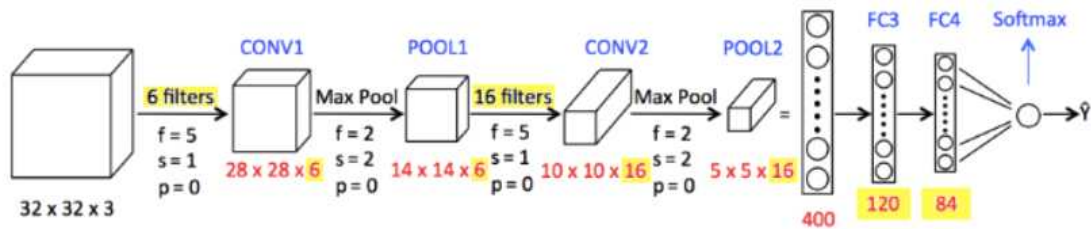


Deep Convolutional Models: Case Studies

2-1 Why look at case studies?



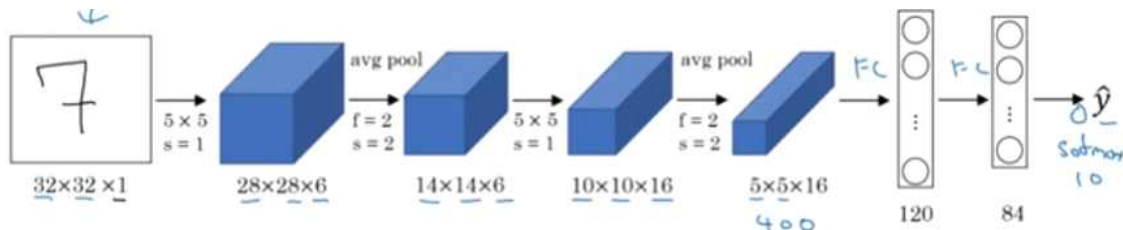
- 컴퓨터 비전 분야의 연구대상: 합성곱 신경망의 기본 구성 요소인 합성곱층, 풀링층, 완전연결층을 조합하여 효과적인 합성곱 신경망을 만드는 것
- Classic networks: LeNet-5, AlexNet, VGG
- ResNet: 신경망의 깊이가 깊어질수록 152개의 층을 훈련시킴
- Inception

요약) CNN의 기본 구성 요소: 합성곱층(특성 추출), 풀링층(특성 압축), 완전 연결층(분류/예측)

- **합성곱층**: 필터를 사용해서 입력 이미지에서 특성 추출 (필터: 입력 데이터의 작은 영역을 스캔 하면서 중요한 패턴을 감지하고, 이를 출력으로 전달함. 이 과정에서 이미지의 공간적 특성을 보존하면서도 중요한 정보를 추출 가능)
- **풀링층**: 합성곱층에서 추출한 특성의 크기를 줄이면서 요약함: Max or Average Pooling
- 풀링층의 주요 목적: 계산량을 줄이고 모델이 과적합되지 않도록 정보를 압축하는 것. 크기를 줄이면서도 중요한 정보는 유지하는 특성이 있음
- **완전 연결층**: 출력을 1차원 벡터로 펼친 후, 전통적인 신경망처럼 모든 노드가 연결되는 층. 이 단계에서 최종적으로 이미지에 대한 분류나 예측 작업이 수행됨

2-2 Classic Network

<LeNet-5>



LeNet-5의 목적: 손글씨의 숫자를 인식하는 것

32 x 32 x 1의 입력 이미지 (LeNet-5는 그레이스케일의 이미지에 훈련되었기 때문)

5 x 5 필터 6개와 stride=1을 사용 -> 28 x 28 x 6

2 x 2 필터(f=2)와 stride=2를 이용한 average pooling -> 14 x 14 x 6

5 x 5 필터 16개와 stride=1을 사용 -> 10 x 10 x 16

2 x 2 필터와 stride=2를 이용한 average pooling -> 5 x 5 x 16 => 400

400개의 노드를 120개의 뉴런에 각각 연결해줘서 완전연결층을 만들

또 다른 84개의 FC(완전연결층)

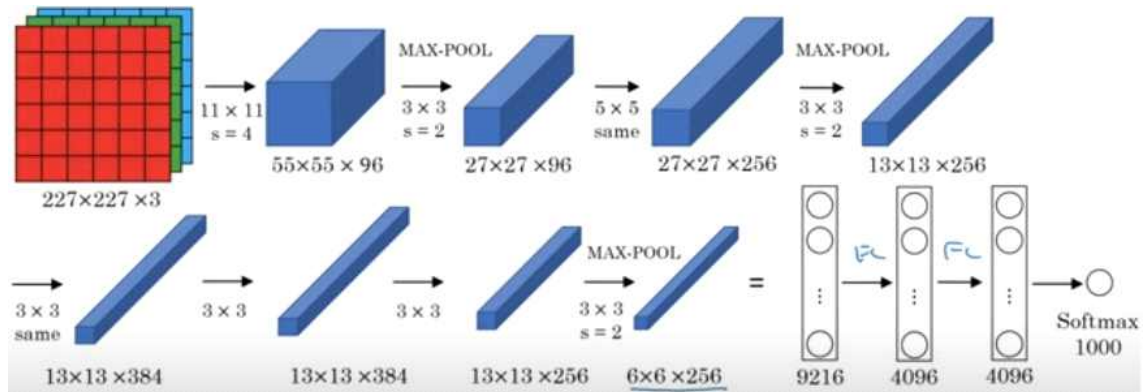
비선형성 함수를 사용해 y의 예측값 - 0~9까지 10가지 가능성의 숫자 인식

이 신경망은 60000개의 변수를 가짐

LeNet-5 특징

- 신경망의 층이 깊어질수록 높이와 너비 감소, 채널의 수 증가
- 합성곱층 -> 풀링층 -> 합성곱층 -> 풀링층 -> 완전연결층 -> 출력
- 당시에는 비선형성 함수로 Relu를 사용하지 않고, tanh와 sigmoid 사용

<AlexNet>



227 x 227 x 3의 입력 이미지 크기

11 x 11 필터 96개와 stride=4을 사용 -> 55 x 55 x 96

3 x 3 필터와 stride=2를 이용한 max pooling -> 27 x 27 x 96

5 x 5의 동일 합성곱 연산 실행 -> 27 x 27 x 128

3 x 3 필터와 stride=2를 이용한 max pooling -> 13 x 13 x 128

3 x 3의 동일 합성곱 연산 실행 -> 13 x 13 x 384

3 x 3의 동일 합성곱 연산 실행 -> 13 x 13 x 384

3 x 3의 동일 합성곱 연산 실행 -> 13 x 13 x 256

3 x 3 필터와 stride=2를 이용한 max pooling -> 6 x 6 x 256 => 9216

9216개의 노드를 FC(완전 연결층)에 연결 -> 4096개의 FC -> 4096개의 FC

softmax를 사용해 1000개의 예측값 출력

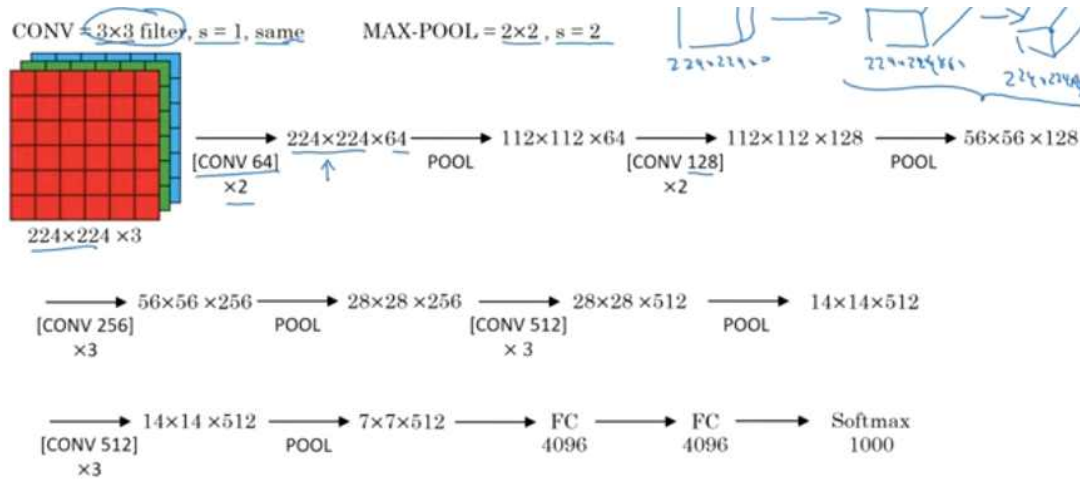
* 동일합성곱: 입력 데이터의 공간적 차원을 유지하기 위해 패딩을 사용하는 합성곱

AlexNet 특징

- LeNet과 매우 유사하지만 훨씬 큰 크기를 가짐 (LeNet 매개변수: 6만 개 / AlexNet 매개변수: 6천만 개)
- 유사한 구성요소를 가졌지만 더 많은 은닉 유닛(각 층에서 더 많은 노드를 사용해 데이터를 처리하고, 특징을 더 세밀하게 추출한다는 뜻)과 더 많은 데이터를 통해 훈련하기 때문에 훨씬 더 뛰어난 성능을 보임
- Relu 활성화 함수를 사용한다는 점이 LeNet과 구별됨
- +) 당시에는 GPU가 느려서, 2개의 GPU를 훈련하는 방법을 가짐. 2개의 GPU에 여러 층들이 나눠서 두 GPU가 소통하는 방식

<VGG-16>

- 주목할 만한 점) 많은 하이퍼파라미터를 가지는 대신 합성곱에서 스트라이드가 1인 3x3의 필터만을 사용해 동일합성곱을 하고, 최대 풀링층에서는 2의 스트라이드의 2x2를 사용함



224 x 224 x 3 입력 이미지

- 첫 두층에서는 64개의 필터로 **동일 합성곱** 진행 → 224 x 224 x 64 ([CONV 64] X 2])
- 앞에서 말했듯이, 합성곱에서 모든 필터는 3x3의 크기를 가지고, 스트라이드 1, 동일합성곱 사용
- 동일 합성곱: 합성곱 연산을 한 후에도 출력의 크기가 입력 크기와 동일하게 유지된다는 의미. 즉, 입력 이미지가 224 x 224 크기면, 출력 이미지도 224 x 224 크기를 유지
- 첫 두 층에서는 64개의 필터를 사용하여 상대적으로 단순한 특징을 추출 → 네트워크가 더 깊어지면, 필터의 개수를 128개, 256개, 512개 등으로 점차 늘려가며 더 복잡한 특징을 학습
- 2 x 2 필터와 stride=2을 이용한 max **pooling** → 112 x 112 x 64
- 128개의 필터를 가진 2개의 합성곱층에 **동일 합성곱** 진행 → 112 x 112 x 128
- 2 x 2 필터와 stride=2을 이용한 max **pooling** → 56 x 56 x 128
- 256개의 필터의 3개의 합성곱층에 **동일 합성곱** 진행 → 56 x 56 x 256
- 2 x 2 필터와 stride=2을 이용한 max **pooling** → 28 x 28 x 256
- 512개의 필터의 3개의 합성곱층에 **동일 합성곱** 진행 → 28 x 28 x 512
- 2 x 2 필터와 stride=2을 이용한 max **pooling** → 14 x 14 x 512
- 512개의 필터의 3개의 합성곱층에 **동일 합성곱** 진행 → 14 x 14 x 512
- 2 x 2 필터와 stride=2을 이용한 max **pooling** → 7 x 7 x 512 ⇒ 4096
- 7 x 7 x 512의 FC(완전 연결층)이 되고, 4096개의 유닛과 1000개의 **소프트맥스 출력**이 나옴

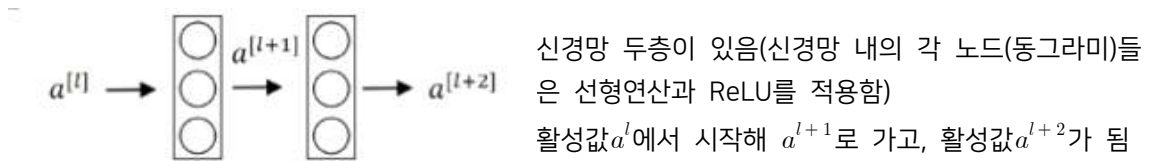
VGG-16 특징

- 많은 하이퍼파라미터를 가지는 대신 합성곱에서 스트라이드가 1인 3x3의 필터만을 사용해 동일 합성곱을 하고, 최대 풀링층에서는 2의 스트라이드의 2x2를 사용함
- 많은 하이퍼파라미터를 가지지만 아주 간결한 구조를 가짐
- VGG-16의 16은 16개의 가중치를 가진 층이 있다는 것을 의미
- 이는 1억 3천 8백만 개 정도의 변수를 가진 큰 네트워크임
- VGG-16의 구조적 장점: 균일하다는 것 - 몇 개의 합성곱층 뒤에, 풀링층이 높고, 너비를 줄여줌 / 신경망의 깊이가 깊어질수록 합성곱층의 필터 수가 매번 두 배씩 늘어남(64-128-256-512).
- 구조의 획일성의 단점: 훈련시킬 변수의 개수가 많아 네트워크의 크기가 커짐

2-3 ResNets

아주 깊은 신경망을 훈련시키기 어려운 이유? 경사가 소실되거나 폭발적으로 증가하는 문제 때문

-> skip connection(short cut)이 문제 해결 가능-> ResNet 형성



- 계산과정: a^l 에 선형연산 적용($z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}$) -> ReLU비선형성을 적용해 a^{l+1} 계산

-> 또 선형연산 적용 -> 또 ReLU 연산 -> a^{l+2}

- a^l 에서 $z^{[l+1]}$ 을 계산하기 위해서는 가중치 행렬을 곱해주고 편향벡터를 더해줌(합성곱 연산 자체가 선형연산임)

- ReLU 비선형성을 적용해 a^{l+1} 계산 : $a^{[l+1]} = g(z^{[l+1]})$ * g()가 ReLU 비선형성 (합성곱 연산 결과에 비선형성을 부여해 합성곱층이 비선형적인 특징을 학습할 수 있도록 함)

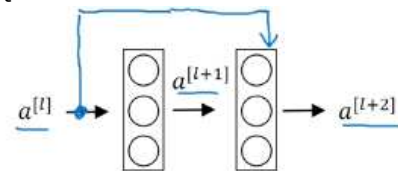
- 또 선형연산 적용: $z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$

- 또 ReLU 연산: $a^{[l+2]} = g(z^{[l+2]})$

- => 이렇듯, a^l 의 정보가 a^{l+2} 로 흐르기 위해서는 위의 모든 과정을 거쳐야 함 = main path

- ResNet에서는 이를 조금 바꿔서, a^l 을 복제해 신경망의 더 먼 곳까지 한 번에 가도록 만든 뒤, ReLU 비선형성을 적용해주기 전에 a^l 을 더해줌 = short cut

- main path를 따르는 대신 short cut을 따라 a^l 은 신경망의 더 깊은 곳으로 갈 수 있음 (a^l 은 선형연산 뒤, ReLU연산 전에 들어감)

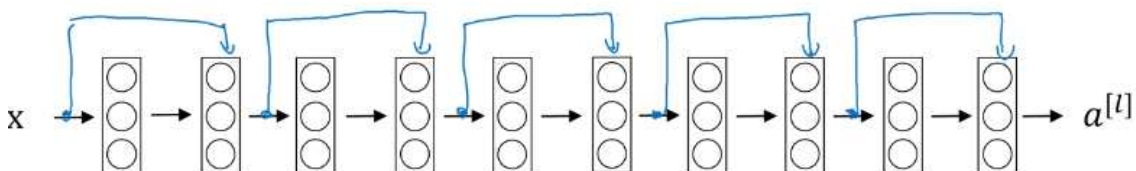


- $a^{l+2} = g(z^{[l+2]} + a^l)$ 이 최종식 <- a^l 이 residual block

- short cut(skip connection) = a^l 이 정보를 전달하기 위해 층을 뛰어넘는 것(신경망의 더 깊은 곳으로)을 의미함 -> 이를 통해 훨씬 깊은 신경망을 훈련시킬 수 있게 함 (층이 깊어져도 훈련 오류가 계속 감소하는 성능을 가짐. 경사 소실 문제 해결)

ResNet: Residual block(a^l)이 층을 뛰어넘으면서(skip connection) 깊은 신경망을 효과적으로 학습할 수 있도록 한 네트워크

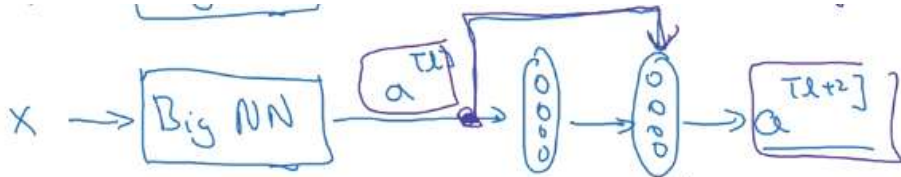
Residual block 사용 시, 훨씬 더 깊은 신경망 훈련 가능 (신경망이 깊어져도 훈련세트에서 오류가 높아지는 것 방지) -> Residual block들을 쌓아서 깊은 신경망 만들



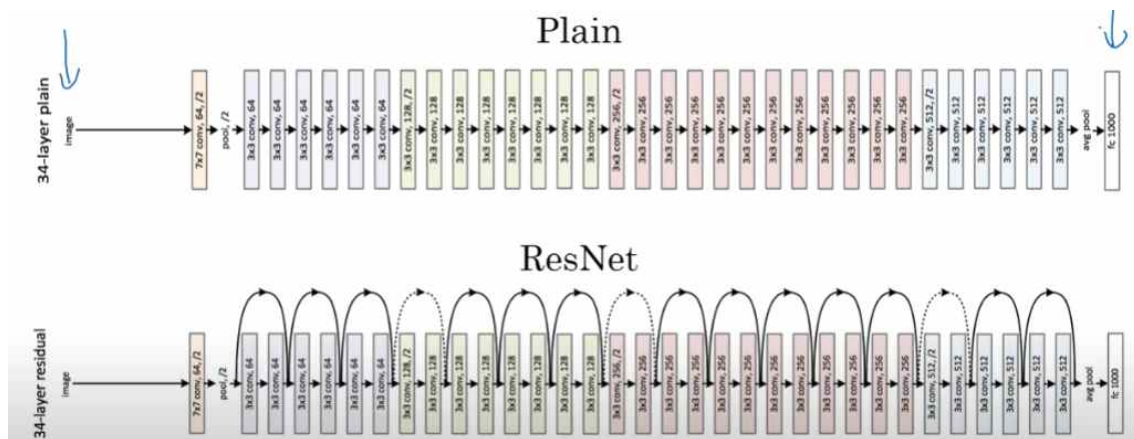
Plain network에 skip connection을 더해 ResNet으로 바꿈-> 5개의 residual block이 합쳐짐

2-4 Why ResNets Work

- $a^{[l+2]} = g(z^{[l+2]} + a^l)$, $a^{l+2} = g(W^{[l+2]}a^{[l+2]} + b^{[l+2]} + a^l) \therefore z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$
- 위에 식에서, 만약 L2 규제나 가중치 붕괴를 적용하면 W와 b값이 감소함
- W와 b이 0이라면 $W^{[l+2]}a^{[l+2]} + b^{[l+2]}$ 이 부분은 사라지고 $a^{l+2} = g(a^l) = a^l$
- $a^{l+2} = g(a^l) = a^l \rightarrow$ 왜냐면, 여기서 ReLu활성화함수를 사용해서 모든 활성화값이 양수임. $g(a^l)$ 은 양수에 ReLu를 적용했기 때문에 a^l 을 돌려받음 (ReLU는 음수를 0으로 만드는 비선형 함수)
- 그래서 항등함수는 residual block의 훈련을 용이하게 만들어줌. (- 신경망의 깊이가 깊어지더라도 입력값이 그대로 다음 층으로 전달되기 때문에 경사 소실 문제 완화 - 학습성능 유지)
- * 항등함수: 입력값을 그대로 출력하는 함수 $f(x)=x$
- $a^l = a^{l+2}$ 인 이유는 스킵연결 때문 \rightarrow 즉, 항등함수를 학습하여 a^{l+2} 에 a^l 을 대입하면 되기 때문에 a^l 과 a^{l+2} 사이에서 두 층을 추가해도, 이 두 층이 없는 네트워크만큼의 성능을 가짐
- \Rightarrow residual block을 거대한 신경망 어딘가에 추가해도 성능에 지장이 없는 이유! - but. 우리의 목표는 성능을 향상시키는 것



- ResNet이 잘 작동하는 주된 이유: 추가된 층이 항등 함수를 학습하기 용이하기 때문 \rightarrow 성능 저하가 없고, 또 경사 하강법을 이용하여 성능을 향상시킬 수도 있음
- $a^{[l+2]} = g(z^{[l+2]} + a^l)$ 식에서 $z^{[l+2]}, a^l$ 이 같은 차원을 가진다고 가정함 \rightarrow ResNet에서 동일 합성곱을 많이 사용 - 스킵연결 전 후의 차원이 같아지게 만들 \rightarrow 두 동일 차원의 벡터 합을 구할 수 있음



Plain을 ResNet으로 바꾸려면 스킵연결을 추가해주어야 함

많은 3x3 합성곱이 있는데, 그 중 대부분은 동일 합성곱, 그래서 같은 차원의 벡터를 더해주는 것 \rightarrow 차원이 유지됨

때때로 있는 풀링층(점선)에서는 차원을 조정해주어야함(W)

2-5 Network In Network

Network In Network: 1 x 1 convolutions. 입력 이미지와 1 x 1 필터와의 합성곱

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

 \times

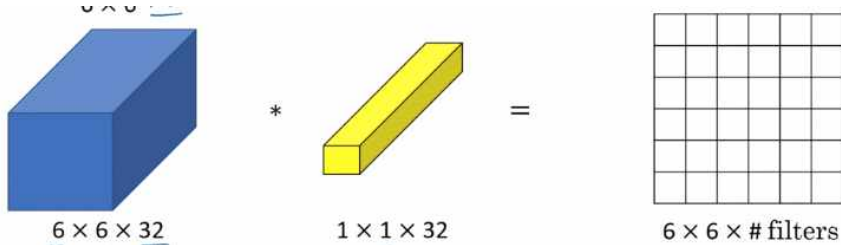
2

 $=$

2	4	6	...		

6×6

- 이미지에 2만큼 곱해주는 셈 - 별로 유용해보이지는 않음

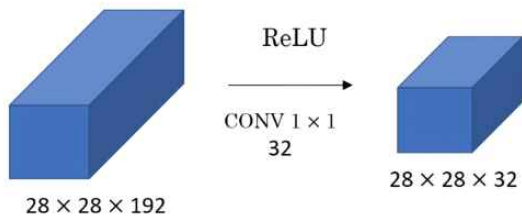


$6 \times 6 \times 32$ $1 \times 1 \times 32$ $6 \times 6 \times \#filters$

- 6x6x32에 1x1필터와 합성곱 하는 것이 훨씬 더 의미있음
- 36개의 위치 각각에서 32개의 숫자를 필터의 32개 숫자와 곱해준 후 ReLU 비선형성을 적용 - 한 지점에서 하나의 숫자 출력

1x1 합성곱이 유용한 경우

<채널의 수를 줄일 수 있다.>



- 높이와 너비를 줄이려면 풀링층을 사용하면 되지만, 채널수를 줄일 때는 1x1합성곱 사용하면 됨
- 여기서는 32개의 1x1x192 필터 사용 (필터와 입력의 채널 수는 일치해야 하기 때문)
- 채널 수를 줄여 네트워크의 계산을 용이하게 함

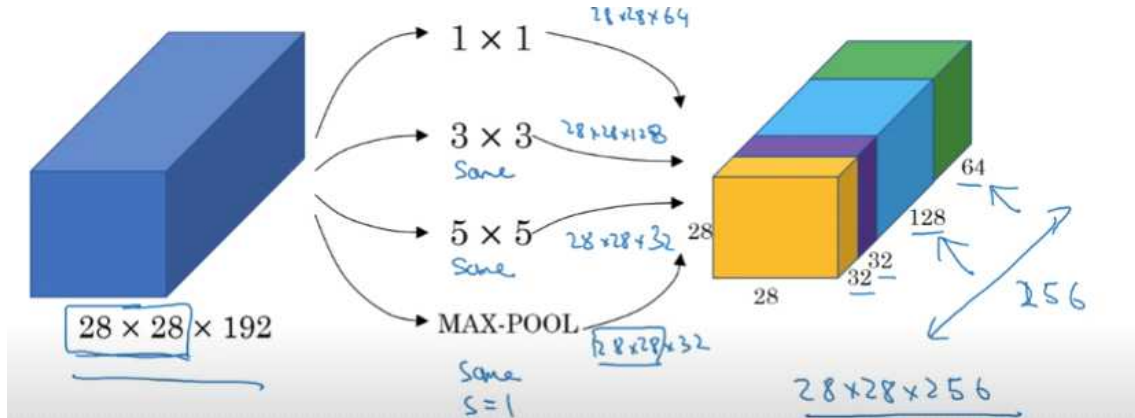
-> 네트워크에 비선형성을 더해주고 채널의 수 조절

- Network In Network 개념은 Inception Network 구축에 유용하게 사용됨

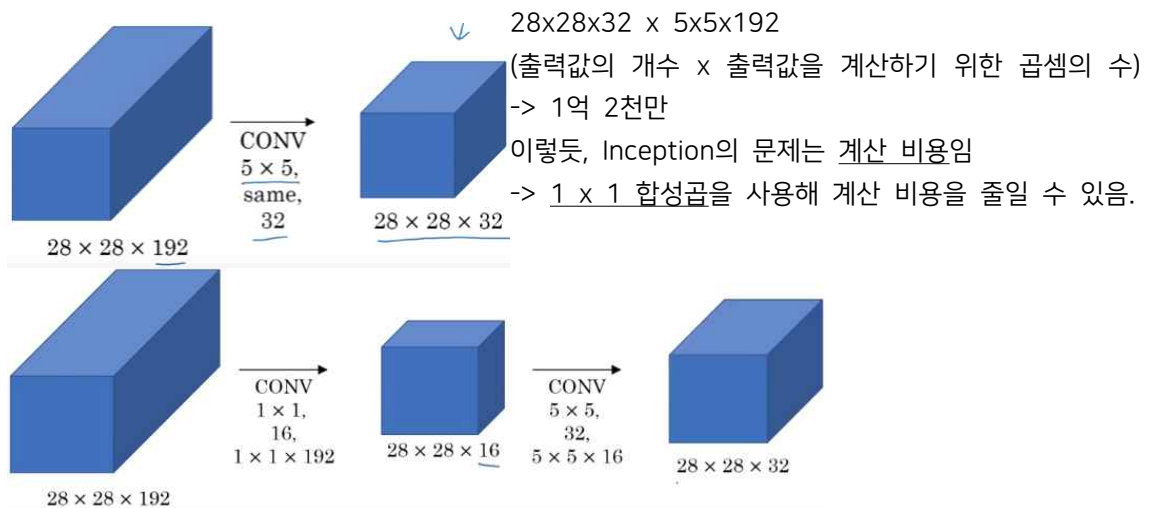
2-6 Inception Network Motivation

Inception Network: 다양한 크기의 필터(합성곱층, 풀링층)를 동시에 사용하는 딥러닝 모델(일반적인 CNN에서는 각 층마다 하나의 필터 크기(예: 3x3 필터)만 사용) - 네트워크가 복잡해지긴 해도 성능은 뛰어남(다양한 스케일의 특징을 한 번에 학습-> 각각의 필터나 풀링 층이 특정 크기에서 효과적인 특징을 추출)

필터의 크기를 정하지 않고 합성곱 또는 풀링층을 모두 사용하고, 출력들을 다 엮어낸 후 네트워크가 스스로 원하는 변수나 필터의 크기의 조합을 학습함



위의 사진의 5x5 필터 부분만 집중해서 보면서 계산비용을 알아보자

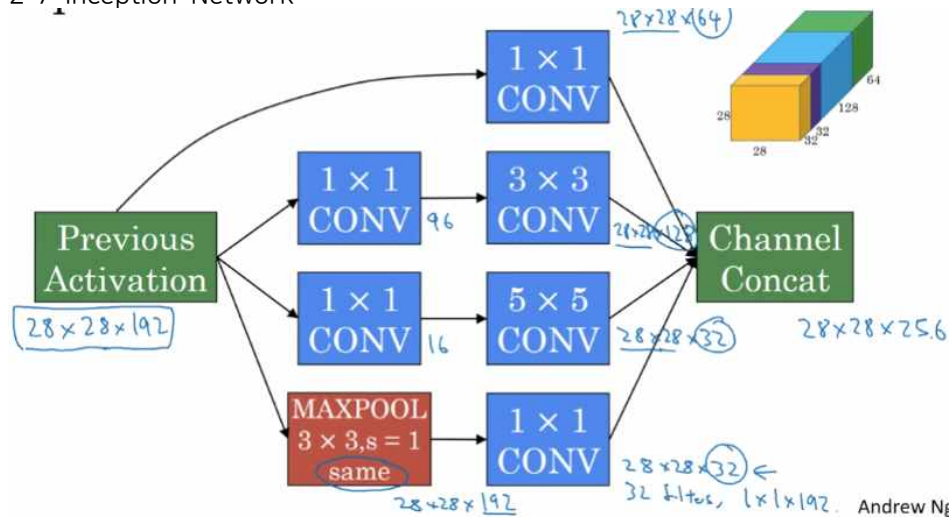


1 x 1 합성곱을 사용해 192개의 채널을 16개로 줄이고 이에 5x5 합성곱을 해 최종출력을 얻음

- 입력과 출력의 크기는 전과 동일
- 가운데 층을 병목층(bottleneck lag; 네트워크에서 작은 부분을 의미)이라 부르며, 크기를 다시 늘리기 전 이미지 줄임
- 첫 번째 합성곱층의 계산비용(병목층)은 $28 \times 28 \times 16 \times 1 \times 1 \times 192 = 240$ 만
- 두 번째 합성곱층의 계산비용은 $28 \times 28 \times 32 \times 5 \times 5 \times 16 = 240$ 만 = 1천만
- 총 필요한 곱셈의 수는 240만과 1천만의 합 = 1천 2백만

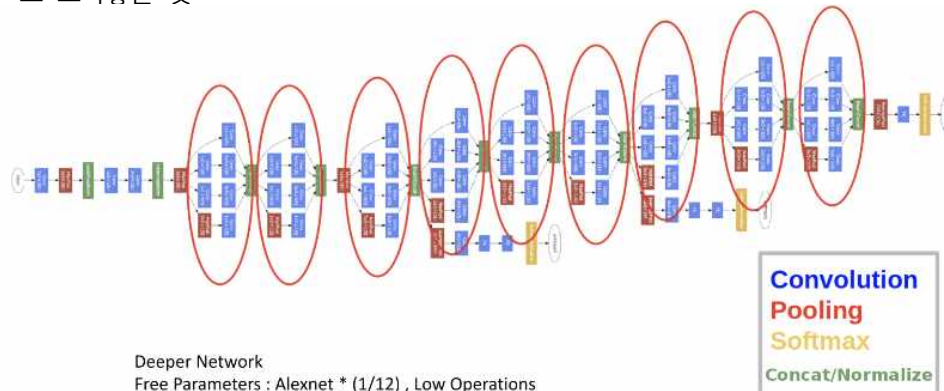
이렇듯 병목층을 적절하게 구현할 수 있다면 성능에 지장 없이 표현 크기를 줄일 수(계산 비용을 줄일 수) 있다.

2-7 Inception Network



Inception module: 여러 크기의 합성곱층(1x1, 3x3, 5x5)과 풀링층(동일 합성곱)을 병렬로 하여 결과를 하나로 합친 구조

- Inception module은 activation 값이나 이전 층의 출력을 입력값으로 받는다.
- 여러 크기의 합성곱(1x1, 3x3, 5x5) 전에 1 x 1 합성곱을 사용해 계산 비용을 줄임
- 풀링층에는 채널 수가 너무 많아 1 x 1 합성곱층을 추가하여 채널 수를 줄임
- channel concat은 모든 블록들을 하나로 연결해주는 것을 의미한다.
- 이것이 하나의 Inception model이고, Inception Network는 이러한 Inception module을 하나로 모아놓은 것



- 네트워크의 마지막 몇 개의 층은 완전 연결층과 예측을 위한 소프트맥스층
- 중간에 튀어 나온 결과지도 은닉층을 가지고, 완전 연결층을 지나 소프트맥스로 예측을 함- 여기서 계산된 특성도 이미지의 결과를 예측하는데 나쁘지 않은 성능-> Inception Network에 정규화 효과를 주고 네트워크의 과대적합을 방지해줌
- Inception Network는 Google의 일원에 의해 개발되어서 GoogLeNet 이라고도 불림
-

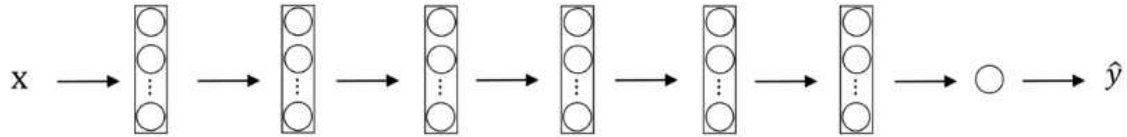
2-8 Using Open Source Implementation

Github로부터 오픈 소스 코드 다운받는 법: google에 [찾고자하는 알고리즘] github 검색 - 코드를 받을 수 있는 URL 복사 - command에 들어가 git clone을 치고 URL 복붙 후 엔터 -> 하드 디스크에 저장됨

- 장점: 네트워크 훈련 시 많은 시간이 소요되는데, 개발자가 다중 GPU를 이용해 거대한 데이터 세트로 네트워크를 미리 훈련 시켜놨기 때문에 전이 학습이 가능함

2-9 Transfer Learning

1. 훈련 데이터셋이 작은 경우 (데이터셋이 작을 때 전이학습이 더 유용함)



- 코드만 다운 받는 것이 아니라 가중치도 다운 받아야 함
- 위의 신경망을 전이학습시킨다고 가정하면, 많은 네트워크가 ImageNet 같은 데이터셋을 기반으로 훈련한 것이며, 수천 개의 클래스가 존재해 소프트맥스 유닛은 그 중 하나를 출력함.
- 따라서 나의 목적에 맞게 클래스를 분류하기 위해, 소프트맥스층을 없애서 직접 소프트맥스 유닛을 만들어야 함(네트워크의 관점에서 소프트맥스층 이전의 모든 층이 고정되어있다고(freeze) 생각하면, 이전 층의 변수들이 고정되어있고(훈련 x), 소프트맥스 층과 관련된 변수만 훈련시킴 - trainableParameter=0 또는 freeze=1 설정)
- 누군가가 이미 훈련시킨 가중치를 사용함으로써 작은 데이터셋으로 좋은 성능 구현 가능

2. 훈련 데이터셋이 중간 크기인 경우

- 네트워크의 일부 층만 동결하고 마지막 몇 개의 층과 소프트맥스층을 class 수에 맞춰 학습시킴
- 데이터가 많을수록 동결시키는 층의 개수는 줄어듦고 훈련시킬 층의 개수가 늘어남
- 훈련 데이터셋이 더 많을 경우, 마지막 몇 개의 층을 조합한 작은 신경망 훈련 가능

3. 훈련 데이터셋이 아주 많은 경우

- 오픈 소스 네트워크와 가중치 전부를 초기화과정으로 사용하고 네트워크 전체를 다시 훈련시킴
- ex) 다운 받은 가중치 전부를 초기 값으로 사용해 무작위 초기화를 대체한 뒤 경사하강법 사용 (네트워크의 모든 층을 훈련시키고 업데이트)
- 이례적으로 큰 데이터셋과 예산을 가지고 있지 않는 이상, 컴퓨터비전은 전이학습을 해야만 하는 분야임.

2-10 Data Augmentation

- Data Augmentation: 적은 데이터로 모델의 성능을 향상시키기 위해 데이터셋에 변형해서 새로운 데이터를 추가적으로 생성하는 기법
- Data Augmentation 종류
 1. Mirroring: 이미지를 좌우 반전시키는 방법
 2. Random Cropping: 이미지에서 랜덤하게 일부를 잘라내는 방법, 너무 랜덤하게 자르면 본질적인 정보를 잃어버릴 수 있음.
 3. Rotation(회전), Shearing(이미지 비틀기), Local warping ...
 4. Color Shifting: 이미지의 RGB값에 작은 변화를 주어 이미지의 색상을 변화시키는 방법, 조명이나 환경에 따른 색상 변화는 이미지의 본질적인 정보에 영향을 주지 않으므로, 이를 학습하게 해 학습 알고리즘이 색의 변화에 더 잘 반응할 수 있게 함
- cf) PCA Color Augmentation: 주성분 분석을 활용해 색조에 변형-> but 전체적인 색조는 유지