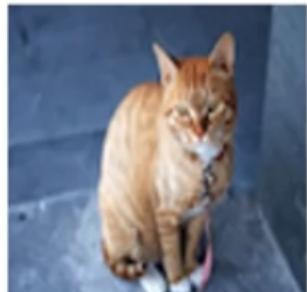




4th Course: CNN

week1

Computer Vision



64x64 × 3

→ Cat? (0/1)

12288



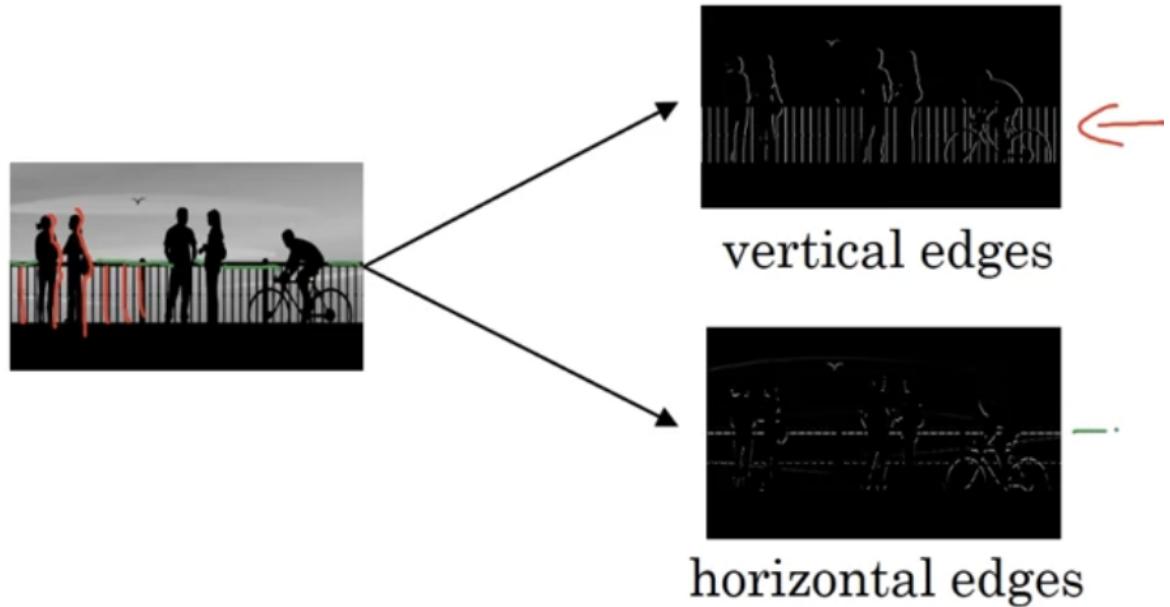
$1000 \times 1000 \times 3$
= 3 million

- $n \times n \times 3$ 형식에서 n 이 커질수록 해상도가 높다
- 3은 RGB를 의미한다

Edge detection example



- 이미지 안의 엣지를 어떻게 감지하는지 알아보는시간

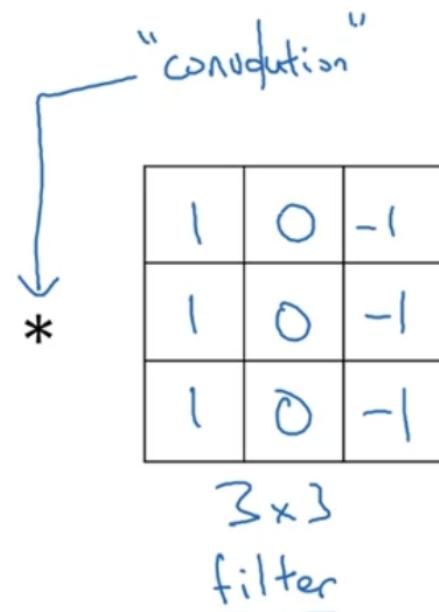


- 처음 이미지가 주어졌을 때 vertical edges(수직 엣지)를 먼저 확인한다
- 이후 horizontal edges(수평 엣지) 검출
- 그래서 이런 이미지에서는 어떻게 엣지를 검출할까?(방법)

Vertical edge detection

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

6×6



- 왼쪽은 이미지, 오른쪽은 filter(kernel)이다. * 연산 기호는 컨볼브(합성곱) 연산이다.
(파이썬에서는 곱셈을 의미)
- $6 \times 6 * 3 \times 3$ 결과는 아래와 같다

=

4×4

3^1	0^0	1^{-1}
1^1	5^0	8^{-1}
2^1	7^0	2^{-1}

- 4×4 행렬이 생성

- 그럼 어떻게 4×4 행렬이 생성된 것일까? → 필터를 가지고 이미지를 도장찍는다고 생각하기
- $(3 \times 1 + 1 \times 1 + 2 \times 1) + (0 \times 0 + 0 \times 5 + 0 \times 7) + (1 \times -1 + 8 \times -1 + 2 \times -1) = -5$ 의 값이 4×4 의 첫번째 칸을 채운다

$$3 \times 1 + 1 \times 1 + 2 \times 1 + 0 \times 0 + 0 \times 5 + 0 \times 7 + 0 \times -1 + 1 \times -1 + 8 \times -1 + 2 \times -1 = -5$$

$\begin{matrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{matrix}$
 $\begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$
 $\begin{matrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{matrix}$

 $\begin{matrix} 6 \times 6 & & 3 \times 3 & & 4 \times 4 \end{matrix}$

- 그럼 왜 수직선 엣지 검출을 하는 것일까?
 - 급격하게 변하는 구간을 이미지, 영상처리에서 수행하는 이유?

$\begin{matrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{matrix}$
 $\begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$
 $\begin{matrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{matrix}$

 $\begin{matrix} 6 \times 6 & & 3 \times 3 & & 4 \times 4 \end{matrix}$

 $\begin{matrix} \downarrow & \downarrow \\ \uparrow & \uparrow \end{matrix}$
 $\begin{matrix} \downarrow & \downarrow & \downarrow \\ \uparrow & \uparrow & \uparrow \end{matrix}$
 $\begin{matrix} \downarrow \\ \uparrow \end{matrix}$

Andrew Ng

- 왼쪽 10이 밝은, 0은 어두운 값
- 오른쪽 필터는 밝 / 중간 / 어두움이 공존

- 결과로 6×6 이미지의 중간에 수직 엣지가 검출된 것처럼 중간에는 연한 픽셀의 구역이 생김
 - 이미지 중간 바로 아래에 강한 수직 엣지가 있는 것처럼 보이는 출력 이미지
- 그래서 왜 한다는 거지?
 - 합성곱 연산은 이미지에서 편리하게 수직 엣지를 찾고 구체화시킬 수 있다

More edge detection

$$\begin{array}{c}
 \begin{array}{|c|c|c|c|c|c|c|} \hline
 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ \hline
 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ \hline
 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ \hline
 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ \hline
 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ \hline
 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ \hline
 \end{array} * \begin{array}{|c|c|c|} \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 \end{array} = \boxed{\begin{array}{|c|c|c|c|} \hline
 0 & 30 & 30 & 0 \\ \hline
 0 & 30 & 30 & 0 \\ \hline
 0 & 30 & 30 & 0 \\ \hline
 0 & 30 & 30 & 0 \\ \hline
 \end{array}}
 \\
 \xrightarrow{\quad\quad\quad} \boxed{\begin{array}{|c|c|} \hline
 \text{white} & \text{black} \\ \hline
 \end{array}}
 \end{array}
 \\
 \begin{array}{c}
 \begin{array}{|c|c|c|c|c|c|} \hline
 0 & 0 & 0 & 10 & 10 & 10 \\ \hline
 0 & 0 & 0 & 10 & 10 & 10 \\ \hline
 0 & 0 & 0 & 10 & 10 & 10 \\ \hline
 0 & 0 & 0 & 10 & 10 & 10 \\ \hline
 0 & 0 & 0 & 10 & 10 & 10 \\ \hline
 0 & 0 & 0 & 10 & 10 & 10 \\ \hline
 \end{array} * \begin{array}{|c|c|c|} \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 1 & 0 & -1 \\ \hline
 \end{array} = \boxed{\begin{array}{|c|c|c|c|} \hline
 0 & -30 & -30 & 0 \\ \hline
 0 & -30 & -30 & 0 \\ \hline
 0 & -30 & -30 & 0 \\ \hline
 0 & -30 & -30 & 0 \\ \hline
 \end{array}}
 \\
 \xrightarrow{\quad\quad\quad} \boxed{\begin{array}{|c|c|} \hline
 \text{black} & \text{white} \\ \hline
 \end{array}}
 \end{array}
 \end{array}$$

- 반대의 경우는 위와 같다. -1 때문에 가운데가 음수가 되어 어두워진다
- 위 두 가지 경우 중 어느 것인지 상관하지 않는다면 출력 행렬의 절댓값을 취할 수 있다

Horizontal

$$\begin{array}{|c|c|c|c|c|c|} \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 10 & 10 & 10 \\ \hline 0 & 0 & 0 & 10 & 10 & 10 \\ \hline 0 & 0 & 0 & 10 & 10 & 10 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -1 & -1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 0 \\ \hline 30 & 10 & -10 & -30 \\ \hline 30 & 10 & -10 & -30 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

- 필터의 형태가 바뀐 것을 알 수 있다
- 요약하자면, 다양한 필터를 사용하여 수직 및 수평 엣지를 찾을 수 있다

$$\begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array} \text{Sobel filter} \quad \begin{array}{|c|c|c|} \hline 3 & 0 & -3 \\ \hline 10 & 0 & -10 \\ \hline 3 & 0 & -3 \\ \hline \end{array} \text{Scharr filter}$$

- 2번째 필터 : 소벨 필터
 - 장점 : 중앙 행, 중앙 픽셀에 조금 더 무게를 실어 조금 더 견고하게 만듦
- 3번째 필터 : 샤르 필터
 - 오직 수직 엣지 검출만 가능

$$\begin{array}{|c|c|c|c|c|c|c|} \hline 3 & 0 & 1 & 2 & 7 & 4 \\ \hline 1 & 5 & 8 & 9 & 3 & 1 \\ \hline 2 & 7 & 2 & 5 & 1 & 3 \\ \hline 0 & 1 & 3 & 1 & 7 & 8 \\ \hline 4 & 2 & 1 & 6 & 2 & 8 \\ \hline 2 & 4 & 5 & 2 & 3 & 9 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline w_1 & w_2 & w_3 \\ \hline w_4 & w_5 & w_6 \\ \hline w_7 & w_8 & w_9 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

- 복잡한 이미지에서 엣지를 감지할 때 CV 연구원이 필터의 숫자를 직접 선택하도록 할 필요가 없다

- 역전파를 통해 배울 수 있는 9개의 숫자를 매개 변수로 다루게 될 것이다
- 목표는 9개의 매개변수를 학습하여 6×6 이미지로 만들고 3×3 필터로 컨볼브하여 좋은 엣지 검출기를 제공하는 것!
- 필터의 숫자들이 매개 변수가 되게하고 데이터(이미지)로부터 자동으로 학습함으로써 CV 연구원들이 일반적으로 손으로 코딩하는 것보다 훨씬 더 강력하게 신경망이 실제로 낮은 수준의 특징을 배울 수 있다는 것을 알 수 있다

Padding

- 위의 예시에서는 첫번째 결과 박스를 얻기위해서 입력 이미지를 처음부터 찍었다.(딱 들어맞게)
 - 그러므로 $6 \times 6 * 3 \times 3 = 4 \times 4$ 가 나오는 것이다
- $n \times n * f \times f = (n-f+1) \times (n-f+1) \rightarrow \text{no padding}$
 - no padding의 두 가지 단점
 - 합성곱 연산을 적용할 때마다 이미지가 축소
 - 딱 들어맞게 도장을 찍기 때문에 모서리나 가장자리에 있는 픽셀은 다른 픽셀들에 비해 사용이 덜 된다 \rightarrow 간단하게 공평성 x, 이 말은 즉 이미지의 엣지에서 많은 정보가 버려진다는 것
- 두 가지 단점을 해결하기 위해서 합성곱 연산을 최대로 적용해주는 **패딩** 적용
 - 가장자리에 추가 \rightarrow 결과를 입력 이미지와 동일하게 맞출 수 있음
 - Ex) $8 \times 8 * 3 \times 3 = 6 \times 6$
 - 8은 6(입력이미지) + 2(패딩)
- 패딩 추가 공식
 - $(n+2p-f+1) \times (n+2p-f+1)$
 - 패딩을 추가해서 이미지의 모서리나 엣지의 정보를 결과 이미지에 동일하게 적용할 수 있다

Valid and Same convolutions

- 패딩의 양에 관해서는 Valid 합성곱과 Same 합성곱이라는 두 가지 일반적인 옵션이 존재

- Valid convolutions : 기본적으로 패딩이 없는 것을 의미
 - $n \times n * f \times f = (n-f+1) \times (n-f+1)$
- Same convolutions : 패딩을 할 때 출력 크기는 입력 크기와 동일
 - $(n+2p-f+1) \times (n+2p-f+1)$
 - 입력 크기와 출력 크기가 동일하게 되려면 다음 수식 과정이 필요
 - $n = n+2p-f+1$
 - $p = (f - 1) / 2$
 - 위의 예시를 들면 f 가 3이므로 p 는 2이다!
 - f 는 주로 홀수! → 그래야지 p 가 정수가 나온다!
 - f 가 짝수일 확률은 적지만, 짝수일 경우 비대칭 패딩을 수행해야 한다

Strided convolutions

- 여기서 stride란 step 을 의미한다
 - 결국 stride가 2라면 2칸을 뛰어서 도장을 찍는다는 것!

The diagram shows a 7x7 input matrix and a 3x3 kernel matrix. The input matrix has values from 0 to 9. The kernel matrix has values 3, 4, 4; 1, 0, 2; and -1, 0, 3. The result of the multiplication is a 3x3 output matrix with values 91, 100, 83; 69.; and empty cells. A blue arrow points from the top-left cell of the input matrix to the top-left cell of the kernel matrix, indicating the step size of 2.

2	3	7	4	6	2	9
6	6	9	8	7	4	3
3	4	8 ³	3 ⁴	8 ⁴	9	7
7	8	3 ¹	6 ⁰	6 ²	3	4
4	2	1 ⁻¹	8 ⁰	3 ³	4	6
3	2	4	1	9	8	3
0	1	3	9	2	1	4

$\begin{matrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{matrix}$

$\begin{matrix} 91 & 100 & 83 \\ 69. & & \\ & & \end{matrix}$

3×3

stride = 2

- strides 포함 식
- Ex) $n \times n * f \times f$, (padding) p , (strides) $s = 2$
 - $((n+2p-f) / s) + 1 \times ((n+2p-f) / s) + 1 \rightarrow$ 이 값에 np.floor 적용(내리기)

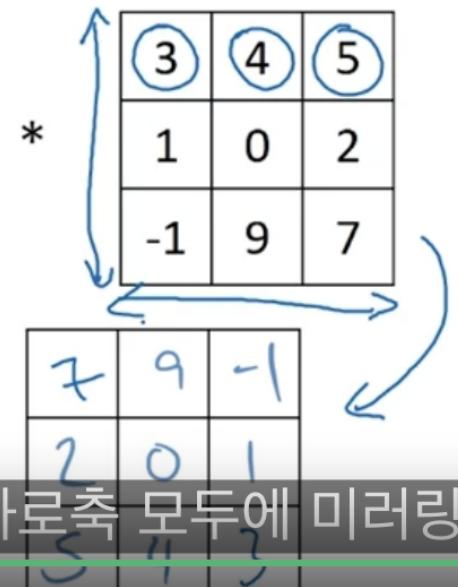
$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \quad \times \quad \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

- 보통 위 식이지만, 가끔은 반올림하는 것도 괜찮다!

Technical note on cross-correlations vs convolution

Convolution in math textbook:

2	3	7	4	6	2
6	6	9	8	7	4
3	4	8	3	8	9
7	8	3	6	6	3
4	2	1	8	3	4
3	2	4	1	9	8



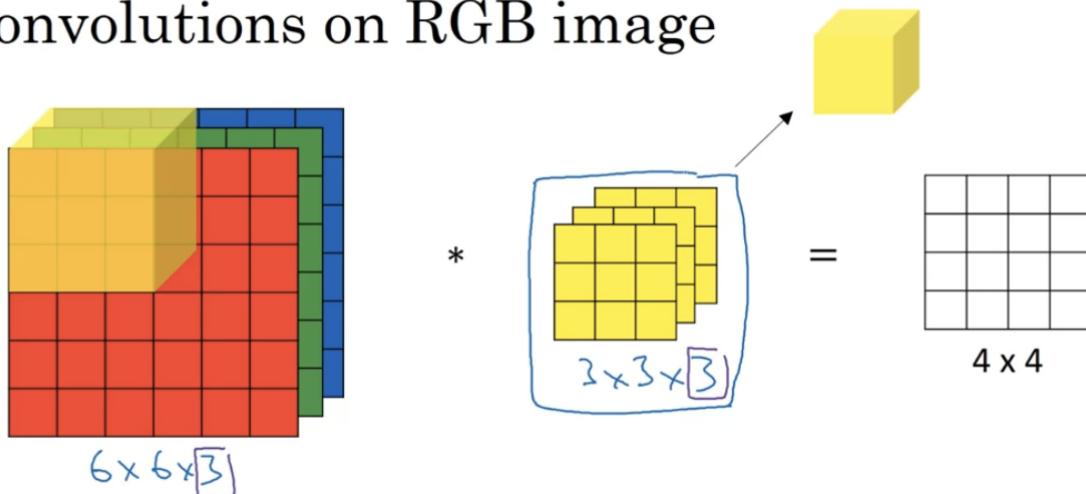
- 추가적으로, 수학적인 의미의 convolution과 CNN에서의 convolution은 방법이 조금 다르다!
- 수학적인 의미에서 convolution은 요소간의 곱을 진행하기 전에 수행해야되는 단계가 하나 더 있으며, 이때 filter를 상하좌우 반전을 시켜준다.
- CNN에서 수행하는 conv연산은 실제적으로는 cross_correlation에 해당하지만, 딥러닝에서는 그냥 convolution이라고 부른다
- 위와 같이 filter를 반전시켜주는 단계가 필요하지만, 이 부분은 신호처리에는 적합하지만 딥러닝과 크게 상관없기 때문에 반전을 시키는 단계(mirroring operation)은 생략한다!

Convolutions over volumes

Convolutions on RGB images

- $n \times n \times 3 \rightarrow$ 여기서 3은 RGB를 의미한다.
 - R + G + B 즉, 3개의 채널(layers)

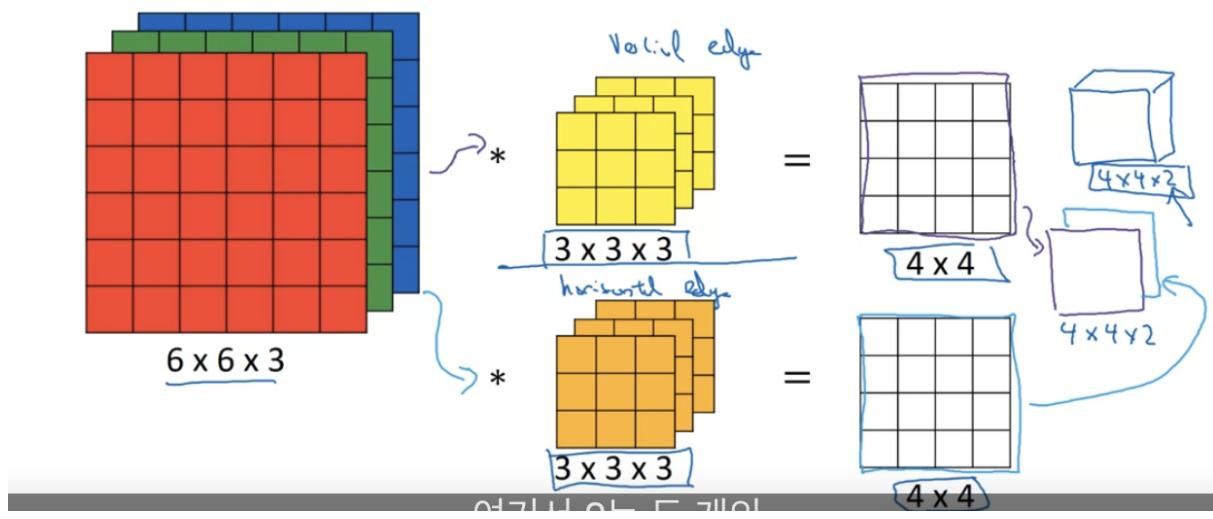
Convolutions on RGB image



- $3 \times 3 \times 3$ 필터를 정육면체라고 간주한 후 도장 찍기
 - 왼쪽 입력 이미지도 channel이 3이기 때문에 딱 알맞게 RGB에 적용 가능
- 빨간색 채널(R)의 엣지를 검출하려 한다면 첫번째 필터는 3×3 형태에
1,1,1,0,0,0,-1,-1,-1
초록색 채널(G)은 0,0,0,0,0,0,0,0,0
파란색 채널(B) 또한 0,0,0,0,0,0,0,0,0
- 결국 엣지를 검출하고 싶은 채널에 적용하고, 나머지는 0으로 적용
- 일반화하면 입력 이미지와 필터의 채널이 같아야지 엣지 검출 등 컨볼루션 계산이 가능하다
- 여기서 $6 \times 6 \times 3 * 3 \times 3 \times 3$ 을 수행하면 2D의 4×4 출력이미지를 얻는다

Multiple filters

Multiple filters

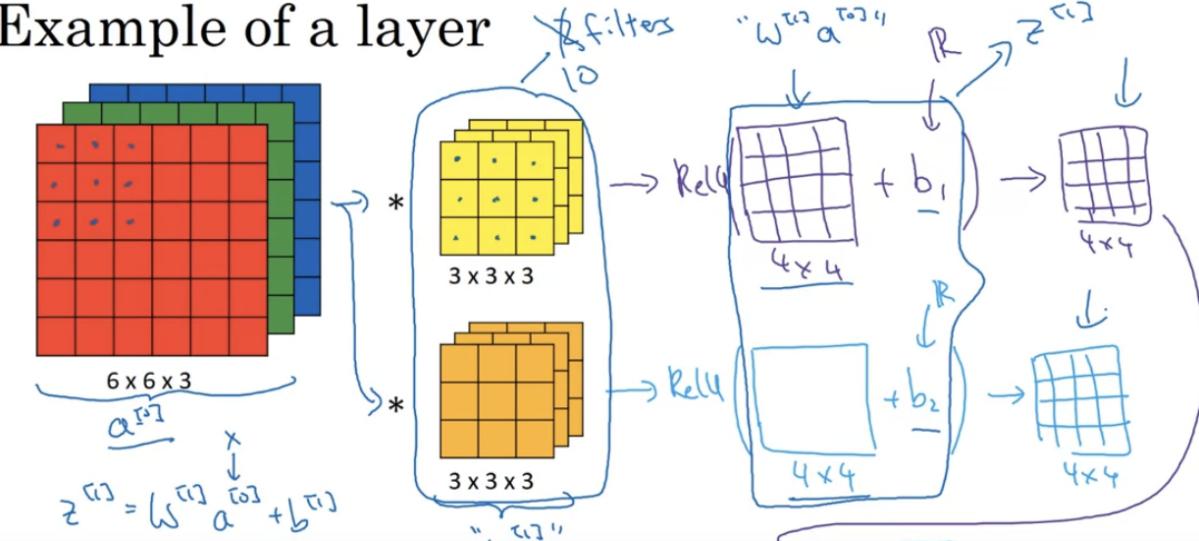


- $3 \times 3 \times 3$ 필터의 종류가 수평, 수직으로 2개의 종류이다.
- 결국 각각 컨볼루션을 적용하면 4×4 출력 이미지를 얻을 수 있는데 이 2개를 합치면 $4 \times 4 \times 2$ 를 얻을 수 있다
- Summary
 - $n \times n \times n_c * f \times f \times n_c = (n-f+1) \times (n-f+1) \times n_c'$
 - 여기서 n_c' 은 필터의 개수(위에서 합친 것을 의미)

One layer of a convolutional network

- 출력이미지를 얻은 후 편향(bias)를 더해야 한다 → bias는 실수
 - 이때 Broadcast 적용됨, 즉 결과는 결과이미지 행렬
- bias를 더하고 활성화함수로 감싼다(결국 1장에서 배운 신경망과 유사)

Example of a layer



Number of parameters in one layer

- 2개가 아닌 10개의 필터가 있고 각각 $3 \times 3 \times 3$ 크기로 신경망의 한 레이어에 있다면 이 레이어는 몇 개의 매개변수를 가질까요?
- $3 \times 3 \times 3 \Rightarrow 27$ parameters, 여기에 bias가 추가되므로 총 28개 변수를 갖게 됨
- 여기에 n_c , 즉 filter 개수를 늘리면 280개의 매개변수를 갖게 된다
- 입력 이미지가 어떻게 되었든 필터는 항상 280 parameters를 갖음 \rightarrow 결국 아주 큰 이미지라도 적은 수의 변수로 검출이 가능하다! \rightarrow 이것이 과적합을 막아주는 합성곱 신경망의 한 성질!
- 이건 따로 찾아보기 !!

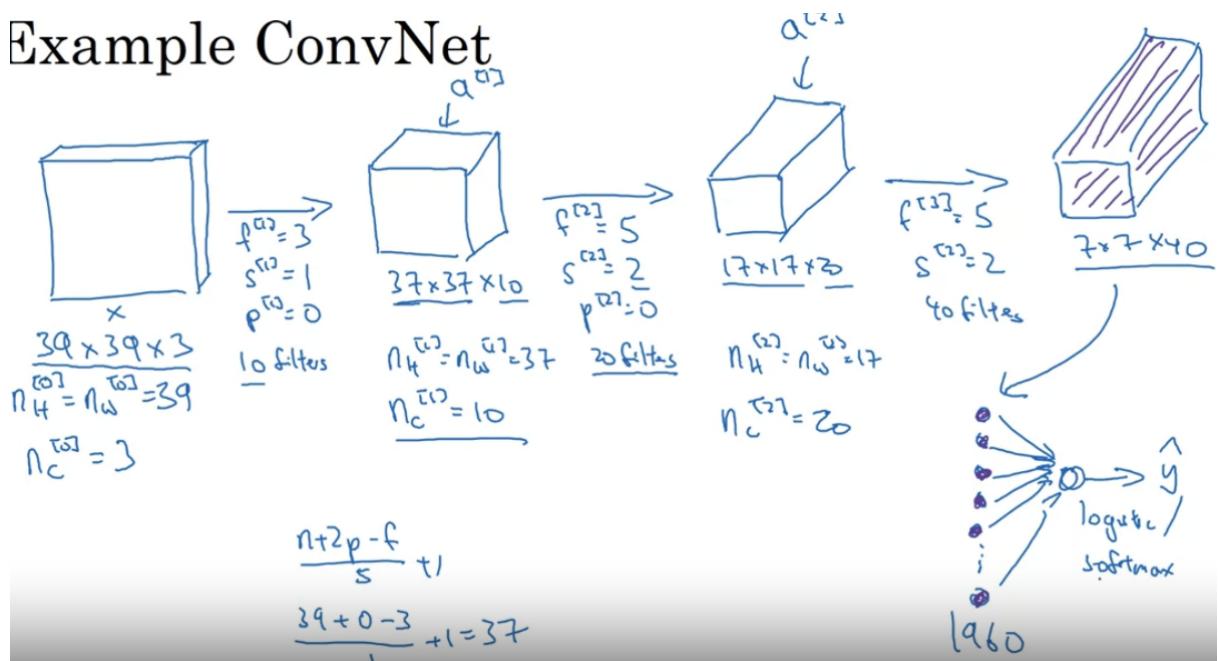
Summary of notation

- If layer l is a convolution layer:
 - f^l = filter size ($f \times f$)
 - p^l = padding
 - s^l = stride
 - Input : $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
 - Output : $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$, $n_{H,W}^{[l]} = \lfloor \frac{n^{[l-1]} + 2p^l - f^l}{s} + 1 \rfloor$
 - $n_c^{[l]}$ = number of filters

- Each filter is : $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
 - Activations : $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- $A^l \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ (m은 example의 수를 의미)
- Weights : $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ ($n_c^{[l]}$ 개의 filter 존재)
 - bias : $n_c^{[l]}$, 실수이므로 차원으로 표현 $\rightarrow 1 \times 1 \times 1 \times n_c^{[l]}$ 가 됨
- 위 notation에서 보편적인 표준규칙은 존재하지 않는다.

A simple convolution network example

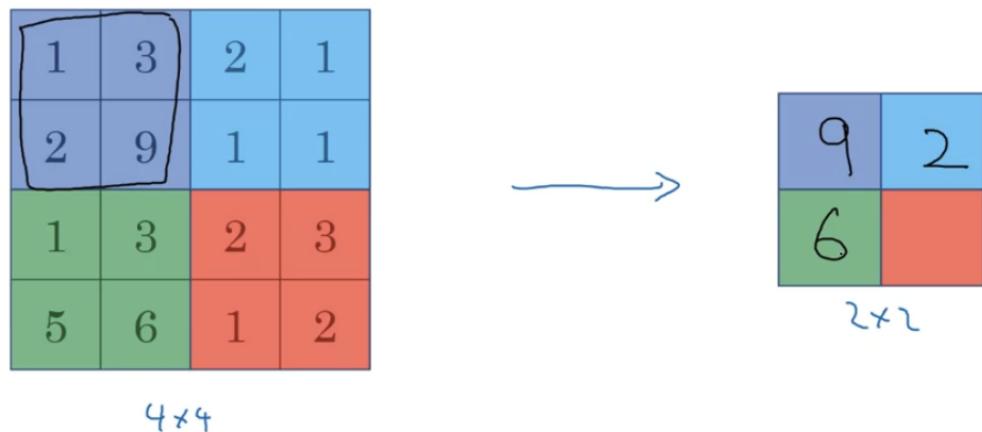
Example ConvNet



- 스트라이드, 패딩 ... 이것들 모두 하이퍼 파라미터!
- 위 예제에서 보이듯이 이미지의 높이, 너비는 줄어들고 채널은 커지는 것을 알 수 있다

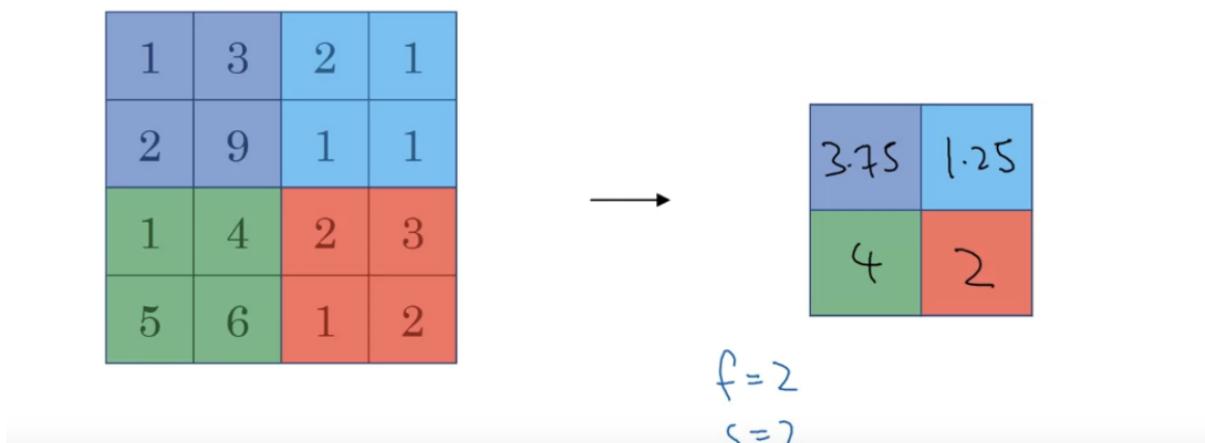
Pooling layers

Pooling layer: Max pooling



- Max pooling은 도장에서 가장 큰 값을 사용하는 것!
 - 위의 예시는 $p=2, s=2 \rightarrow$ hyper-parameter
- Max pooling이 하는 작업은?
 - 한 특성이 필터의 한 부분에서 검출되면 높은 수를 남기고, 만약 특성이 검출되지 않아서 특성이 오른쪽 상단 사분면에 존재하지 않으면 그 안의 최대값은 여전히 작은 수로 남게 된다
 - 즉, 추출된 각 지역별 피처의 특성을 강조
 - 예를 들면 앞선 레이어들을 거치고 나서 나온 output feature map의 모든 data가 필요하지 않기 때문!, 결국 추론하는 데 있어 적당량의 데이터만 있어도 되기 때문에 Max pooling을 사용한다.
 - 이에 대한 효과로 parameter가 줄어들기 때문에 과적합을 억제하고, 계산량이 줄어들어 하드웨어 리소스를 절약해서 속도가 빠르다

Pooling layer: Average pooling



- Max pooling과 달리 평균값(Average)를 사용!
- 사용 빈도 수는 Max pooling이 더 높다
- 하지만 신경망의 아주 깊은 곳에서는 Average pooling을 사용하기도 한다

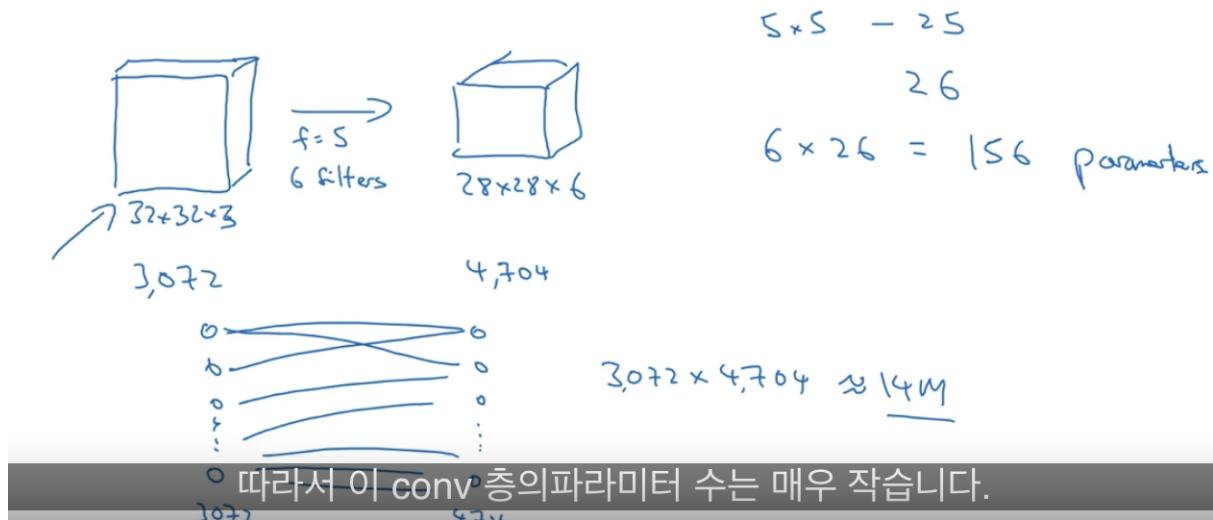
Summary of pooling

- Hyper-parameters
 - f : filter size
 - s : stride
 - Max or average pooling
 - 이유는 다음 강의에 나오는데 Max or average pooling에서 padding은 0으로 설정(즉, hyper-parameter가 아니다)

Why Convolutional network?

- 장점
 - 파라미터 공유와 연결의 희소성

Why convolutions



- $32 \times 32 \times 3(3,072) \rightarrow 28 \times 28 \times 6(4,704)$
- 한 층에 3,072개의 유닛이 있는 신경망을 만들고, 다음 층에 4,704개의 유닛이 있는 신경망을 만든 후 각각 연결하면 가중 매트릭스의 파라미터 수는 $3,072 \times 4,704$ 가 될 것이며 이는 약 1,400만 개이다.
 - 이것은 훈련시킬 파라미터가 많이 있다는 것
- 만약 $1,000 \times 1,000$ 이미지라면 훈련시킬 파라미터 수는 엄청 커진다는 것을 알 수 있다
- 필터가 5×5 이라면 25 parameters, + 1(bias) 이므로 26 parameters이다. filter가 6개라면 총 156 parameters와 같다. 따라서 이 conv 층의 파라미터 수는 매우 작다.
- Conv 네트워크가 작은 파라미터에 도달한 이유는 2가지!
 - 파라미터 공유 : 수직 옆지를 적용하기 위해 3×3 필터를 형성한 후 도장을 찍어나가면서 특성을 감지할 수 있는데 이는 결국 파라미터를 공유(필터)하는 것이다. 동일한 파라미터를 사용하여 출력을 모두 계산 가능! → 파라미터 수 줄이는 방법이다.
 - 연결의 희박함 : 도장을 찍을 때 다른 것에 영향을 주지 않음, 다시 말해서 모든 픽셀 값이 출력에 영향을 미치지 않는다는 것(도장을 찍어서 결국 출력 이미지 1칸에 영향을 주기 때문)

week2

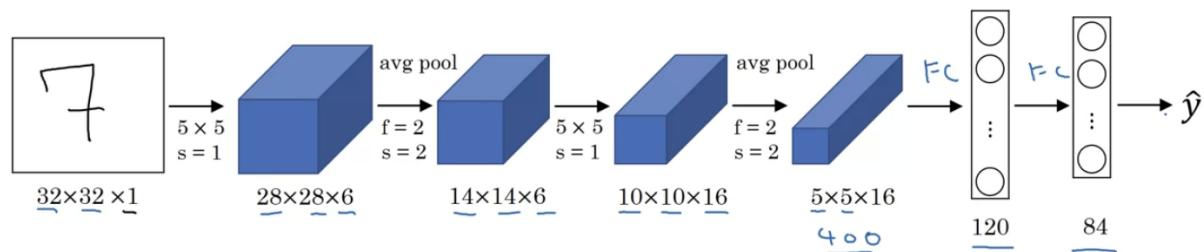
Why look at case studies?

Outline

- Classic networks:
 - LeNet-5
 - AlexNet
 - VGG
- ResNet

Classic networks

LeNet - 5

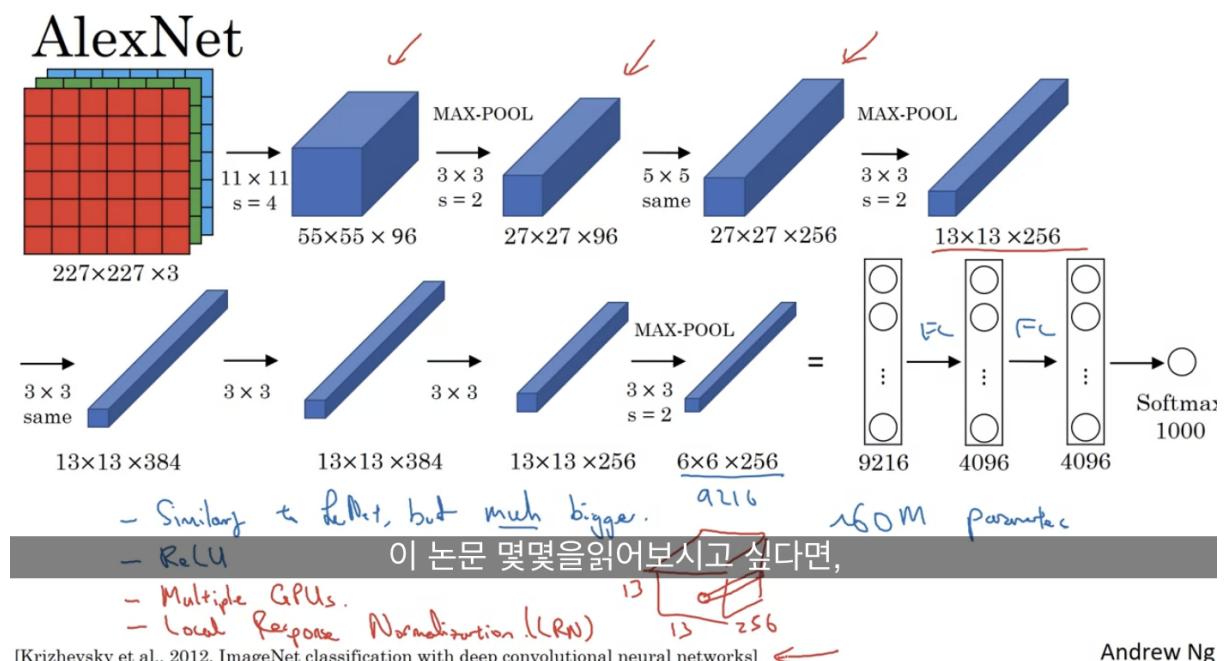


- 처음에 패딩이 없고 스트라이드가 하나인 Conv 적용해서 $32 \times 32 \rightarrow 28 \times 28$ 로 줄어듦
- 이후 avg pool 적용(옛날이니까 avg pool 사용)
- 적용하면 $14 \times 14 \times 6$ ($f=2, s=2$ 이니까 2배 줄어듦)
- 이후 처음에 적용한 Conv 적용, $10 \times 10 \times 16$
- 또 avg pool 적용해서 $5 \times 5 \times 16 \rightarrow 400$
- 그리고 다음 층은 fully-connected layer로 400개의 노드 각각 120개의 뉴런을 연결
- 또 fully-connected layer를 적용해서 84
- 그림상에서는 없지만 $y\hat{}$ 은 0 ~ 9까지의 숫자를 인식 할 때 가능한 10개의 가능한 값을 취했다.

- 해당 신경망의 최신 버전에서는 10방향 분류 출력에 softmax층 사용
- 약 60,000 parameters를 가짐
- 그림을 보면 오른쪽으로 진행할 수록 높이와 폭이 줄어드는 경향이 있다는 것을 알 수 있다

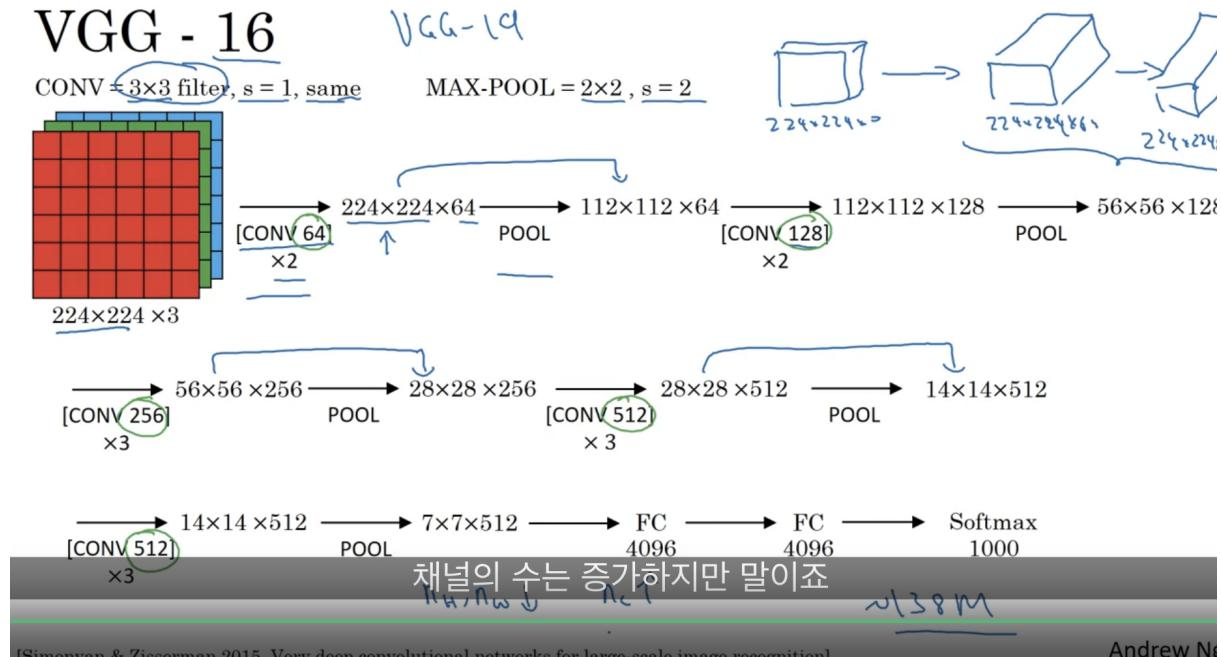
층 순서 설명

conv → pool → conv → pool → fc → fc → output



- $227 \times 227 \times 3$ 이미지 시작
- $11 \times 11, s = 4$ 필터 적용해서 $55 \times 55 \times 96$ 얻기
- $3 \times 3, s = 2$ filter Max pool 적용해서 $27 \times 27 \times 96$ 얻기
- 5×5 의 같은 Conv 적용해서 $27 \times 27 \times 256$ 얻기
- $3 \times 3, s = 2$ filter Max pool 적용해서 $13 \times 13 \times 256$ 얻기
- 3×3 의 같은 Conv 적용해서 $13 \times 13 \times 384$ 얻기
- 3×3 의 같은 Conv 적용해서 $27 \times 27 \times 384$ 얻기
- 3×3 의 같은 Conv 적용해서 $13 \times 13 \times 256$ 얻기
- $3 \times 3, s = 2$ filter Max pool 적용해서 $6 \times 6 \times 256$ 얻기 $\rightarrow 9,216$
- $9,216$ 유닛 fc 2번 적용해서 $4,096$ 유닛 2번 얻고 softmax 적용

1. LeNet과 많은 유사점을 가지고 있었지만, 훨씬 더 커졌다(parameter)
2. 최종적으로 AlexNet은 6,000만개의 파라미터가 있기 때문
3. LeNet보다 우수한 아키텍처 장점은 **가치 활성화 기능**을 사용하는 것

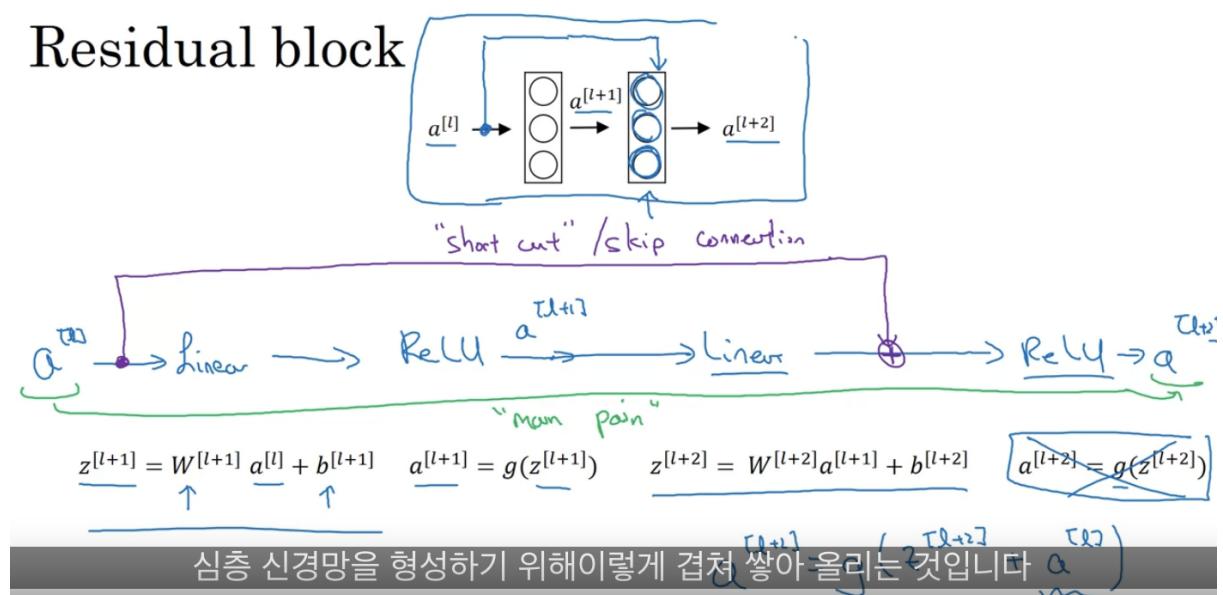


1. 사전에 알고 있으면 좋은 것은 VGG-16은 많은 하이퍼 파라미터를 가지고 있는 대신에 더 간단한 네트워크를 사용해서 conv-layers에만 집중했다.
2. 단지 하나의 스트라이드와 언제나 동일한 패딩의 3×3 필터인 Conv를 사용하는 것에 초점을 맞추고 그리고 스트라이드 2를 가진 2×2 , max pooling 레이어를 만든다는 점이다.
3. 그래서 VGG-16의 장점은 신경망 아키텍처를 단순화
 - 위의 예제와 같이 항상 $\text{Conv} = 3 \times 3$ filter, $s = 1$ 로 동일하다
 - pool도 적용해나가면서 최종적으로 얻는 레이어의 수는 16 layers이다. 그래서 VGG - 16
 - 해당 네트워크는 약 1억 3천 8백만개의 parameter 존재
 - 이렇게 parameter가 많지만 아키텍처의 단순성이 매우 매력적이다

Residual Networks(ResNets)

Residual block

Residual block

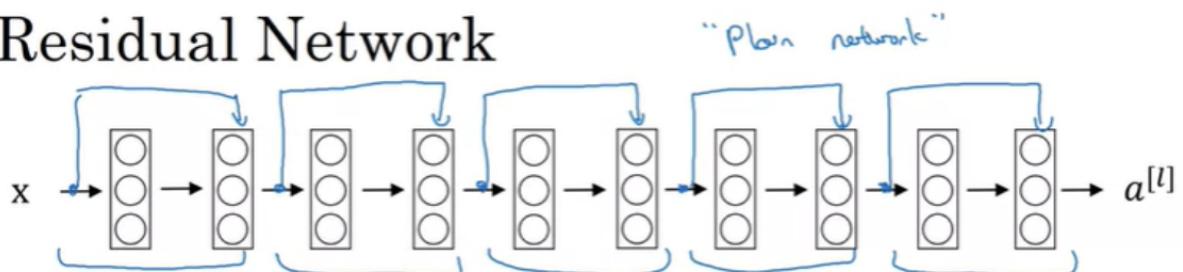


- a^l 이 2개의 layer를 초록색의 main path로 지나게 되면 a^{l+2} 를 얻을 수 있다
- Residual Block에서는 입력인 a^l 를 바로 다음 layer를 뛰어넘고, 그 다음 layer(ReLU를 적용시키기 전)로 연결시켜준다. 이 path를 short cut이라고 부르며, a^l 의 정보가 short cut을 따라 더 깊은 layer로 바로 전달된다.
- 따라서 마지막 값은 $a^{l+2} = g(z^{l+2} + a^l)$ 가 된다.
- 즉, a^l 이 layer를 하나 혹은 두 개씩 건너뛰는 것을 의미한다

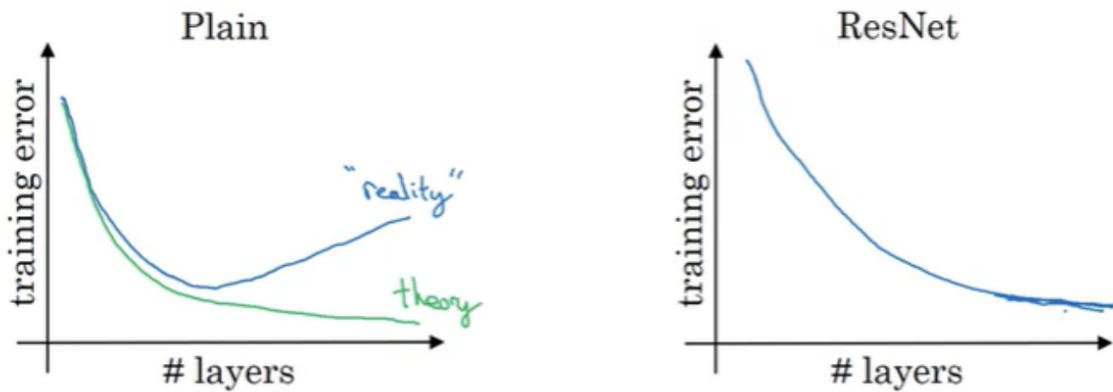
→ ResNet의 구성하는 방법은 이런 Residual Block들을 많이 사용해서 Deep NN을 구성한다.

- 논문에서 사용되는 용어로 short cut이 없는 일반적인 NN을 Plain Network라고 한다

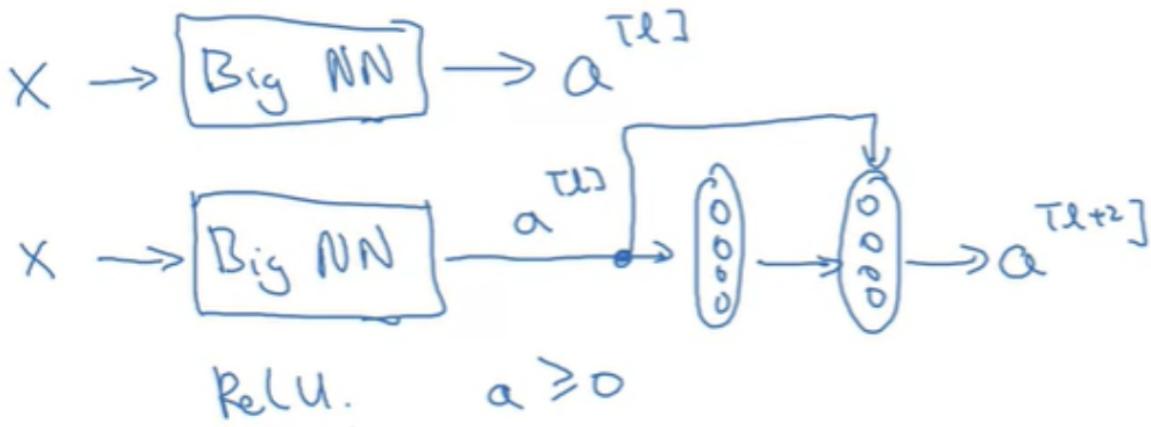
Residual Network



- ResNets은 위와 같이 Residual Block으로 이루어진 네트워크



- 만약 Residual Block, shortcut없이 학습을 수행한다면, 아래 왼쪽 그래프처럼 네트워크가 더 깊어질수록 성능이 안 좋아지는 것을 볼 수 있다. 하지만, ResNet을 사용한다면 Deep network를 학습하는 데 효과적이다.
- 즉, layer가 많아져도 train error를 저하시킬 수 있다는 것을 보여줌
- ResNet을 사용하면 중간에 있는 activation이 더 깊이 전달되어서 vanishing / exploding gradient 문제를 해결 가능



- 그럼 왜 ResNet이 Deep Network에서 잘 동작하는 것일까?
- 위와 같은 구조가 있고, 네트워크 전체에 걸쳐서 activation function으로 ReLU를 사용한다고 가정한다면, 모든 activation은 0보다 크거나 같을 것이다.
- 그리고 skip connection의 short cut에 의해서 다음과 같이 된다

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

- 만약 L2 Regularization을 사용한다면 $W^{[l+2]}$ 가 줄어드는 경향이 있을 것
- bias에도 적용하고 있다면 $b^{[l+2]}$ 도 줄어들 수도 있다
- 극단적으로 $W^{[l+2]} = 0, b^{[l+2]} = 0$ 이라고 가정해보자

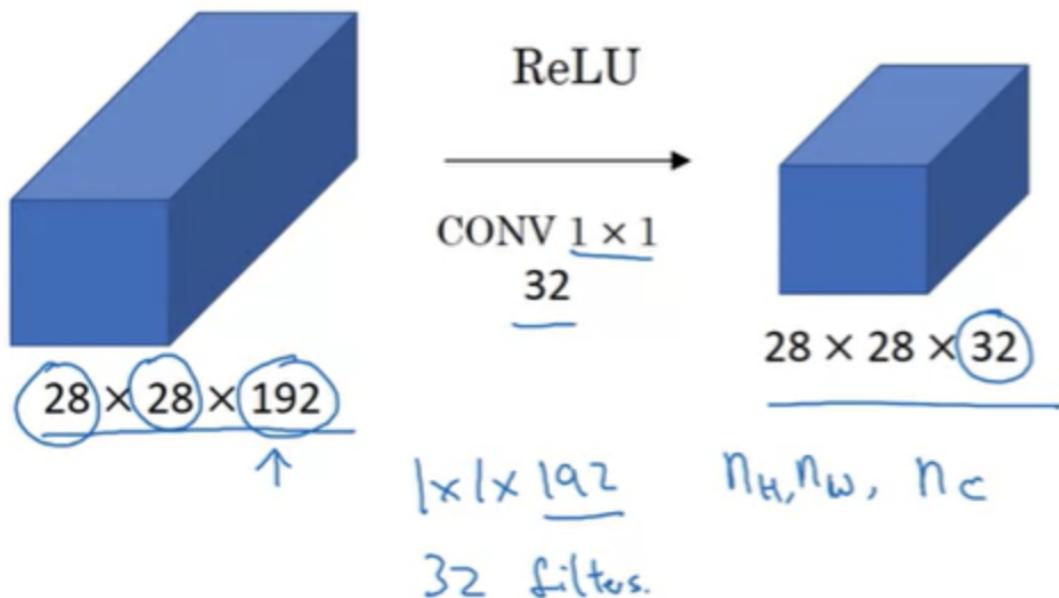
$$\begin{aligned} a^{[l+2]} &= g(\underline{z^{[l+2]} + a^{[l]}}) \\ &= g(\cancel{w^{[l+2]} \cancel{a^{[l+1]}}} + \cancel{b^{[l+2]}} + a^{[l]}) = g(a^{[l]}) \\ &\quad \text{IF } w^{[l+2]} = 0, b^{[l+2]} = 0 \qquad \qquad \qquad = \underline{a^{[l]}} \end{aligned}$$

- 결국 $a^{[l]}$ 만 남게 된다. \rightarrow ReLU이기 때문!
- 따라서 $a^{[l+2]} = a^{[l]}$ 이라는 것을 보여주고 있고, 또한 identity function(항등 함수)을 학습하는 것은 쉽기 때문에 신경망의 성능을 저하시키지 않는다는 것을 보여준다. 만약 residual block이 없는 deep network에서는 Identity function을 위한 파라미터 선택(학습)이 어렵고, 성능이 떨어짐
- 주의할 점
 - $z^{[l+2]}$ 와 $a^{[l]}$ 이 같은 차원을 가져야한다는 것!(앞에서 RGB를 설명할 때 맞춰주듯이!)
 - 차원을 맞춰줘야지 shortcut을 적용할 수 있다
 - 만약 다른 차원이라면 추가적으로 matrix를 사용해서 차원을 동일하게 만들어 줄 수 있다
 - 참고로 W_s 는 파라미터일 수도 있고, 고정된 matrix일 수도 있다.

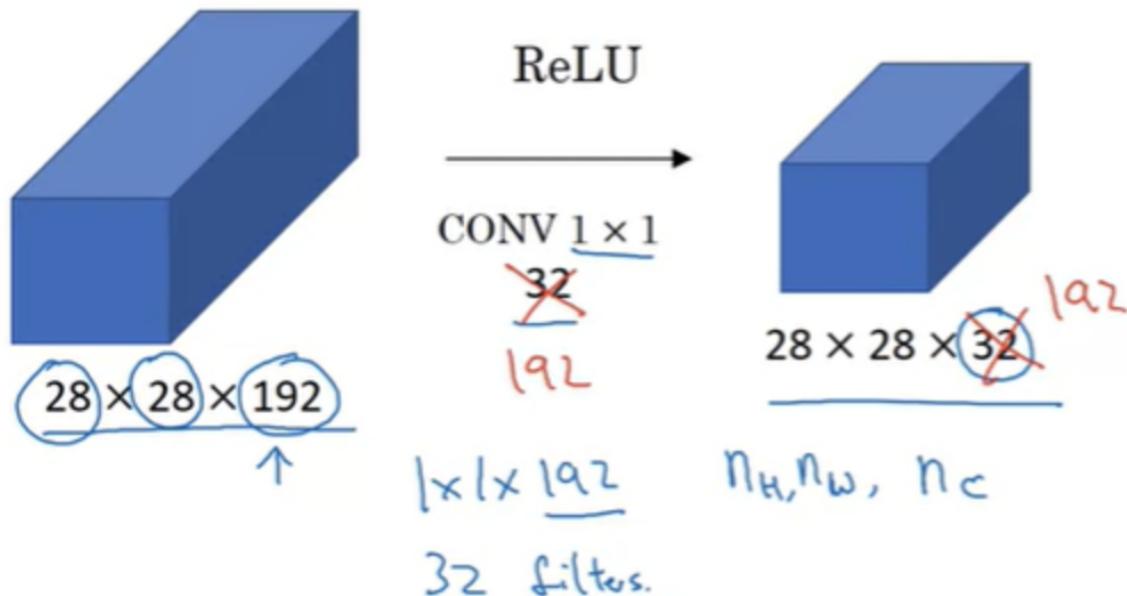
$$a^{[l+2]} = g(z^{[l+2]} + W_s a^{[l]})$$

Network in Network and 1 x 1 convolutions

Using 1×1 convolutions



만약 채널의 수를 192로 유지하고자 한다면, 아래와 같이 filter수를 설정하면 된다.



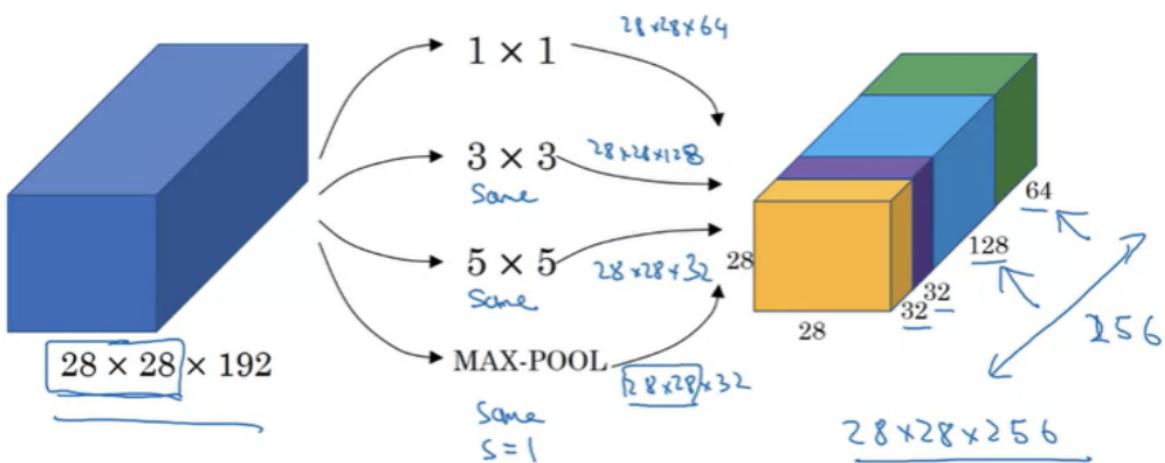
- 왜 1×1 convolution을 사용하는 것일까?
 - Channel 수 조절
 - 연산량 감소

- 비선형성

- 만약 $28 \times 28 \times 192$ 이 있을 때, 높이와 넓이를 줄이고 싶다면 Pooling layer를 사용할 수 있다
- 채널이 너무 많다면 위의 예제에서는 32개의 1×1 filter를 사용하면 된다.
- 2번째 예제에서 filter를 192개로 설정하였는 데 이때는 192 채널로 유지하고 싶을 경우 192 filter를 적용한다.

Inception Network Motivation

Motivation for inception network

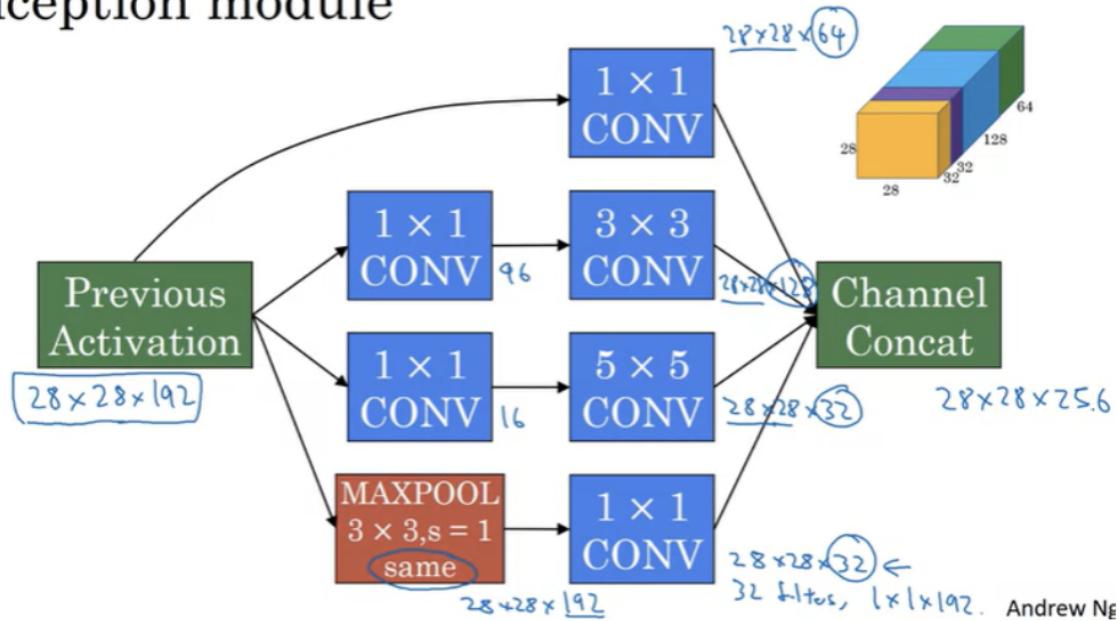


- Inception Module이며, feature들을 효과적으로 추출하기 위해서 Conv(1×1 , 3×3 , 5×5), Max-pooling(3×3)을 각각 수행해서 결과물을 쌓아올린다.
 - $28 \times 28 \times 192 \rightarrow 28 \times 28 \times 256$ 이 된다
- 해당 예제에 관한 기본 아이디어는 네트워크가 원하는 파라미터가 무엇이든, 원하는 filter 사이즈 조합이 무엇이든, 이것들을 모두 다 학습하게 하는 것!
- 하지만 문제점이 존재 → 연산량이 너무 많다 !
- 예를 들어, 5×5 conv에 대해서 연산량 확인
 - $28 \times 28 \times 192$ 에 5×5 conv 산을 수행 $\rightarrow 28 \times 28 \times 32 \times 5 \times 5 \times 192 = 1\text{억 } 2\text{천만번 연산 수행}$

- 결국 5×5 대신에 1×1 conv를 수행하면 연산량을 확실히 줄일 수 있다

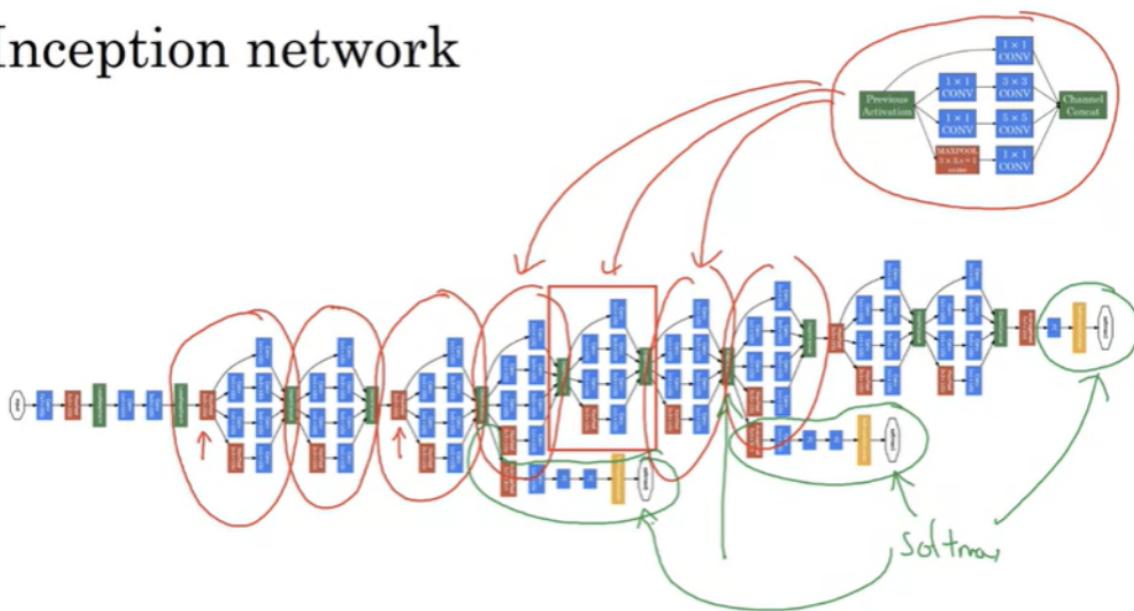
Inception Network

Inception module



- 위에서는 기본적인 Inception Building Block을 살펴보았고, 이 block들을 결합해서 Inception Network를 구현
- 순서
 - $\text{input}(28 \times 28 \times 192) \rightarrow 1 \times 1 \text{ conv} / 1 \times 1 \text{ conv} \rightarrow 3 \times 3 \text{ conv} / 1 \times 1 \text{ conv} \rightarrow 5 \times 5 \text{ conv} / \text{Max pooling} \rightarrow 1 \times 1 \text{ conv}$
 - 각각 거치도록해서 나온 결과를 합쳐서 $28 \times 28 \times 256$ 의 Output을 얻는다
 - Maxpool layer 뒤에 1×1 Conv layer가 오는 것 주의!(max pool을 통해서는 channel 수를 감소시킬 수 없어서 1×1 conv 사용해서 channel 수 줄여줌)

Inception network



- 위의 그림은 각각의 요소마다 들어가 있다
- 논문에는 초록색으로 표시된 모델 output layer 말고, 보조로 사용되는 softmax layer 가 존재
 - 파라미터가 잘 업데이트되도록 도와주며, Output의 성능이 나쁘지 않게 도와준다
 - 정규화 효과
 - 과적합 방지

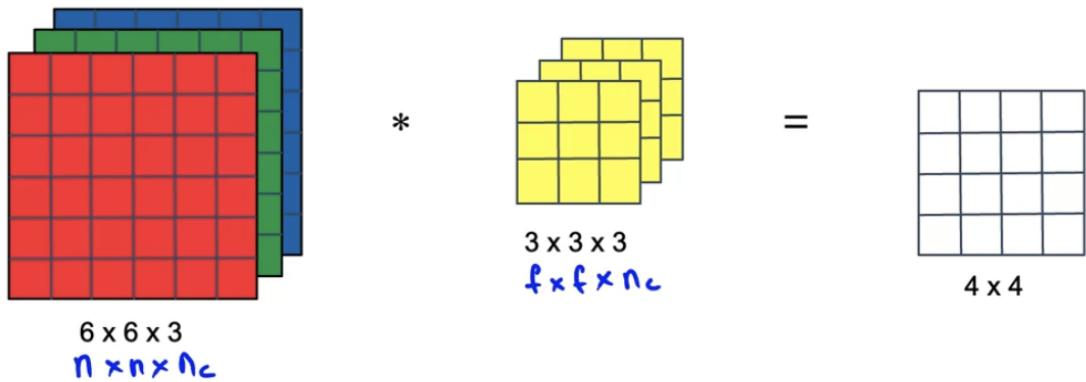
MobileNet

- 컴퓨터 비전에서 사용되는 또 다른 기초 컨볼루션신경망 아키텍처 : MobileNet

Motivation for MobileNets

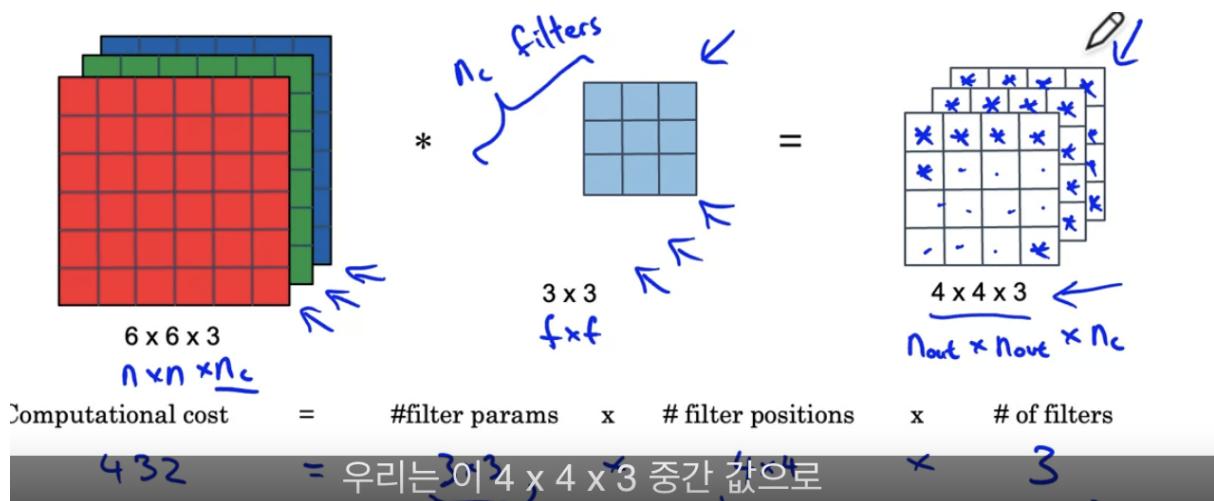
- 기존에 배운 신경망은 상당히 많은 비용이 듈다
 - CPU, GPU ...
- MobileNet은 저비용!

Normal Convolution



- 이전에 배운 구조와 유사
- 계산량을 줄여볼까?

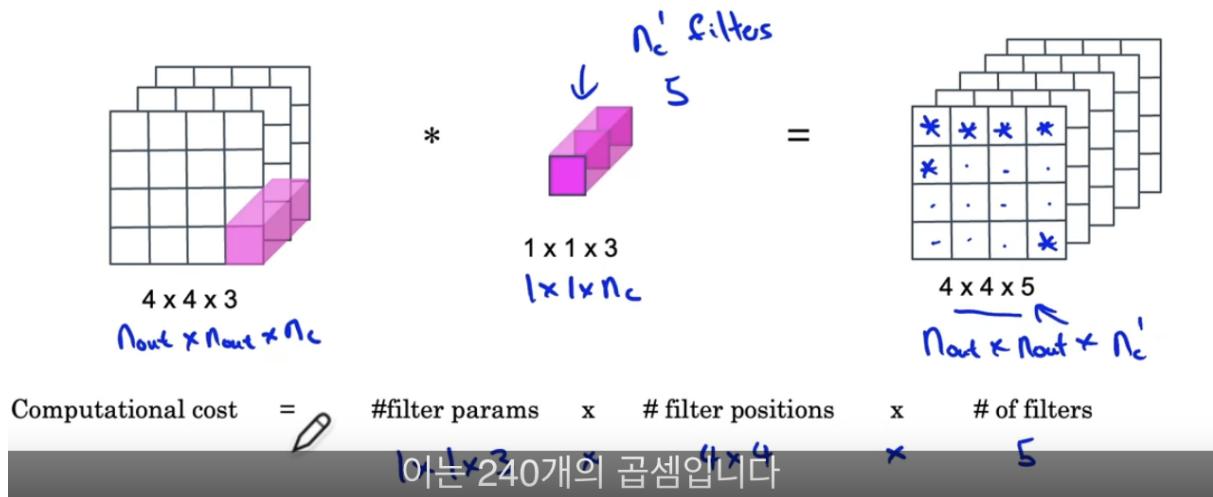
Depthwise Separable Convolution



- 실행 순서는 다음과 같다
 1. filter(3×3)에 초점을 빨간색 Input 채널에 사용하는 것을 집중
 - a. 계속 도장을 찍어 Output 첫 번째 채널에 값을 채움
 2. 빨간색이 끝나면 초록색 Input 채널 수행
 3. 초록색이 끝나면 파란색 Input 채널 수행
- Computational cost = #filter params \times #filter positions \times #of filters
- 총 계산 비용은 3×3 에 필터 위치의 수(해당 예시에서는 4×4)를 곱하고 filter 수 곱함
- 결국 $3 \times 3 \times 4 \times 4 \times 3 = 432$

- 여기서 끝나지 않고 한 가지 단계를 더 수행해야한다!
 - Pointwise Convolution

Pointwise Convolution



- Depthwise separable convolution 결과물을 이용해서 진행
- 분홍색 필터는 $1 \times 1 \times n_c$ 로 계산
 - 이렇게 진행하면 Pointwise convolution의 결과물은 4×4 가 나온다(channel : 1)
- 하지만 위의 예시는 분홍색 filter가 5개 이므로 Pointwise Convolution의 결과물 또 한 5 channels이다!
- Computational cost = #filter params \times #filter positions \times #of filters
 - $(1 \times 1 \times 3) \times (4 \times 4) \times 5 = 240$

Cost Summary

- 결국 계산량은 많은 차이를 보인다
- cost of normal convolution : 2,160
- cost of depthwise separable convolution : $432 + 240 = 672$
- 성능 비교

$$= \frac{1}{n_c} + \frac{1}{f^2}$$

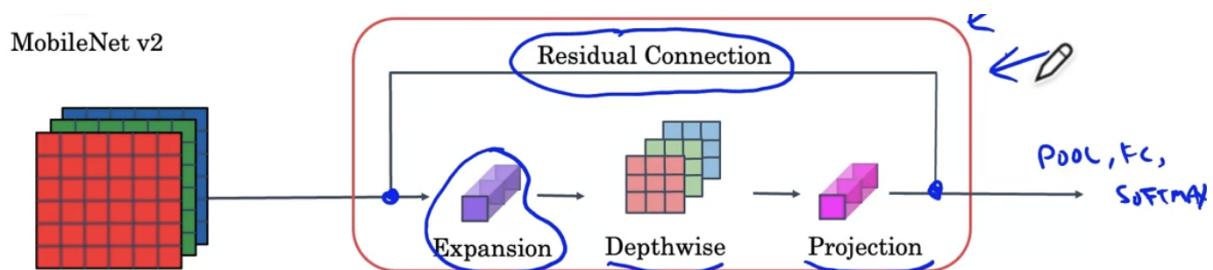
$$\frac{1}{s} + \frac{1}{q}$$

$$= \frac{1}{512} + \frac{1}{3^2}$$

- 해당 수식을 가지고 비교할 수 있는 약 10배 정도 비용이 저렴하다는 것을 알 수 있다!

MobileNet v2

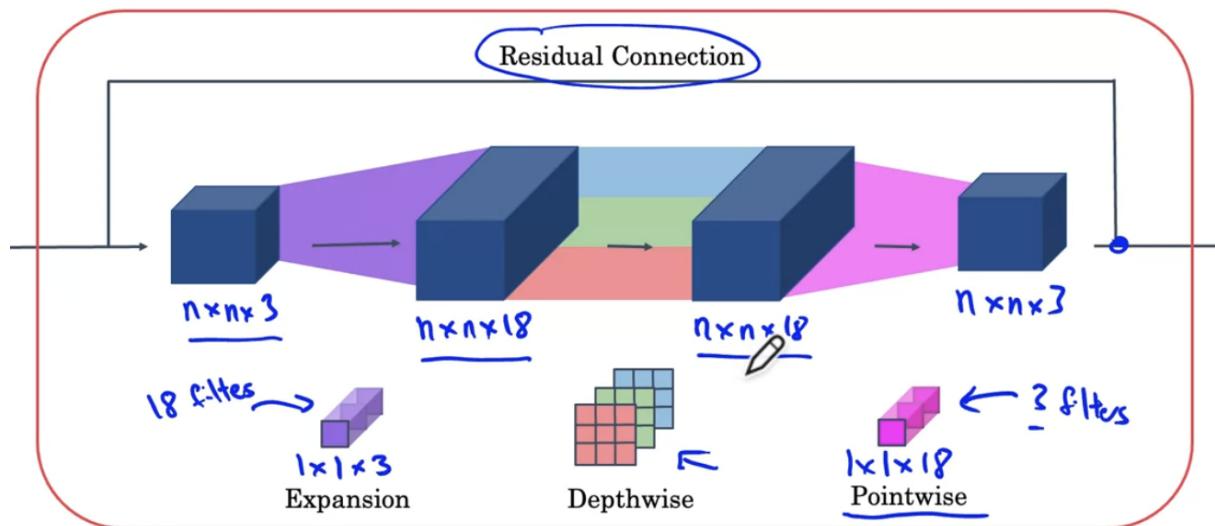
- 위에서 언급한 depthwise convolution, pointwise convolution을 사용해서 진행하는 것은 MobileNet v1



- 더 발전한 MobileNet v2의 주요 변경사항은 2가지
 - Residual Connection 추가
 - 이전 레이어에서 입력을 가져와서 이를 합산하거나 다음 레이어로 직접 전달하여 구성 요소가 더 효율적으로 뒤로 전파되도록 한다
 - 확장 레이어로서도 사용할 수 있다

- 참고로 pool, fc, softmax를 마지막에 적용했고 softmax는 클래스 분류!
- 빨간색 박스는 병목 현상 블록이라고 한다!

MobileNet v2 Bottleneck



Expansion

- $1 \times 1 \times n_c$ 를 적용하는 것을 의미
- filter의 개수를 많이 가져갈수록 채널 증가 (expansion)

Depthwise

- 앞에서 설명과 동일

Pointwise(Projection)

- $1 \times 1 \times 18$ 행렬에서 filter의 개수로 줄일 수 있음
- 그럼 왜 병목 블록(bottleneck)을 사용하는 것일까?
 - Expansion을 통해서 신경망이 더 풍부한 기능을 배울 수 있게 해줌 → 많은 계산량 존재
 - 모델명인 것처럼 Mobile같은 경우 무거운 메모리 제약이 존재하기 때문에 Pointwise conv 혹은 더 작은 값 집합으로 다시 투영하는 작업을 통해 값을 저장하는 데 필요한 메모리 줄일 수 있음

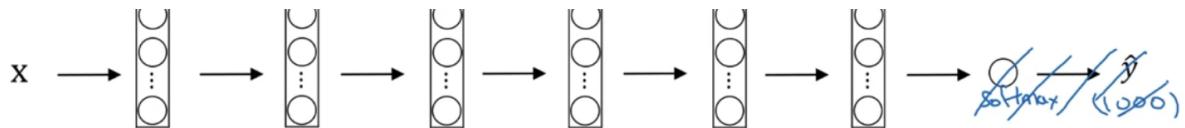
- V1 보다 좋은 점은?
 - 적은 양의 계산 및 메모리로 좋은 성능

EfficientNet

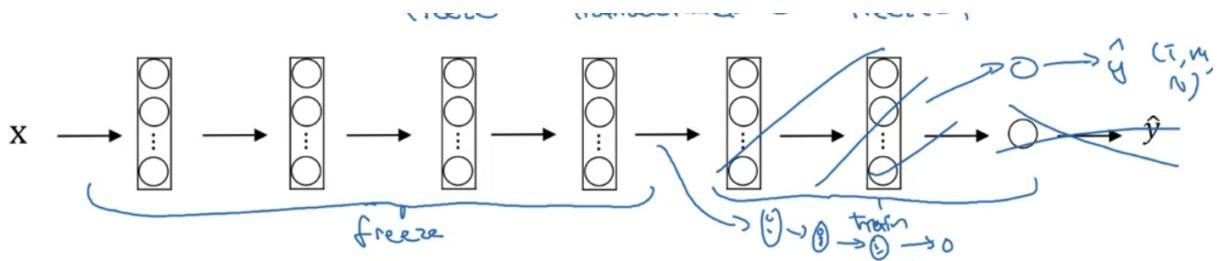
- 특정 장치의 신경망을 자동으로 확장하거나 축소할 수 있는 방법
 - Higher Resolution : 고해상도 이미지 사용
 - Deeper : 신경망을 깊게
 - Wider : 레이어의 너비를 넓게
- 3개를 합한 것을 Compound Scaling이라고 하는데 3개의 비율은 EfficientNet 라이브러리 찾아보기

Transfer Learning

- 직접 신경망 구축이 아닌! 다른 사람이 오픈 소스로 업로드한 것을 사용
- 다운받는 파일은 코드뿐만 아니라 가중치도 다운로드 가능



- 기존 코드를 보면 softmax 레이어가 존재할 텐데 이것을 없애고, 우리가 구하고자 하는 softmax unit(class)을 생성
 - 다시말해서 기존에 코드는 다른 것의 특화되었기 때문에 우리의 프로젝트에 맞게 수행하기 위해서 softmax 레이어를 수정하는 것!
 - 이때 일단 파라미터(앞에 존재)는 고정하고 softmax 레이어와 관련된 파라미터를 조정하자
- softmax를 조정했다면 점차 뒤에 있는 레이어부터 freeze를 풀면서 진행 (파라미터 수정한다는 뜻 : train)
- 아니면 마지막 몇개의 레이어를 제거하고 새로운 은닉 유닛과 최종 softmax 출력으로 사용 가능



- 많은 양의 데이터를 가지고 있다면 네트워크 전체를 초기화하고 훈련시킬 수 있음

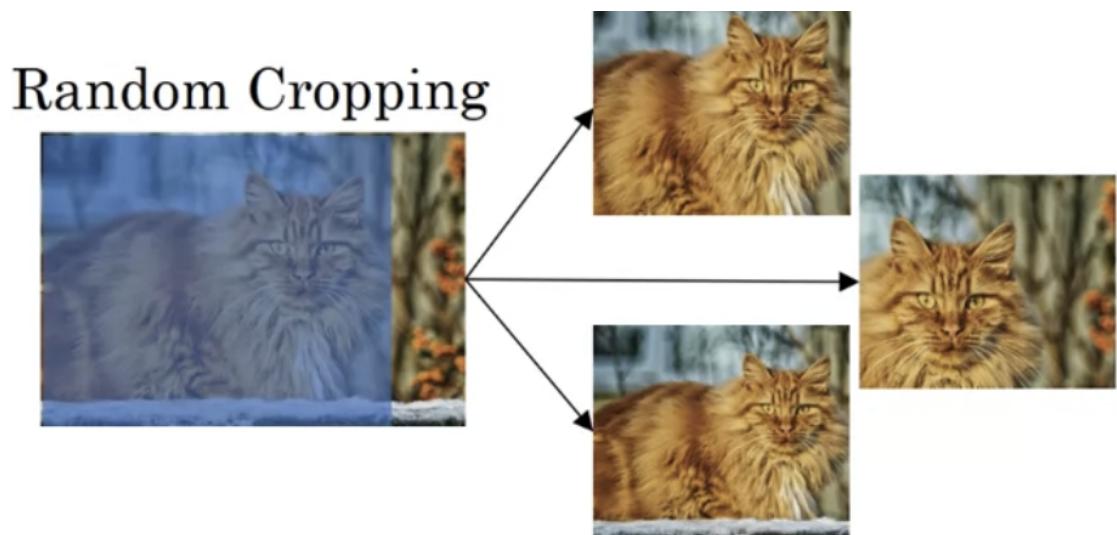
Data augmentation

변형

1. 수직축 미러링

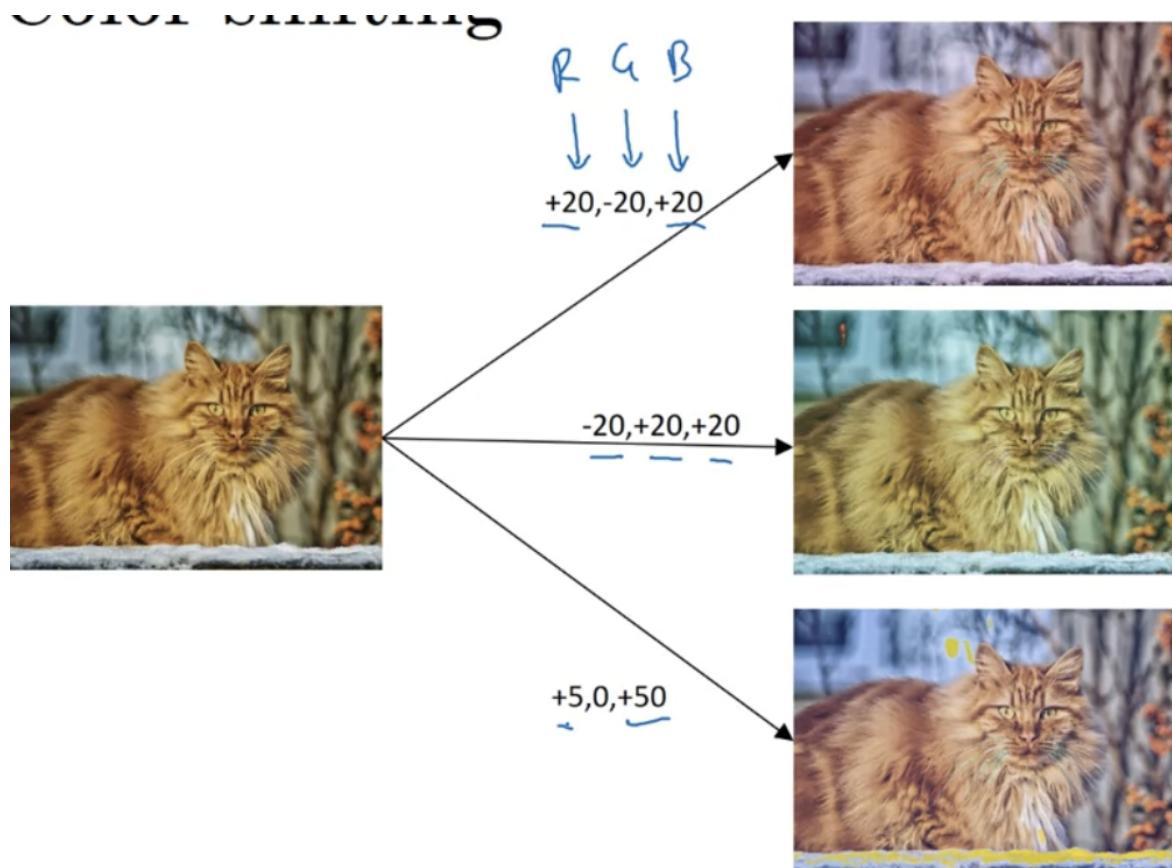


2. 랜덤 자르기(Random Cropping)



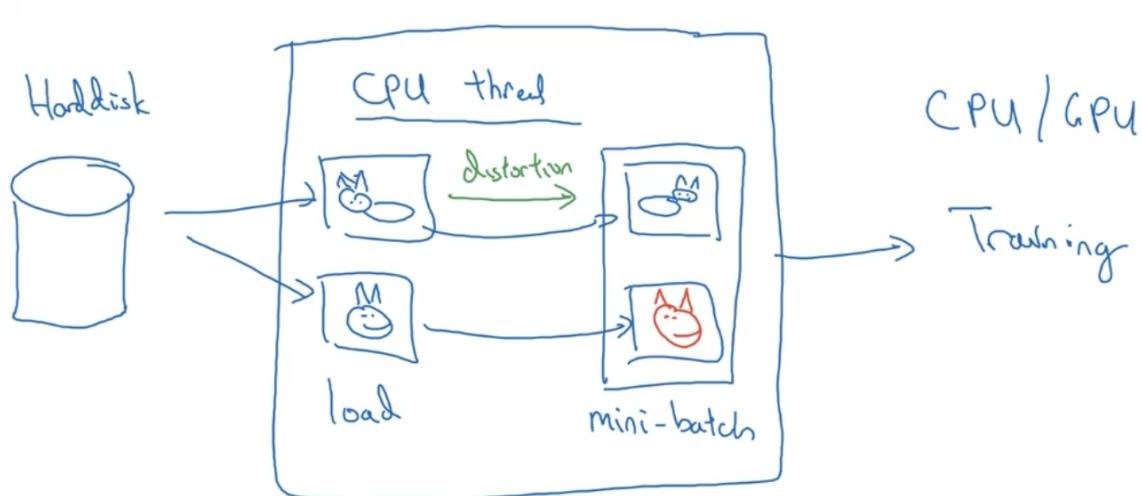
3. Rotation ...

Color shifting



- RGB 색상 변화

Implementing distortions during training



- 하드디스크에 있는 이미지를 CPU 쓰레드를 통해서 변형 진행
- cpu thread 하나가 변형을 진행하는데 이때 병렬적으로 처리
- cpu / gpu 통해서 최종적으로 training