for each example is as follows. Overall cost function is the summation of cost functions over all examples.

$$J(W,b;x,y)=\frac{1}{2}\left\|y-h_{w,b}(x)\right\|^2$$

  Error signals of the square error cost function for each example are propagated using derivatives of function elements w.r.t. inputs.

$$\delta^{(a)}=\frac{\partial}{\partial a}J(W,b;x,y)=-(y-a)$$

$$\delta^{(z)}=\frac{\partial}{\partial z}J(W,b;x,y)=\frac{\partial J}{\partial a}\frac{\partial a}{\partial z}=\delta^{(a)}\bullet a\bullet(1-a)$$

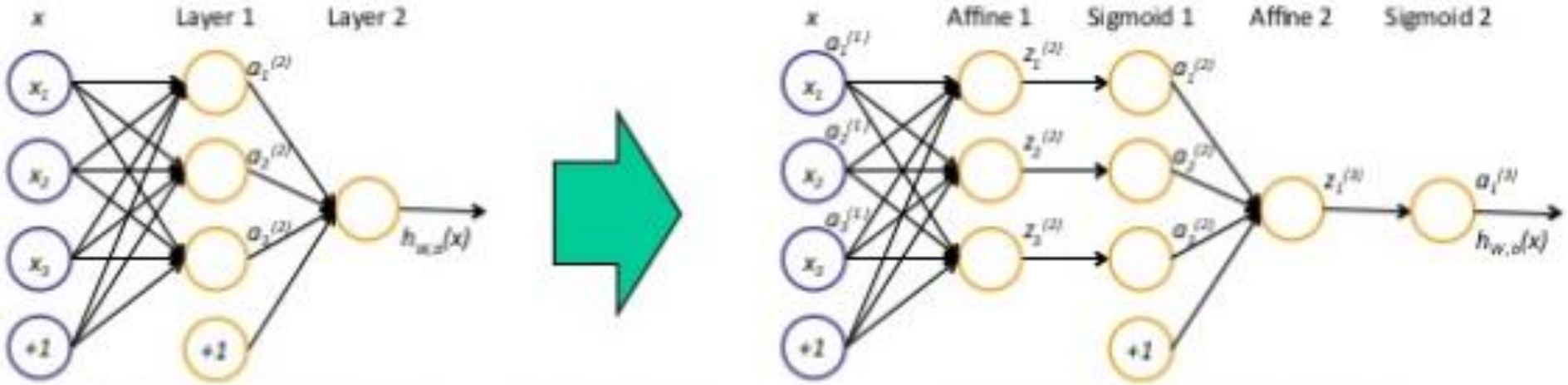
  Gradient of the cost function w.r.t. parameters for each example is computed using error signals and derivatives of function elements w.r.t. parameters. Summing gradients for all examples gets overall gradient.

$$\nabla_W J(W,b;x,y)=\frac{\partial}{\partial W}J(W,b;x,y)=\frac{\partial J}{\partial z}\frac{\partial z}{\partial W}=\delta^{(z)}x^T$$

$$\nabla_b J(W,b;x,y)=\frac{\partial}{\partial b}J(W,b;x,y)=\frac{\partial J}{\partial z}\frac{\partial z}{\partial b}=\delta^{(z)}$$

Decomposition of Multi-Layer Neural Network

- Composite function representation of a multi-layer neural network



$$h_{W,b}(x) = \left(\text{sigmoid} \circ \text{affine}_{W^{(2)},b^{(2)}} \circ \text{sigmoid} \circ \text{affine}_{W^{(1)},b^{(1)}}\right)(x)$$

- Derivatives of function elements w.r.t. inputs and parameters

$$a^{(1)} = x, a^{(l_{max})} = h_{w,b}(x)$$

$$\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} = a^{(l+1)} \bullet \left(1 - a^{(l+1)}\right) \text{ where } a^{(l+1)} = \text{sigmoid}\left(z^{(l+1)}\right) = \frac{1}{1+\exp\left(-z^{(l+1)}\right)}$$

$$\frac{\partial z^{(l+1)}}{\partial a^{(l)}} = W^{(l)}, \frac{\partial z^{(l+1)}}{\partial W^{(l)}} = a^{(l)}, \frac{\partial z^{(l+1)}}{\partial b^{(l)}} = I \text{ where } z^{(l+1)} = \left(W^{(l)}\right)^T a^{(l)} + b^{(l)}$$

Error Signals and Gradients in Multi-Layer NN

- Error signals of the square error cost function for each example

$$\delta^{(a^{(l)})} = \frac{\partial}{\partial a^{(l)}} J(W,b;x,y) = \begin{cases} -\left(y - a^{(l)}\right) \text{ for } l = l_{\max} \\ \dfrac{\partial J}{\partial z^{(l+1)}} \dfrac{\partial z^{(l+1)}}{\partial a^{(l)}} = \left(W^{(l)}\right)^T \delta^{(z^{(l+1)})} \text{ otherwise} \end{cases}$$

$$\delta^{(z^{(l)})} = \frac{\partial}{\partial z^{(l)}} J(W,b;x,y) = \frac{\partial J}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} = \delta^{(a^{(l)})} \bullet a^{(l)} \bullet \left(1 - a^{(l)}\right)$$

- Gradient of the cost function w.r.t. parameters for each example

$$\nabla_{W^{(l)}} J(W,b;x,y) = \frac{\partial}{\partial W^{(l)}} J(W,b;x,y) = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial W^{(l)}} = \delta^{(z^{(l+1)})} \left(a^{(l)}\right)^T$$

$$\nabla_{b^{(l)}} J(W,b;x,y) = \frac{\partial}{\partial b^{(l)}} J(W,b;x,y) = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial b^{(l)}} = \delta^{(z^{(l+1)})}$$

Backpropagation in General Cases

1. Decompose operations in layers of a neural network into function elements whose derivatives w.r.t inputs are known by symbolic computation.

$$h_\theta(x) = \left(f^{(l_{max})} \circ \ldots \circ f^{(l)}_{\theta^{(l)}} \circ \ldots \circ f^{(2)}_{\theta^{(2)}} \circ f^{(1)}\right)(x) \text{ where } f^{(1)} = x,\ f^{(l_{max})} = h_\theta(x) \text{ and } \forall l : \frac{\partial f^{(l+1)}}{\partial f^{(l)}} \text{ is known}$$

2. Backpropagate error signals corresponding to a differentiable cost function by numerical computation (Starting from cost function, plug in error signals backward).

$$\delta^{(l)} = \frac{\partial}{\partial f^{(l)}} J(\theta; x, y) = \frac{\partial J}{\partial f^{(l+1)}} \frac{\partial f^{(l+1)}}{\partial f^{(l)}} = \delta^{(l+1)} \frac{\partial f^{(l+1)}}{\partial f^{(l)}} \text{ where } \frac{\partial J}{\partial f^{(l_{max})}} \text{ is known}$$

3. Use backpropagated error signals to compute gradients w.r.t. parameters only for the function elements with parameters where their derivatives w.r.t parameters are known by symbolic computation.

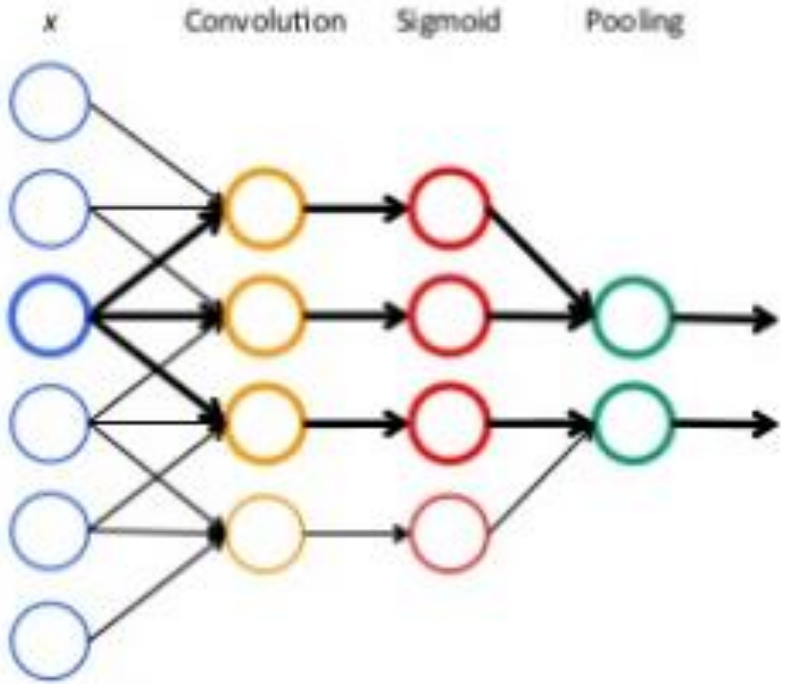
$$\nabla_{\theta^{(l)}} J(\theta; x, y) = \frac{\partial}{\partial \theta^{(l)}} J(\theta; x, y) = \frac{\partial J}{\partial f^{(l)}} \frac{\partial f^{(l)}_{\theta^{(l)}}}{\partial \theta^{(l)}} = \delta^{(l)} \frac{\partial f^{(l)}_{\theta^{(l)}}}{\partial \theta^{(l)}} \text{ where } \frac{\partial f^{(l)}_{\theta^{(l)}}}{\partial \theta^{(l)}} \text{ is known}$$

4. Sum gradients over all example to get overall gradient. $\nabla_{\theta^{(l)}} J(\theta) = \sum_{i=1}^{m} \nabla_{\theta^{(l)}} J\left(\theta; x^{(i)}, y^{(i)}\right)$

Convolutional Neural Network

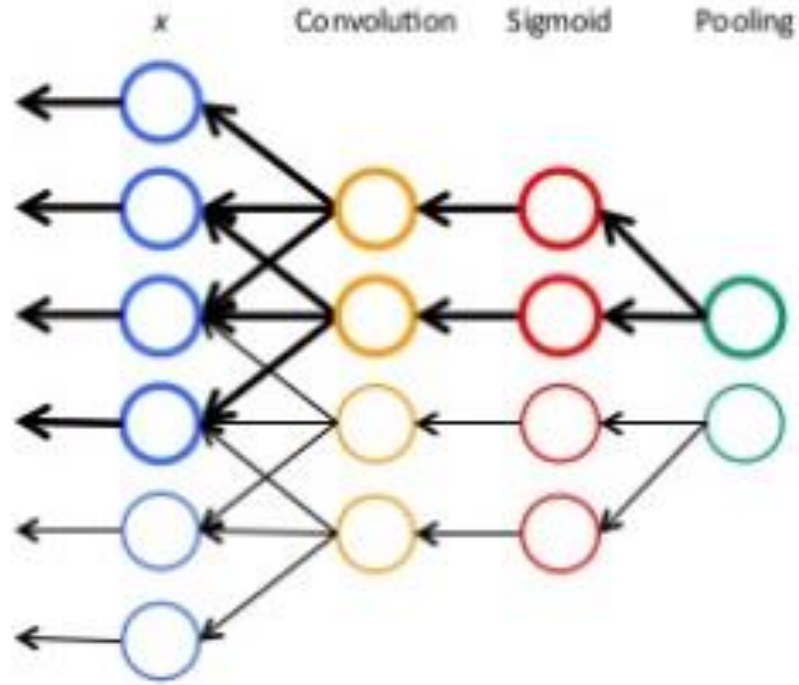
A convolution-pooling layer in Convolutional Neural Network is a composite function decomposed into function elements $f^{(conv)}$, $f^{(sigm)}$, and $f^{(pool)}$.

Let x be the output from the previous layer. Sigmoid nonlinearity is optional.



$$\left(f^{(pool)} \circ f^{(sigm)} \circ f_w^{(conv)}\right)(x)$$

Forward propagation



Backward propagation

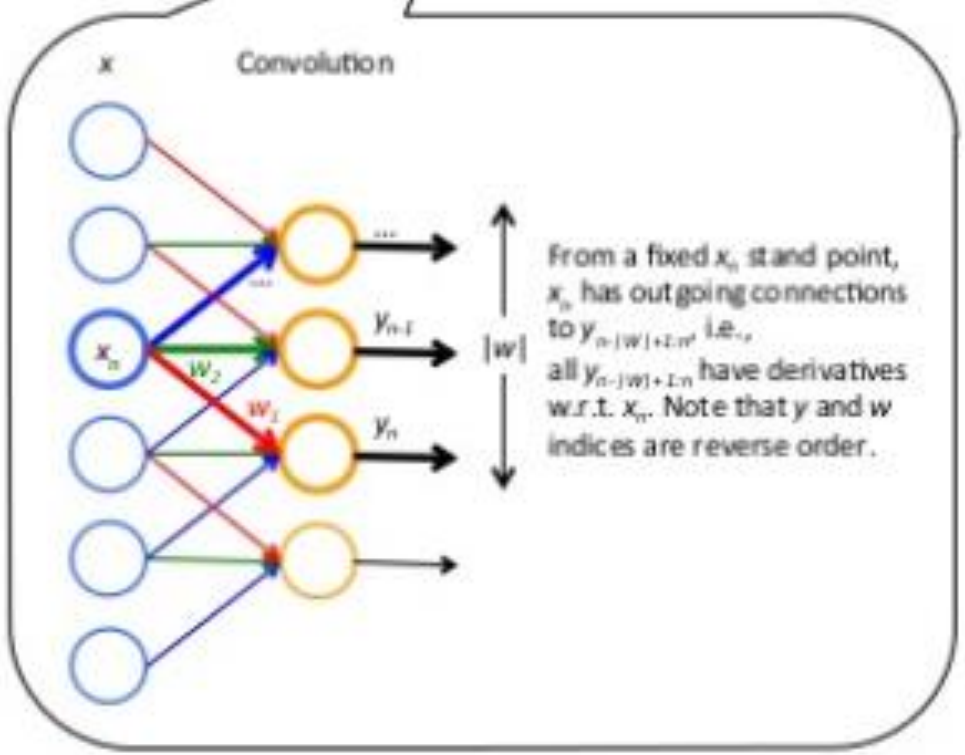
Derivatives of Convolution

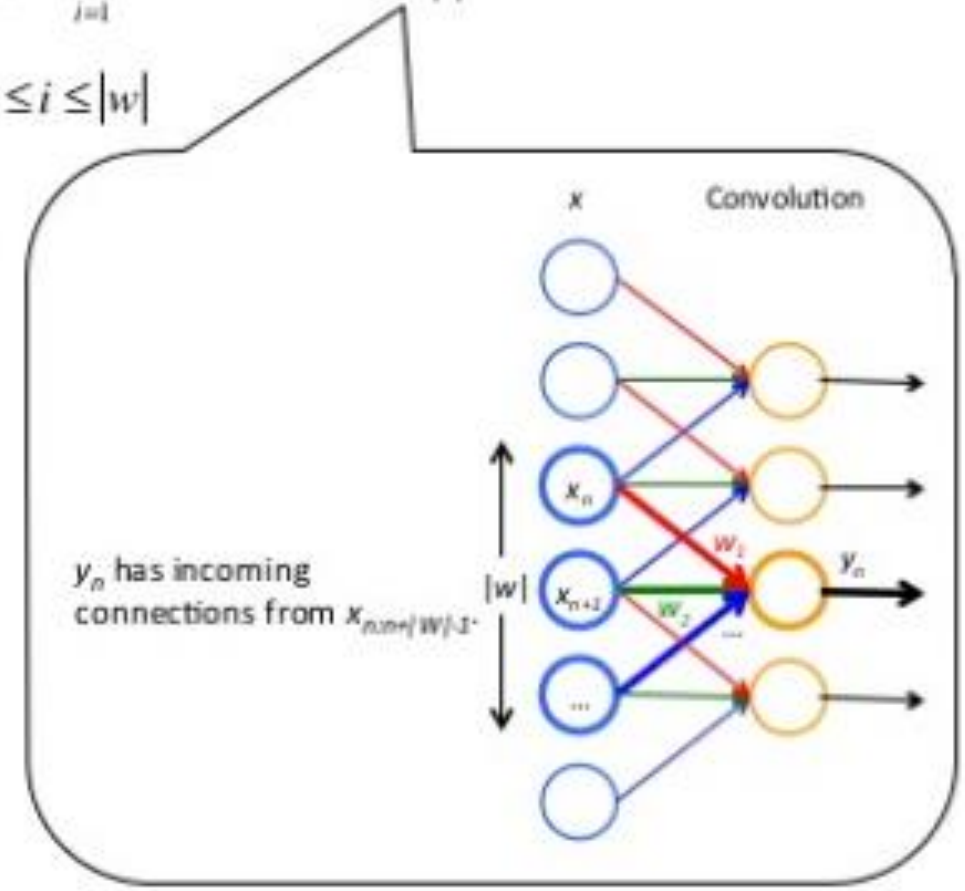
- Discrete convolution parameterized by a feature w and its derivatives

  Let x be the input, and y be the output of convolution layer. Here we focus on only one feature vector w, although a convolution layer usually has multiple features $W = [w_1\ w_2\ \dots\ w_n]$. n indexes x and y where $1 \le n \le |x|$ for x_n, $1 \le n \le |y| = |x| - |w| + 1$ for y_n. i indexes w where $1 \le i \le |w|$. $(f*g)[n]$ denotes the n-th element of $f*g$.

$$y = x * w = [y_n], y_n = (x * w)[n] = \sum_{i=1}^{|w|} x_{n+i-1} w_i = w^T x_{n:n+|w|-1}$$

$$\frac{\partial y_{n-i+1}}{\partial x_n} = w_i,\ \frac{\partial y_n}{\partial w_i} = x_{n+i-1} \text{ for } 1 \le i \le |w|$$





Backpropagation in Convolution Layer

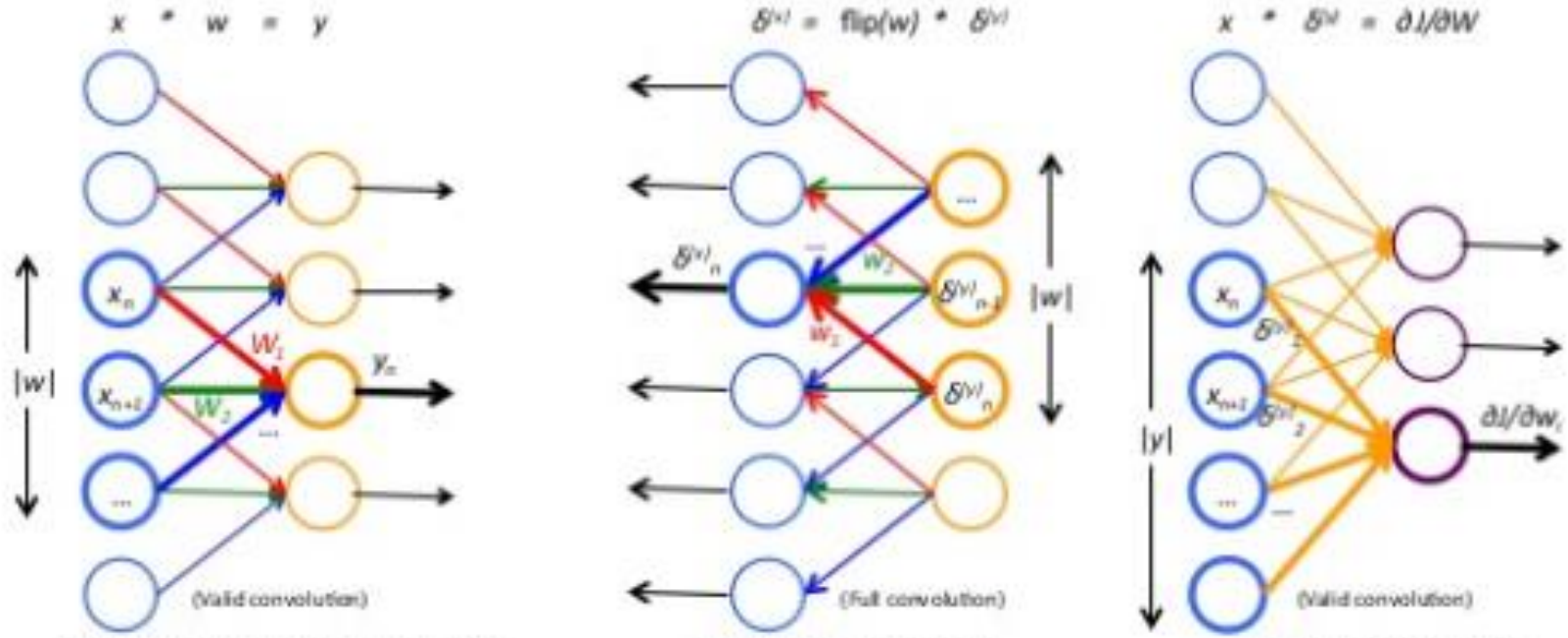
Error signals and gradient for each example are computed by convolution using the commutativity property of convolution and the multivariable chain rule of derivative.

Let's focus on single elements of error signals and a gradient w.r.t. w.

$$\delta_n^{(x)} = \frac{\partial J}{\partial x_n} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x_n} = \sum_{i=1}^{|w|}\frac{\partial J}{\partial y_{n-i+1}}\frac{\partial y_{n-i+1}}{\partial x_n} = \sum_{i=1}^{|w|}\delta_{n-i+1}^{(y)} w_i = \left(\delta^{(y)} * \mathrm{flip}(w)\right)[n],\ \delta^{(x)} = \left[\delta_n^{(x)}\right] = \delta^{(y)} * \mathrm{flip}(w)$$

↑ Reverse order linear combination

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial w_i} = \sum_{n=1}^{|x|-|w|+1}\frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_i} = \sum_{n=1}^{|x|-|w|+1}\delta_n^{(y)} x_{n+i-1} = \left(\delta^{(y)} * x\right)[i],\ \frac{\partial J}{\partial w} = \left[\frac{\partial J}{\partial w_i}\right] = \delta^{(y)} * x = x * \delta^{(y)}$$



Forward propagation (convolution) Backward propagation Gradient computation

Derivatives of Pooling

Pooling layer subsamples statistics to obtain summary statistics with any aggregate function (or filter) *g* whose input is vector, and output is scalar. Subsampling is an operation like convolution, however *g* is applied to disjoint (non-overlapping) regions.

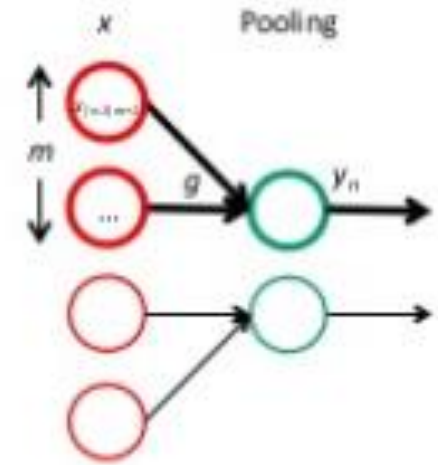
- Definition: *subsample* (or *downsample*)

  Let *m* be the size of pooling region, *x* be the input, and *y* be the output of the pooling layer. subsample(*f*, *g*)[*n*] denotes the *n*-th element of subsample(*f*, *g*).

$$y_n = \text{subsample}(x, g)[n] = g\left(x_{(n-1)m+1:nm}\right)$$

$$y = \text{subsample}(x, g) = [y_n]$$

$$g(x) = \begin{cases} \dfrac{\sum_{k=1}^{m} x_k}{m}, \dfrac{\partial g}{\partial x} = \dfrac{\mathbf{1}}{m} & \text{mean pooling} \\ \max(x), \dfrac{\partial g}{\partial x_j} = \begin{cases} 1 \text{ if } x_j = \max(x) \\ 0 \text{ otherwise} \end{cases} & \text{max pooling} \\ \|x\|_p = \left(\sum_{k=1}^{m} |x_k|^p\right)^{1/p}, \dfrac{\partial g}{\partial x_j} = \left(\sum_{k=1}^{m} |x_k|^p\right)^{1/p-1} |x_j|^{p-1} & \text{L}^p \text{ pooling} \\ \text{or any other differentiable } \mathbf{R}^m \to \mathbf{R} \text{ functions} \end{cases}$$



Backpropagation in Pooling Layer

Error signals for each example are computed by upsampling. Upsampling is an operation which backpropagates (distributes) the error signals over the aggregate function g using its derivatives $g'_n = \partial g / \partial x_{(n-1)m+1:nm}$. g'_n can change depending on pooling region n.

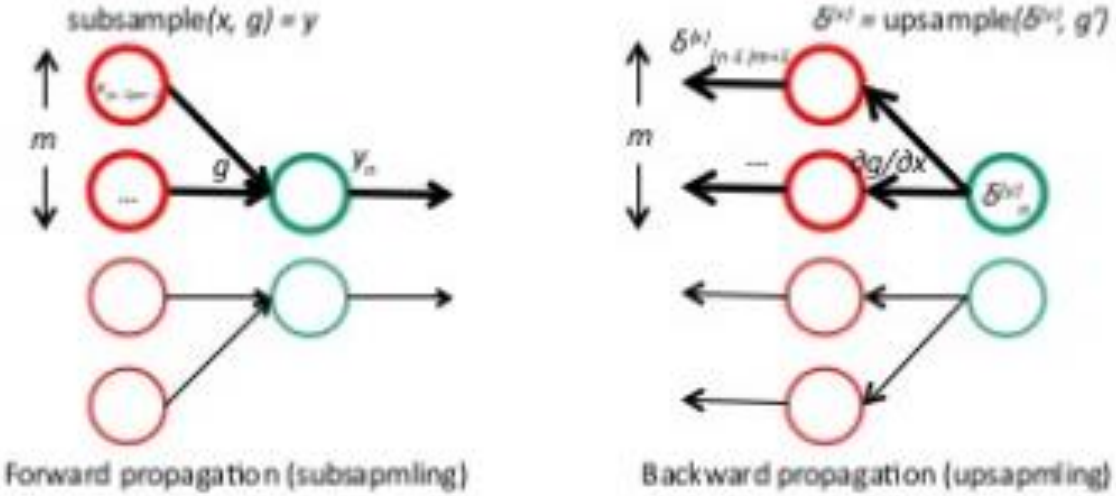
- In max pooling, the unit which was the max at forward propagation receives all the error at backward propagation and the unit is different depending on the region n.

- Definition: *upsample*

  upsample(f, g)[n] denotes the n-th element of upsample(f, g).

$$\delta^{(s)}_{(n-1)m+1:nm} = \text{upsample}\left(\delta^{(s)}, g'\right)[n] = \delta^{(s)}_n g'_n = \delta^{(s)}_n \frac{\partial g}{\partial x_{(n-1)m+1:nm}} = \frac{\partial J}{\partial y_n} \frac{\partial y_n}{\partial x_{(n-1)m+1:nm}} = \frac{\partial J}{\partial x_{(n-1)m+1:nm}}$$

$$\delta^{(s)} = \text{upsample}\left(\delta^{(s+1)}, g'\right) = \left[\delta^{(s)}_{(n-1)m+1:nm}\right]$$



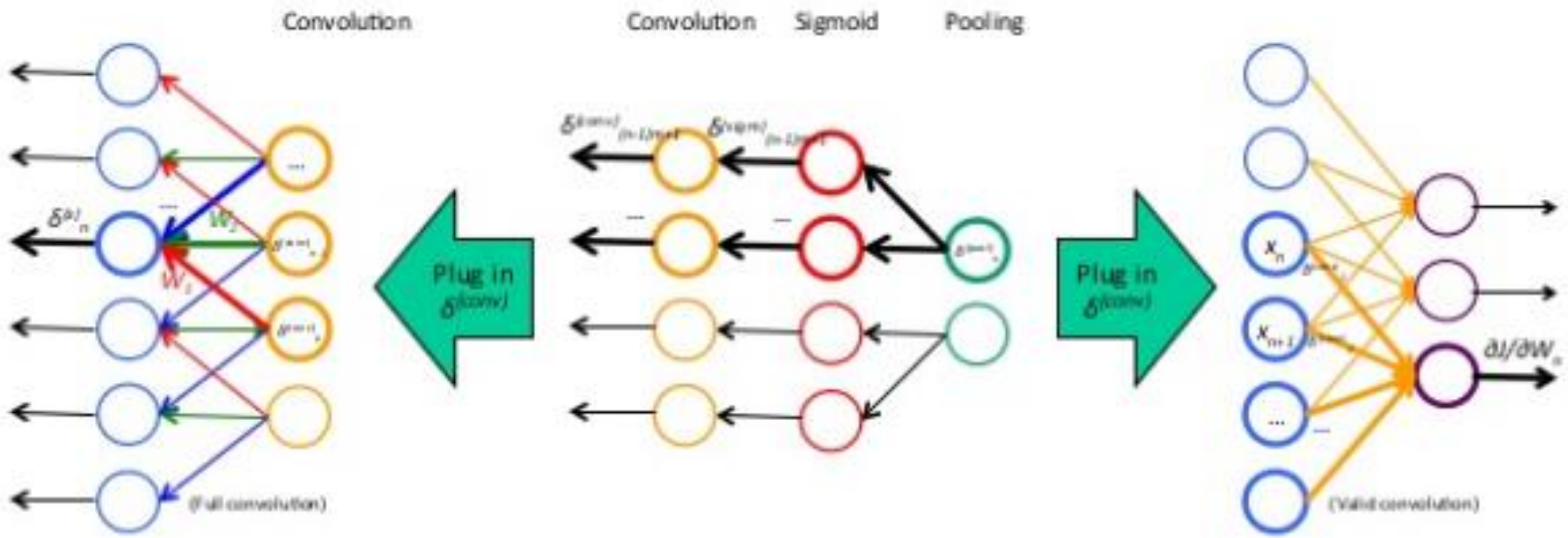
Forward propagation (subsapmling) Backward propagation (upsapmling)

Backpropagation in CNN (Summary)

2. Propagate error signals $\delta^{(conv)}$

1. Propagate error signals $\delta^{(pool)}$

3. Compute gradient $\nabla_w J$



$$\delta^{(s)} = \delta^{(conv)} * \text{flip}(w)$$

$$\delta^{(conv)} = \text{upsample}\left(\delta^{(pool)}, g'\right) \bullet \underbrace{f^{(sigm)} \bullet \left(1 - f^{(sigm)}\right)}_{\text{Derivative of sigmoid}}$$

$$x * \delta^{(conv)} = \nabla_w J$$

NN Backpropagation을 코드로 확인할 수 있는 자료

https://github.com/rasmusbergpalm/DeepLearnToolbox/tree/master/NN

CNN Backpropagation을 코드로 확인할 수 있는 자료

https://github.com/rasmusbergpalm/DeepLearnToolbox/tree/master/CNN