Lecture 8

1. CPU와 GPU

2. Deep Learning Frameworks

   2.1 Caffe/ Caffe2

   2.2 Theano / TensorFlow

   2.3 Torch / PyTorch

CPU(central processing unit): 적은 수의 코어로 처리할 수 있는 작업이 많다.

GPU(graphics processing unit): 코어가 많고, 단순한 계산을 매우 빠르게 처리 가능함.

-deep learning Framework

1. Caffe 에서 Caffe2로 발전.

2. Torch에서 PyTorch로 발전.

3. Theano가 TensorFlow 로 발전.

딥러닝 프레임워크의 특징:

1. 대규모 계산 그래프를 쉽게 구축 가능.

2. 계산 그래프에서 기울기 계산 쉽게 가능.

3. GPU라이브러리 활용하여 효율적 GPU 실행 가능.

넘파이로는 GPU를 돌릴 수 없어서, TensofFlow, Pytorch 함께 사용.

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

TensorFlow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

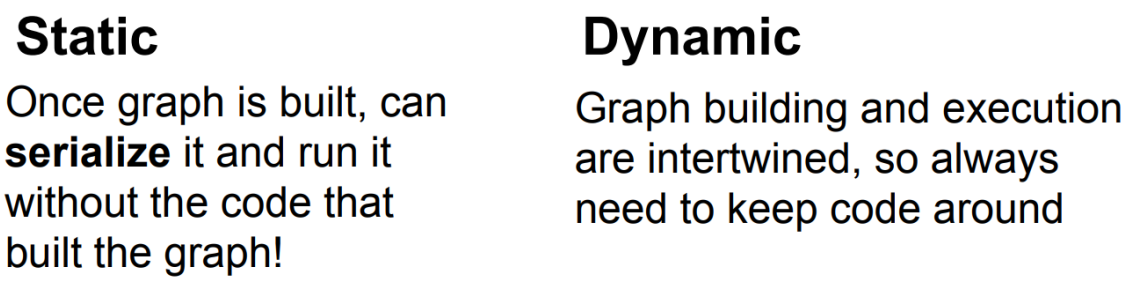
각각의 형태.

Torch:

- 단점: Lua 언어를 사용한다.
- 단점: 자동 미분 기능이 없다.
- 장점: 안정적인 소프트웨어이다.
- 장점: 많은 기존 코드가 있다.
- 공통점: 빠르다.

PyTorch:

- 장점: Python 언어를 사용한다.
- 장점: 자동 미분 기능이 있다.
- 단점: 상대적으로 새로운 프레임워크이며 아직 변화하는 중이다.
- 단점: 기존 코드 양이 적다.
- 공통점: 빠르다.



->

Static: Pytorch나 TensorFlow같은 프레임워크 사용하면, 그래프 재사용이 가능.

Dynamic: 계산 그래프의 구성과 실행이 강하게 연결, 그래프 구성 코드를 항상 유지해야 함.

-Lecture 9-

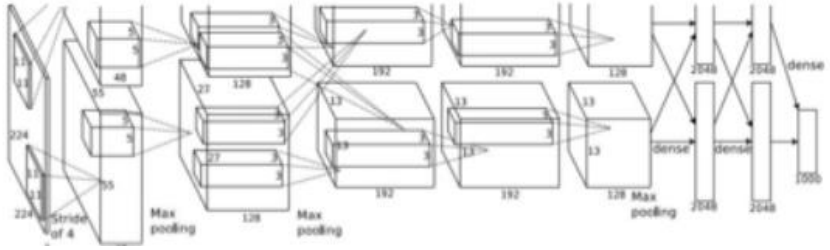
CNN 연구

LeNet-5(1998년):

숫자 인식에 사용.

이미지 Input -> [5 X 5] 필터 사용 -> 스트라이드 1 적용 -> Fully Connect Layer -> output

AlexNet(2012년):

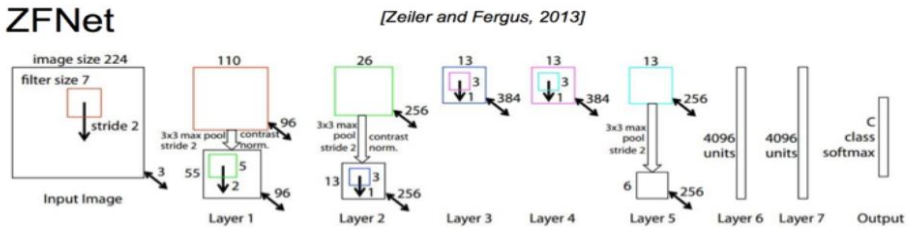


input([227 * 227 * 3]) -> CONV1(1st 계층, 96개의 11 * 11 필터가 스트라이드 4로 적용.)

-> 출력([55 * 55 * 96])

파라미터: (11 * 11 * 3) * 96 = 35000

ZFNet:



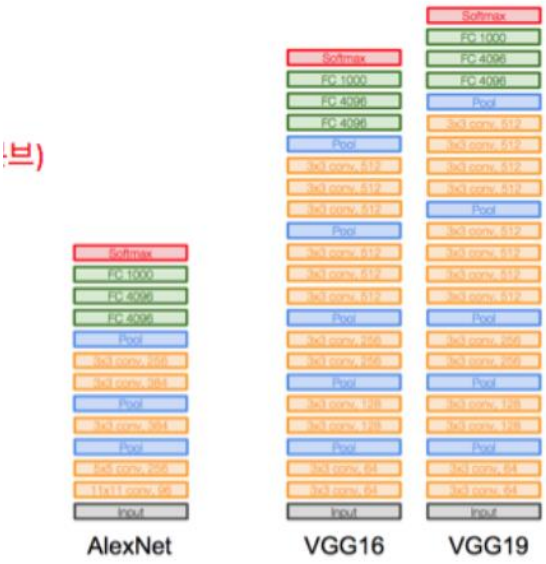
AlexNet과 비슷한 구조.

CONV1(1st 계층):11*11 스트라이드 4에서 7*7 스트라이드 2로 변경.

CONV3,4,5: 384, 384, 256개 필터 대신, 512, 1024, 512개 사용.

-> 오류율 개선.

VGGNet:



더 깊은 계층.

1. AlexNet의 8계층에서 16,19계층으로 늘림.

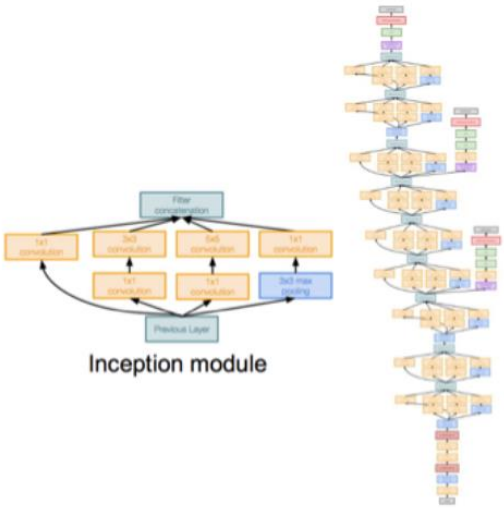
2. 작은 필터 사용(더 적은 파라미터, 그걸 더 깊이.)

but 더 깊으면 깊을수록 비선형적.

-> VGG19가 더 좋은데, 더 많은 메모리 사용.

더 좋은 효과를 얻으려면, 앙상블 사용.

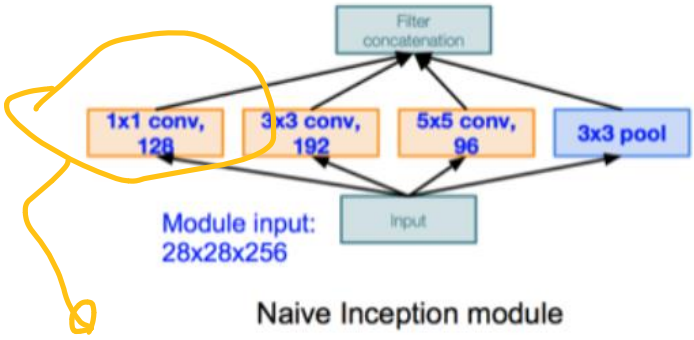
GoogLeNet:



더 깊은 계층. (22개 계층)

Fully Connected(FC)계층 없음.

인셉션 모듈(좋은 지역 망 위상)차곡차곡 쌓는 느낌.

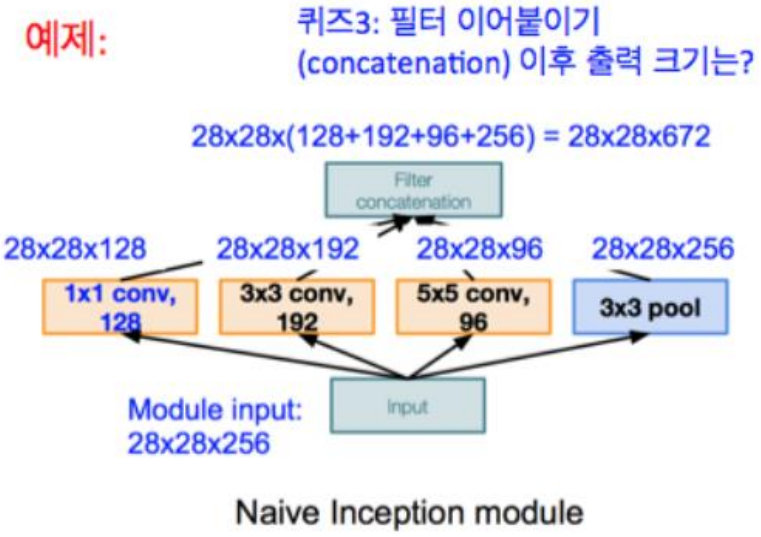


하나의 필터.

첫번째 conv -> 28*28*128 크기 출력

두번째 conv -> 28*28*192 크기 출력

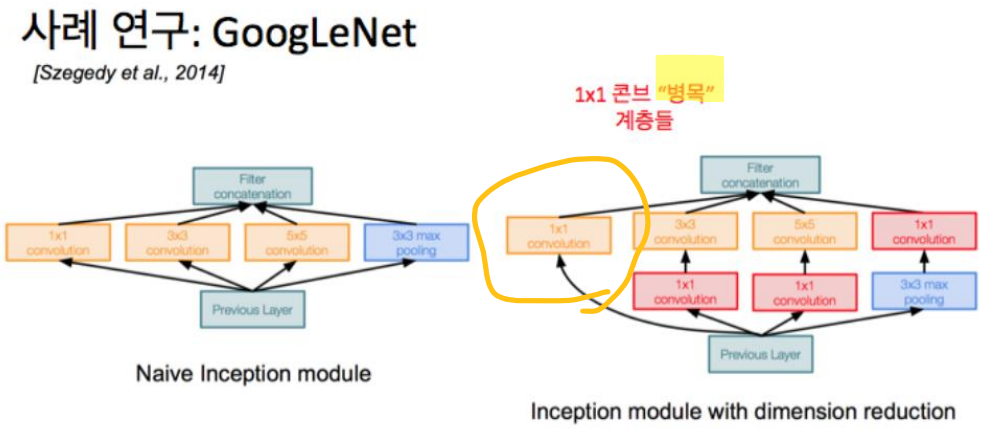
세 번째 conv -> 28*28*96



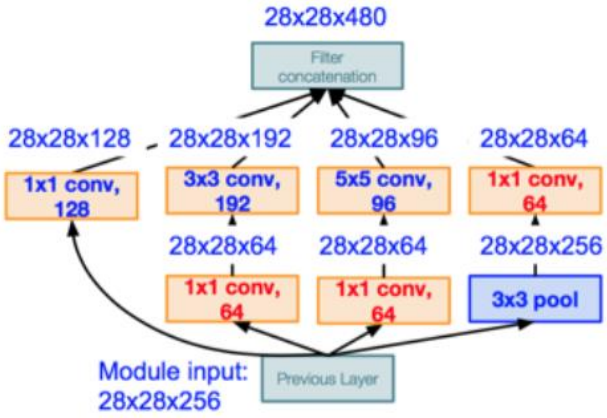
그래서 결국 최종 필터에는 28*28*672 크기.

-> 점점 계산을 늘려가는 느낌이기 때문에 매우 비쌈.

그래서 이걸 병목(bottleneck)계층 사용해서 해결.

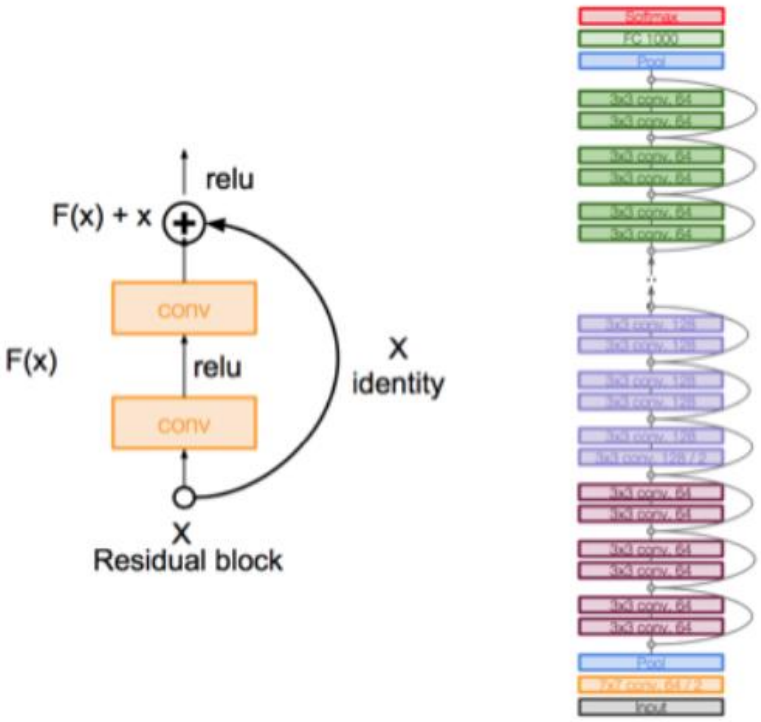


이런식으로 병목 계층 추가.(1*1 conv계층)



실제 최종 필터, [28*28*672]에서 [28*28*480]로 감소.

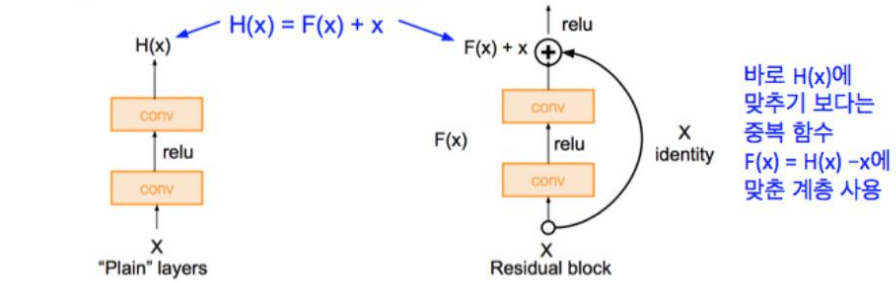
ResNet:



기존의 망보다 훨씬 더 깊은 망.

152계층 모델. 계층이 깊으면 깊을수록 loss가 더 낮아지는 것은 아님.

해결책: 직접적으로 바라는 기저 사상 (desired underlying mapping)에 맞추려고 하지말고
중복 사상 (residual mapping)에 맞춘 망 계층 사용.



ResNet이 계층이 매우 깊은 것은 사실. 하지만 깊다고 낮은 손실값을 내는 것은 아니기 때문에, 몇 가지 계층은 뛰어넘는 식의 계층 구성.

1. CONV계층 이후 배치 정규화

2. SGD 모멘텀

3. 드랍아웃 사용 안함.

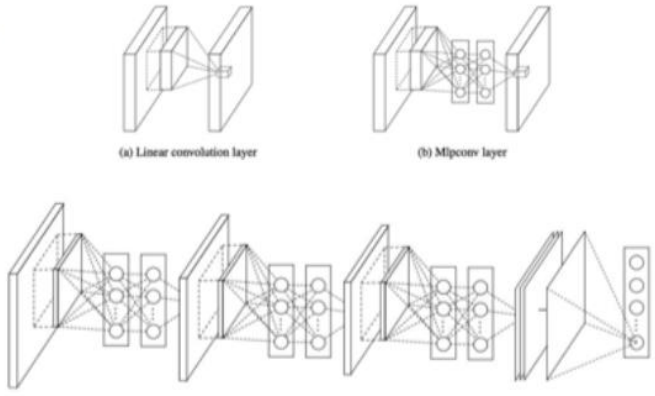
4. GoogleNet처럼 병목 계층 사용.

Network in Network(NiN):

Network in Network (NiN)

[Lin et al. 2014]

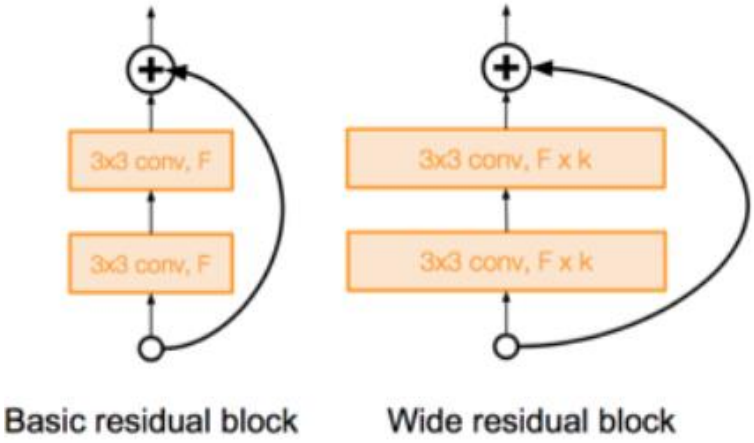
- Mlpconv 계층은 "micronetwork"가 있어서 각 콘브 계층내에서 지역 패치에 대한 더 추상적인 피쳐들을 계산함
- Micronetwork는 다계층 퍼셉트론을 사용 (FC, 즉, 1x1 콘브 계층들)
- GoogLeNet과 ResNet "병목" 계층에 대한 선구자
- GoogLeNet에 대한 철학적 영감을 줌



Figures copyright Lin et al., 2014. Reproduced with permission.

ResNet 개선 연구들.

1. Wide Residual Netrwoks

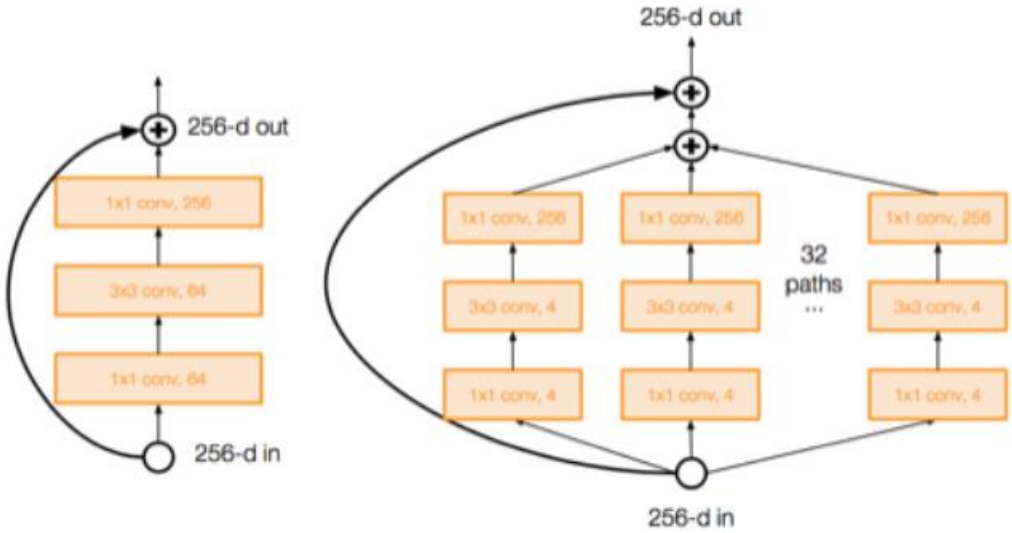


더 넓은 중복 필터 사용.

깊이 대신 넓이 증가하는게 더 효율적.

-> 그래서 이게 계층 50개인데, 계층 152개인 ResNet이김. ㅋㅋ

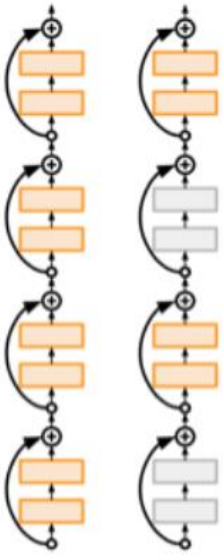
2. ResNeXt:



같은 ResNet 연구진이 만든 것.

블록 넓이 키우고, 경로는 인셉션 모듈(GoogleNet)과 비슷.

Deep Network with Stochastic Depth:

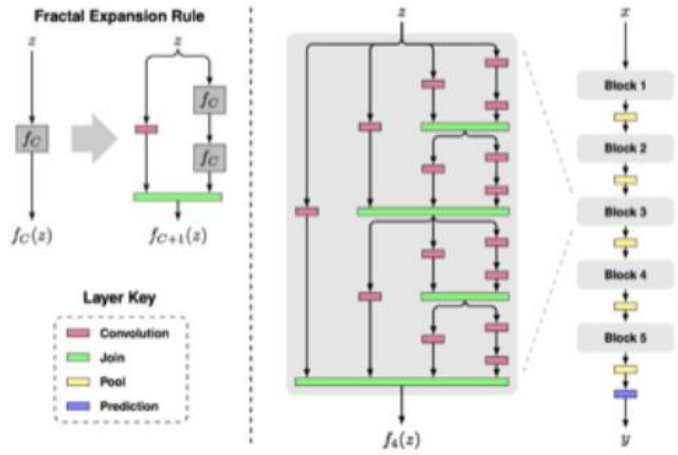


경사 사라지는 것 줄이고, 훈련동안 짧은 망 거쳐서 훈련 시간 줄임.

훈련 중 임의로 계층 드랍.

테스트 할 때는 또 전체 계층 사용.

FractalNet:

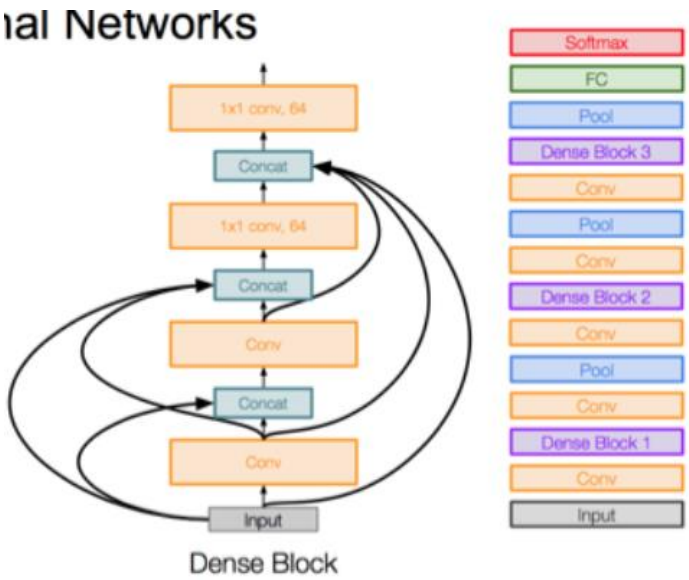


Figures copyright Larsson et al., 2017. Reproduced with permission.

중복표현 사용 안함.

훈련할 땐, 드랍아웃. 테스트엔 전체 계층 사용.

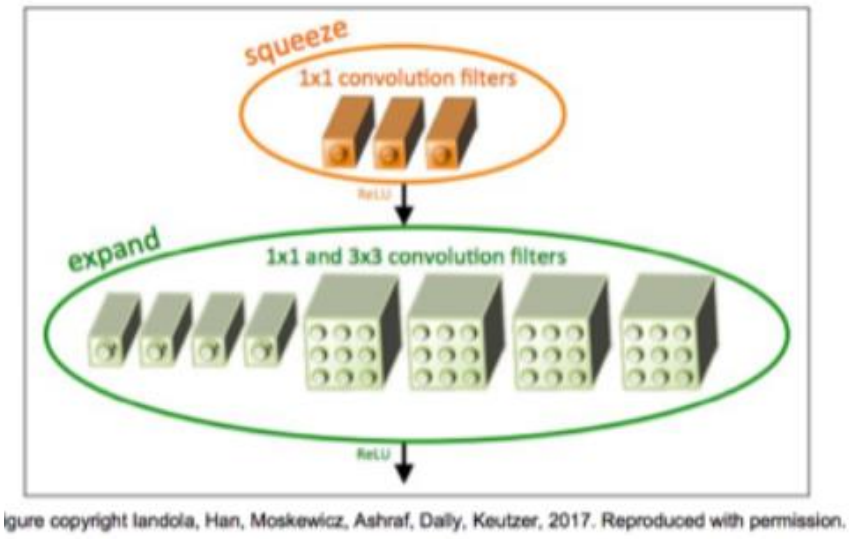
DenseNet:



각 블록, 밀도높게.

피쳐 재사용 구조.

SqueezeNet:



igure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.

squeeze 계층을 expand계층으로 전달.

50배 적은 파라미터로 AlexNet 수준의 정확도.

AlexNet보다 510배 더 작게 압축 가능.