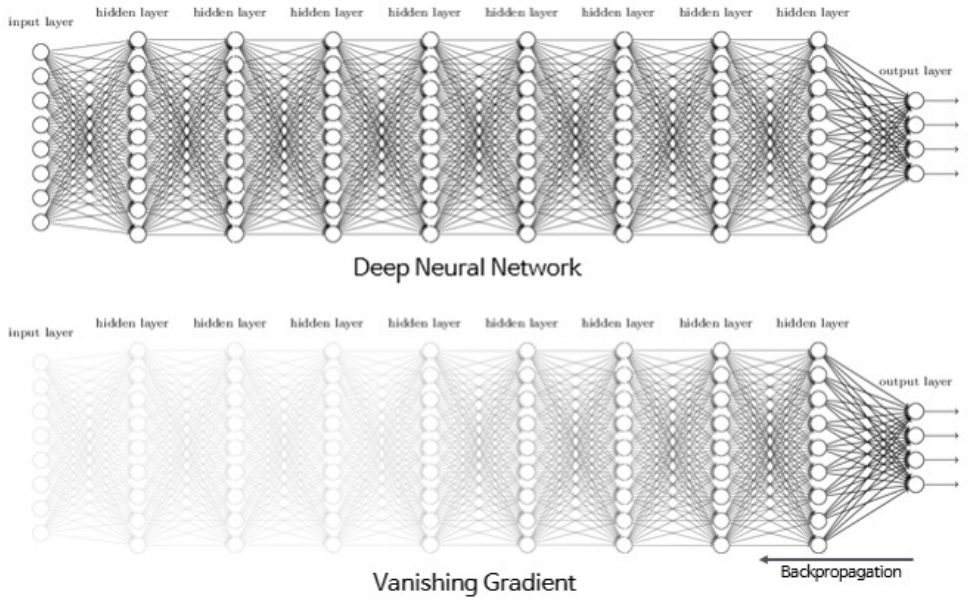
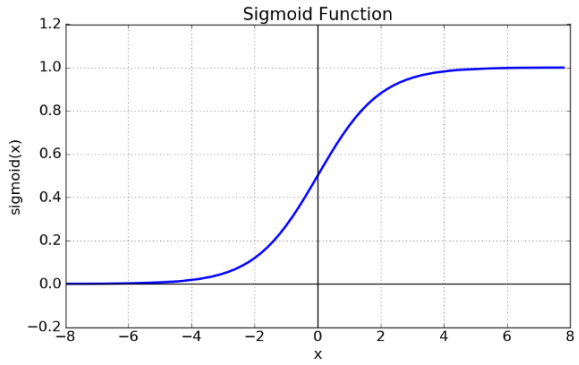
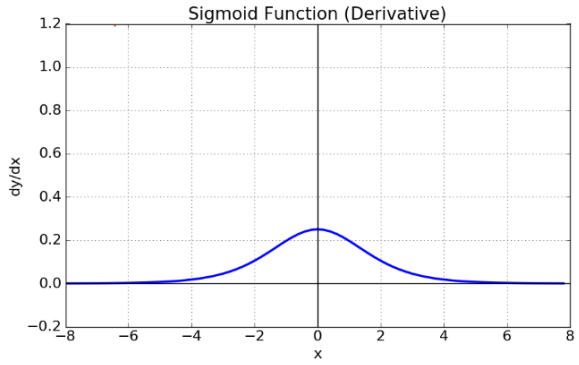
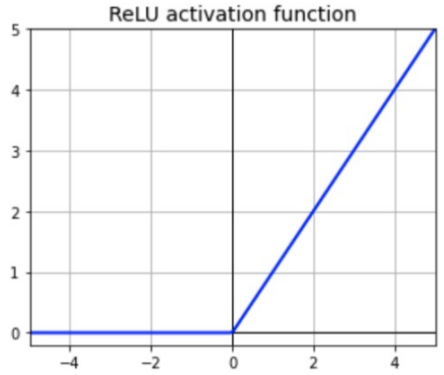
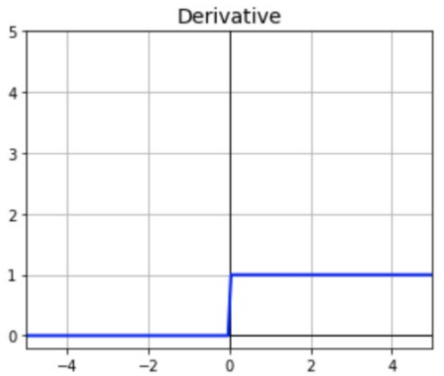
Background











Layer 개수가 100개를 넘어가는, 깊은 CNN 모델들이 발표되면서, 그와 동시에 깊은 CNN 모델들의 문제점인 "Vanishing Gradient" 문제가 대두되었다.

Vanishing Gradient란, 역전파 과정에서 입력층으로 갈수록 기울기가 점차 작아지는 현상을 말한다.

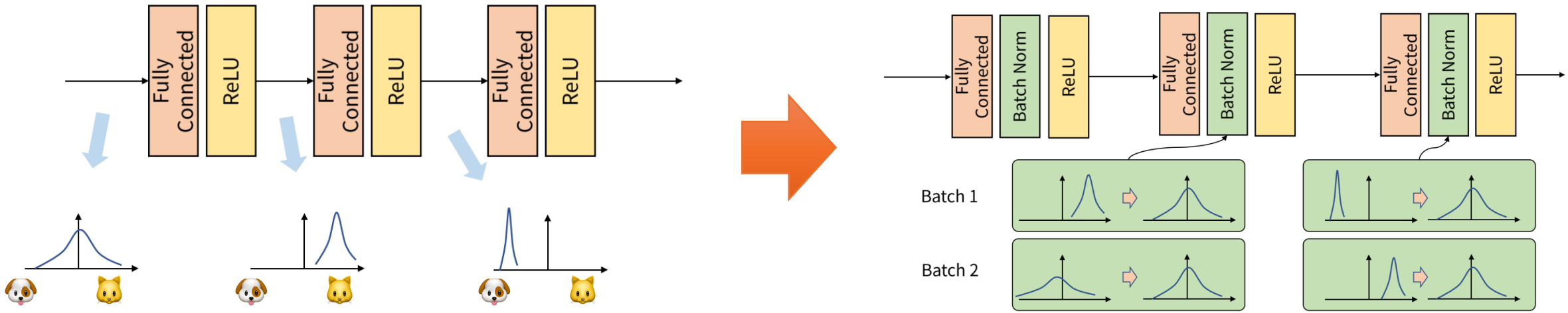
이를 해결하고자 다양한 방법들이 연구되었는데, 그 중 하나는 활성화 함수로 ReLU 함수를 사용하는 것이었다.

기존에는 sigmoid 함수를 사용하여, 역전파 과정에서 이 함수의 미분값을 계속 곱해 나가는 형식이었는데,

최대값이 0.25에 불과할 뿐더러 양 끝으로 갈수록 0에 가까워지는 터라 역전파를 진행할수록 Gradient가 많이 손실되는 것이 문제였다.

이에 대한 해결책으로 제시된 것이 ReLU 함수였고, 실제로 적용한 결과, Gradient 손실을 많이 방지할 수 있었다.

Background



Vanishing Gradient의 또 다른 해결책으로 Batch Normalization 기법을 제안했다.

극단적인 예를 들자면, training 과정에서는 고양이 사진만 보여주고 test 과정에서 강아지 사진만을 보여주면 좋은 학습 결과를 기대하기 힘들 것이다.

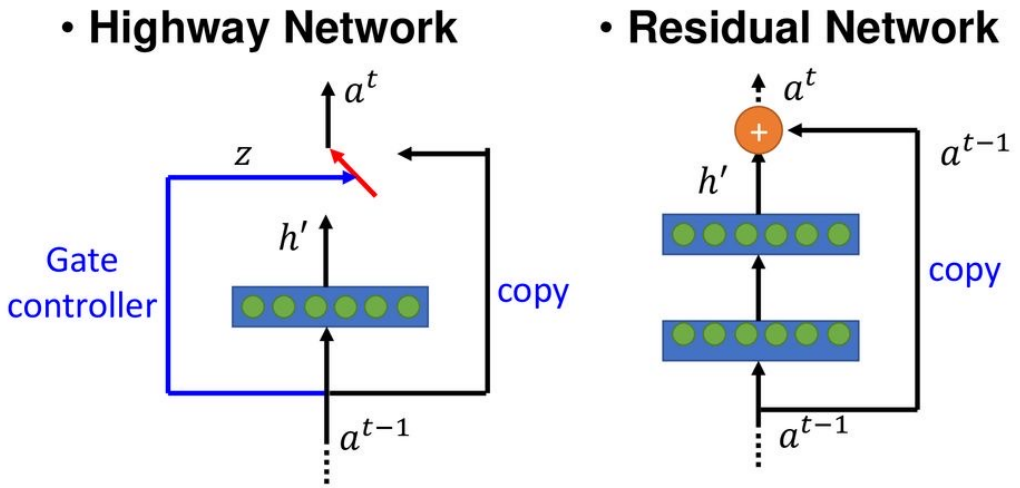
균일한 양의 이미지를 입력했다 하더라도, training 중에 연산 전/후로 데이터 간 분포가 달라질 수 있다. 이를 Internal Covariant Shift라고 한다.

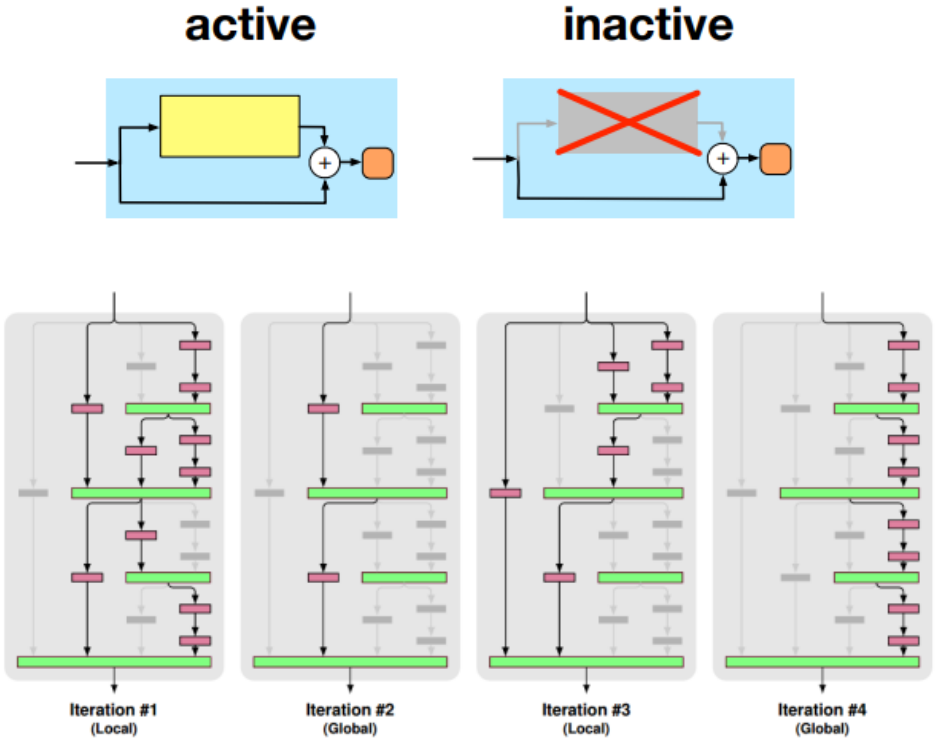
이에 대한 해결책으로 나온 것이 Batch Normalization 기법이다.

feature map들을 여러 mini batch들로 나눈 다음, 해당 batch의 평균과 표준편차를 구하여 batch 내의 값들을 Gaussian 형태로 정규화시키는 것이다.

그 결과 각각의 batch들의 평균은 모두 0이 되어 균형이 맞춰지게 되고, 이는 학습이 더 용이해질 수 있도록 도와주며 Gradient 소실 문제도 해결 가능하다고 한다.

Background





이 외에도 Vanishing Gradient 문제를 해결하기 위해 다양한 Technic들이 발표되었다.

Highway Network와 ResNet은 Identity Connection이라는 방식으로 Layer를 건너뛰는 방식을 선택했고,

Stochastic depth는 학습 중 Layer를 무작위로 Drop하는 방식을 적용시켜 Information과 Gradient의 흐름을 개선했다고 한다.

그리고 FractalNet의 경우, 여러 개의 병렬 Layer Sequence와 다양한 Convolution Block을 반복적으로 결합함으로써,

Network는 서로 다른 path를 통해 Layer들을 거치게 되고 실질적인 깊이보다 더 깊은 "명목상의" 깊이를 얻게 된다.

이 Model들의 공통점은, 선행 Layer와 후속 Layer 사이에 short-path를 형성하여 Training 과정에서 주요 특성을 공유한다는 것이다.

Background

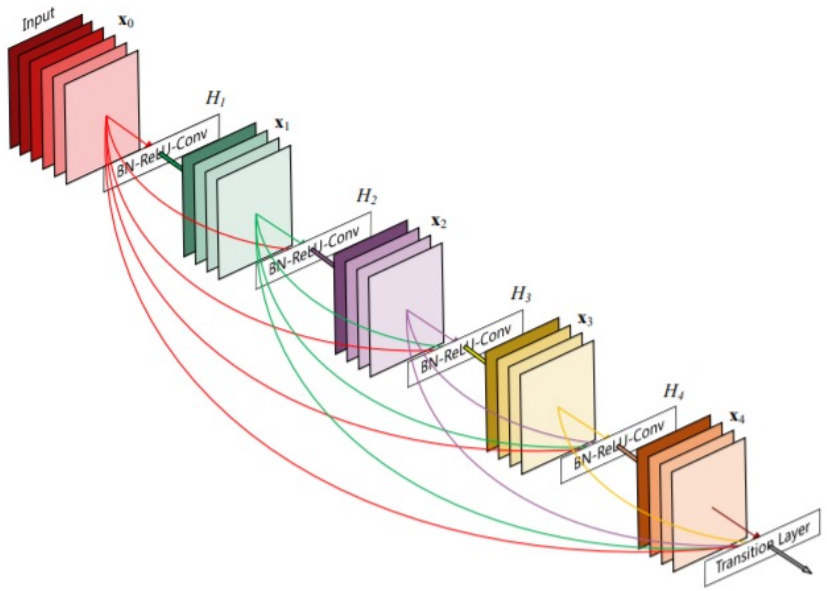


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

본 논문에서도 이 증명된 연구들을 기반으로 하여 새로운 Model 구조를 제안한다.

우선 앞선 Model들과 마찬가지로, 각각의 모든 Layer들을 순차적으로 연결하여 input부터 output까지 단방향으로 진행되는 Feed-forward 방식을 적용했다.

그리고 또 다른 공통점으로, Layer들 사이에 Connection Path를 놓음으로써 Layer들이 서로 연결되도록 구성하였다.

여기서 DenseNet만의 특징이라고 할 수 있는 것이, Feature Map 크기가 같은 모든 Layer들이 서로 직접 연결되었다는 것이다.

즉, 각각의 Layer들은 자신 앞에 있는 모든 선행 Layer로부터 정보를 제공받고, 자신 뒤에 있는 모든 후속 Layer로 정보를 전달하게 된다.,

ResNet과 마찬가지로 Gradient를 다양한 path를 통해 입력받고, ResNet보다 더 먼 거리의 Layer까지 Gradient 정보 전달이 가능하기에 Gradient 소실 방지 효과가 크다.

Background

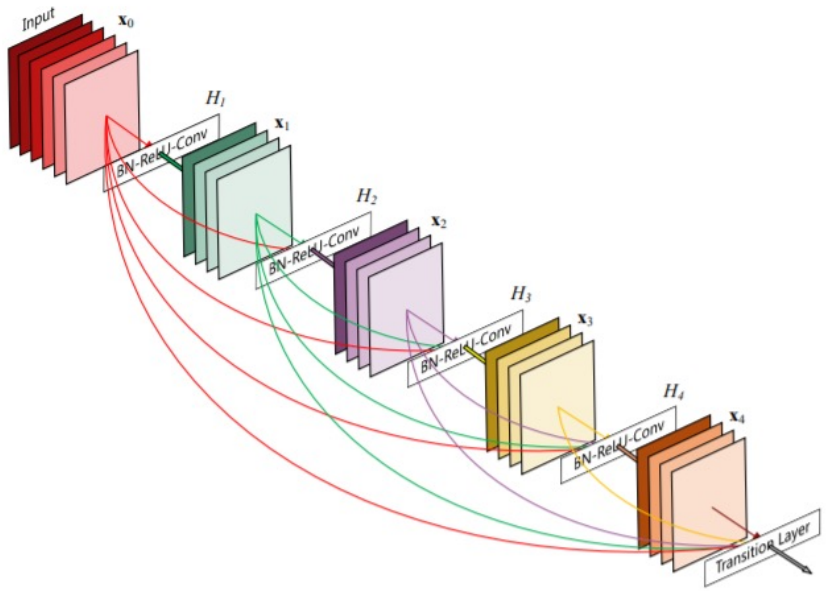


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

그 외에도 이러한 DenseNet의 구조로 인해 다양한 이점들이 존재한다.

우선 Feature Propagation에 강하다는 점이다.

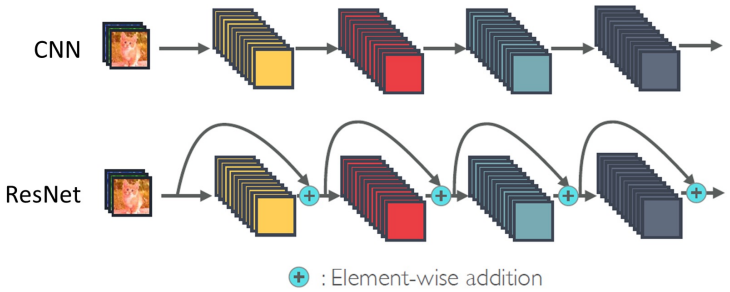
앞에서 만들어진 Feature Map을 뒤쪽까지 그대로 전달 가능하며, 기존 정보와 새로 만들어진 정보를 구분하여 전달하기에, 학습에 유리하다는 것이다.

또한 Parameter 수가 타 Model에 비해 적다고 할 수 있다.

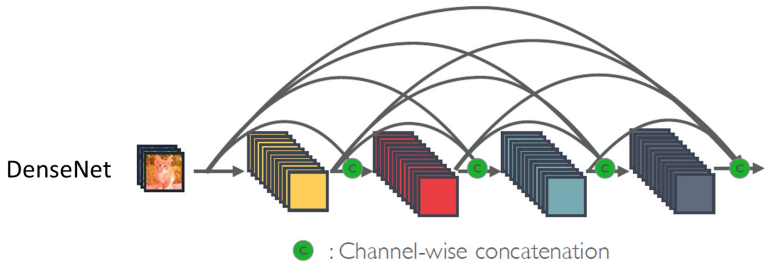
예를 들어, Channel 수가 32, 64, 128, 256, … 처럼 등비수열의 형태로 증가하는 VGG 같은 Model에 비해,

DenseNet은 등차수열의 형태로 증가하며, 공차를 Hyper Parameter로 조정 가능하기에 Parameter 수가 상대적으로 더 적다고 말한다.

Dense Connectivity



$$X_l = H_l(X_{l-1}) + X_{l-1}$$



$$X_l = H_l([X_0, X_1, \ldots, X_{l-1}])$$


```
import keras
import tensorflow as tf
import keras.backend as K

a = tf.constant([1,2,3])
b = tf.constant([4,5,6])

add = keras.layers.Add()
print(K.eval(add([a,b])))
#output: [5 7 9]

concat = keras.layers.Concatenate()
print(K.eval(concat([a,b])))
#output: array([1, 2, 3, 4, 5, 6], dtype=int32)
```

DenseNet도 ResNet과 마찬가지로 Layer 간 연결을 필요로 한다. 하지만 이 두 가지 Model 사이에는 차이점이 존재한다.

ResNet의 경우, 선행 Layer와 후속 Layer를 같은 위치의 Feature Map끼리 덧셈하는 방식으로 이루어진다.

때문에 덧셈을 위해서는 Layer 간 Channel 수가 같아야 한다는 조건이 있다.

하지만 DenseNet의 경우, Feature Map을 이어붙이는 방식이기에, 서로 Channel 수가 달라도 무관하다.

이러한 특성 덕에, DenseNet은 두 Layer가 합해진 후에도 합해지기 전의 정보가 그대로 남아있게 된다.

즉 새로운 정보를 학습함과 동시에 이전 Layer의 정보도 보존 가능하다는 장점이 있다.

Growth Rate

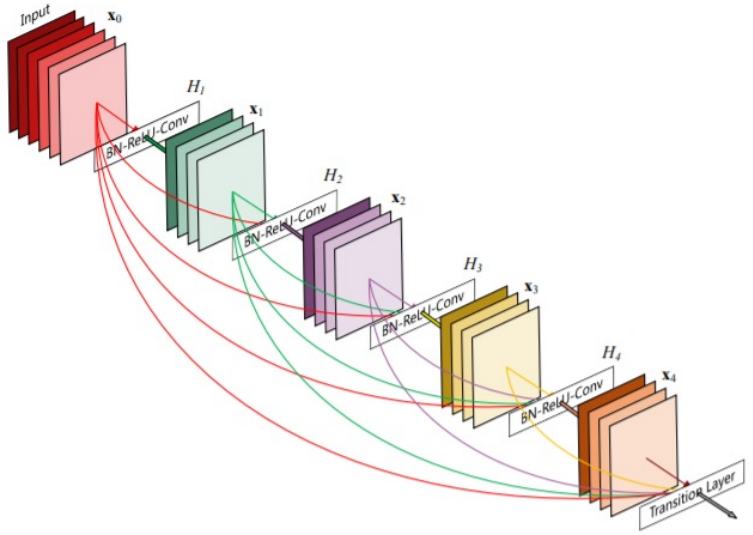


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

각 Layer에서 몇 개의 feature map을 뽑을지 결정하는 Parameter

혹은 각 Layer가 전체 output에 어느정도 기여할지를 결정하는 Parameter

DenseNet에서는 각각의 모든 Layer들을 서로 연결짓기 때문에

Layer의 Feature map 수가 조금만 늘어나도, 연결은 급격하게 증가한다

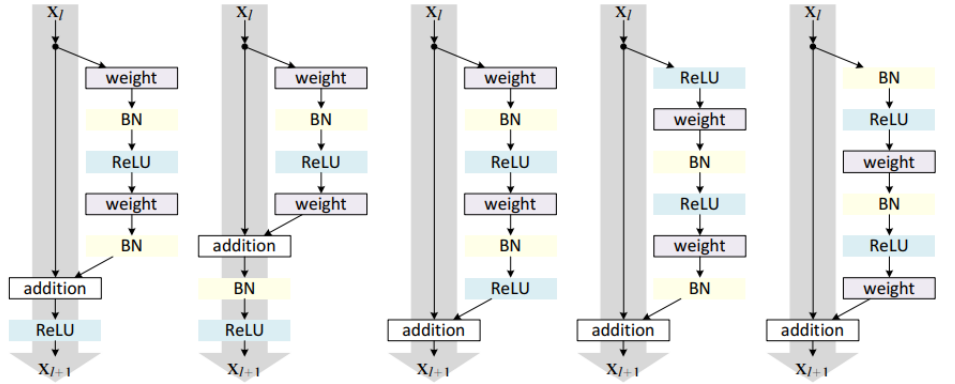
따라서 Growth Rate를 통해 channel 크기를 제한하여 Feature map 수를 조절할 필요가 있다.

Composite Function

$$X_l = H_l([X_0, X_1, \ldots, X_{l-1}])$$

H : Batch Normalization + ReLu + 3*3 Conv

case	Fig.	ResNet-110	ResNet-164
original Residual Unit [1]	Fig. 4(a)	6.61	5.93
BN after addition	Fig. 4(b)	8.17	6.50
ReLU before addition	Fig. 4(c)	7.84	6.14
ReLU-only pre-activation	Fig. 4(d)	6.71	5.91
full pre-activation	Fig. 4(e)	**6.37**	**5.46**



(a) original (b) BN after addition (c) ReLU before addition (d) ReLU-only pre-activation (e) **full pre-activation**

Layer들 간의 합산이 끝나면 이를 비선형 변환 함수에 통과시킨다.

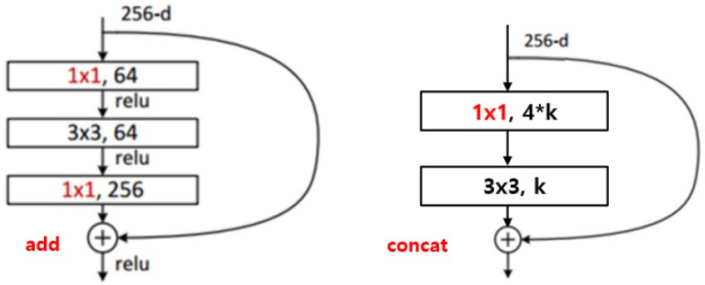
비선형 함수 H는 Batch Normalization, ReLu, Convolution Layer의 세 가지 연산의 복합 함수로 정의한다.

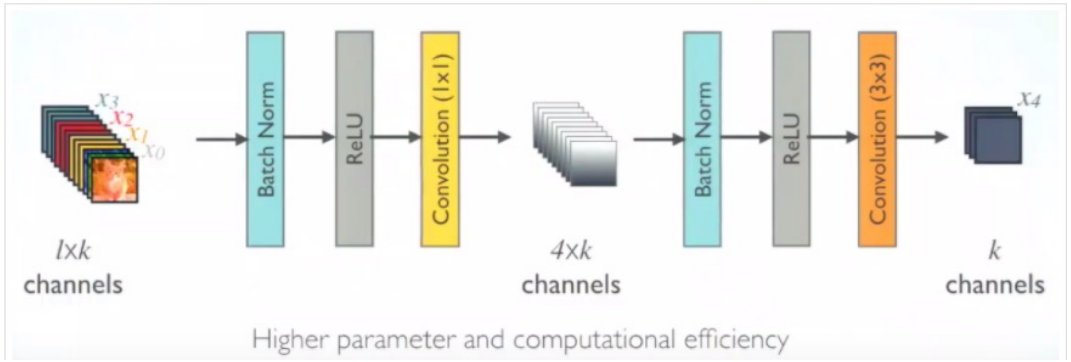
이는 ResNet의 개량형 Model에서 영감을 얻은 것이다.

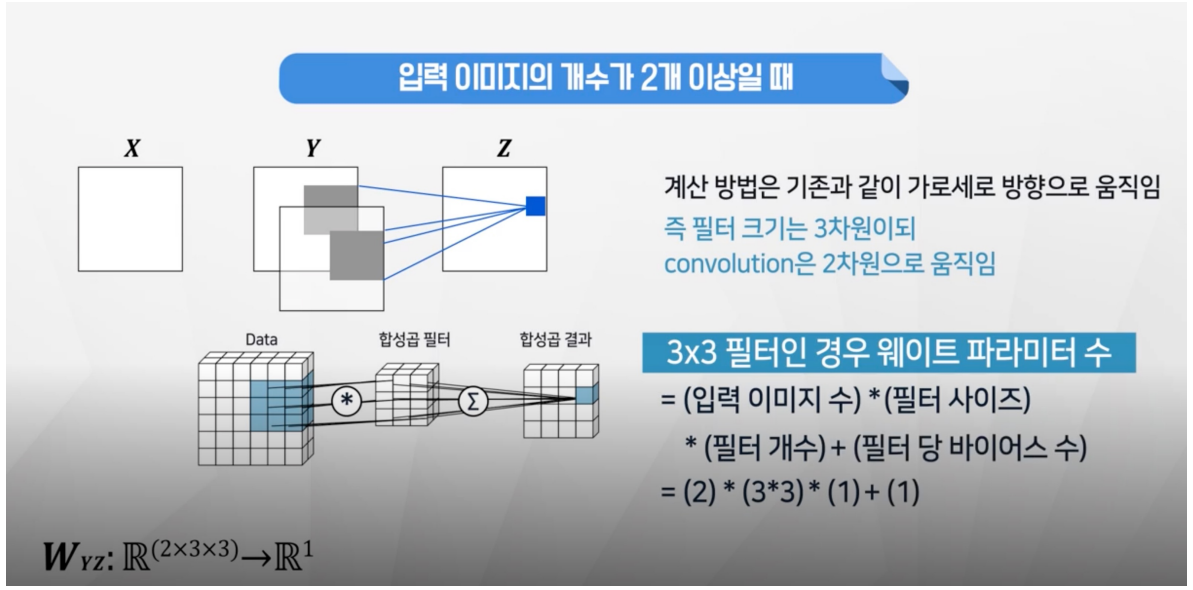
제일 Error율이 낮았던 것이 BN – ReLu – Convolution Layer 순으로 연결된 Pre-activation(Activation Function인 ReLu를 먼저 통과한다는 의미) 형태를 두 번 반복하는

Full Pre-activation의 형태로 이루어져 있다.

Bottleneck Layer







앞서 살펴본 Full pre-activation 형태에, ResNet에서 제안되었던 Bottleneck Layer 개념을 접목시키려고 한다.

바로 1x1 Convolution을 사용하는 것이다.

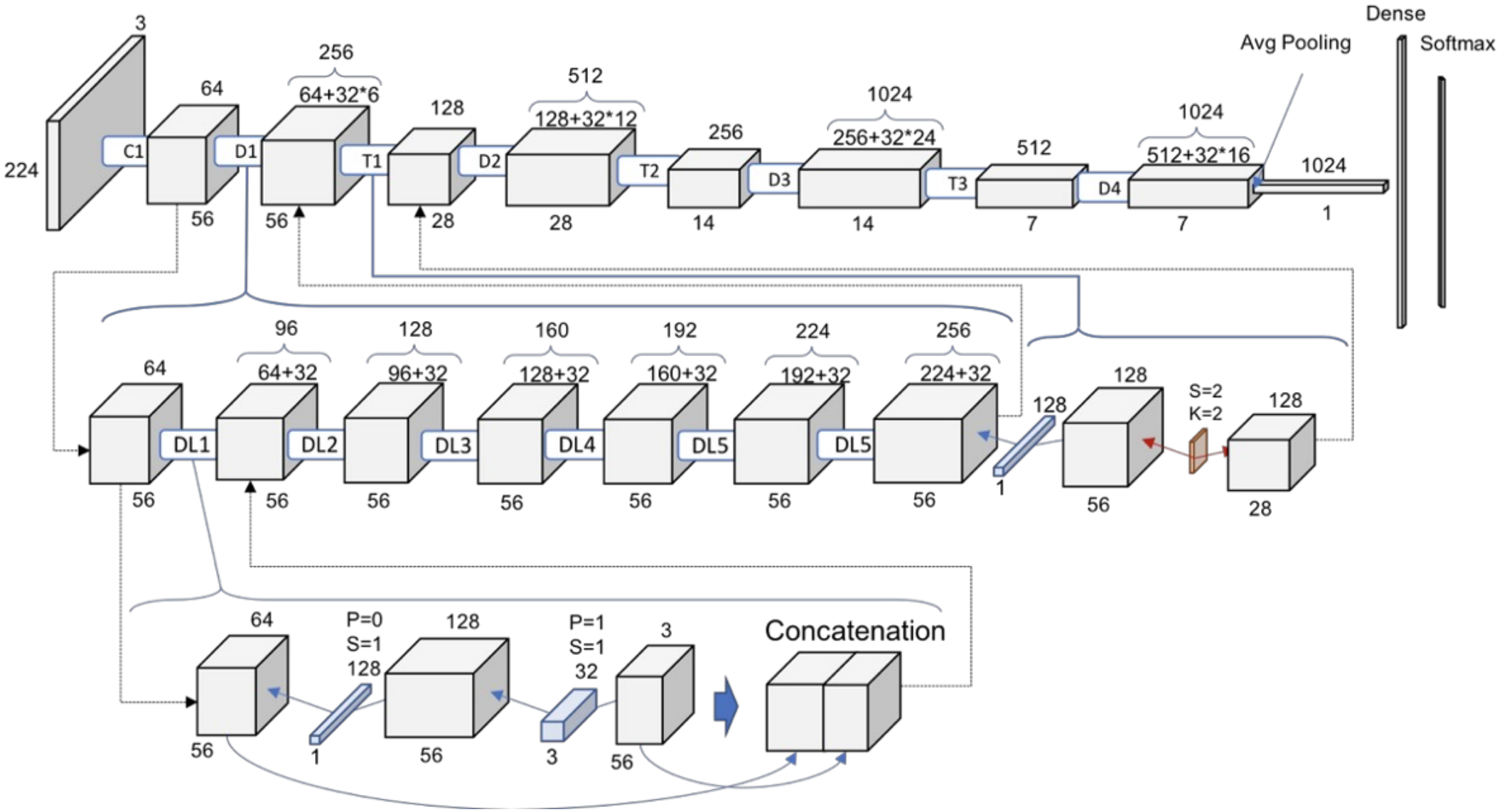
CNN에서는 Convolution Layer의 Channel 개수가 곱 연산으로 들어가기 때문에, Feature Map 개수가 많을 수록 Parameter 수가 급격하게 증가한다.

하지만 Channel 수가 적은 1x1 Convolution Layer를 사이에 집어넣음으로써 곱해지는 Channel 수를 줄여주면, 전체 Parameter 수도 감소하게 된다.

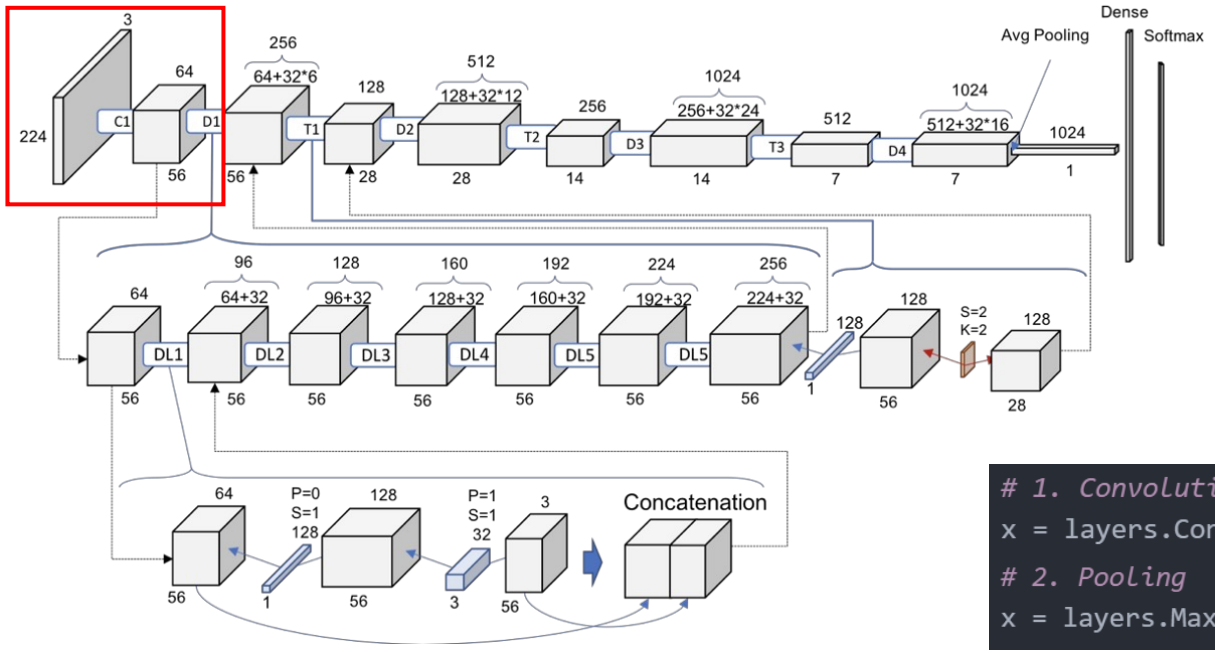
또한 Parameter 수가 적기 때문에 Overfitting이 일어나는 것도 어느 정도 방지해 줄 수 있다는 장점도 있다.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Dense-121


3
224
C1
64
56
D1
256
64+32*6
56
T1
128
28
D2
512
128+32*12
28
T2
256
14
D3
1024
256+32*24
14
T3
512
7
D4
1024
512+32*16
7
Avg Pooling
1024
1
Dense
Softmax
DL1
64
56
DL2
96
64+32
56
DL3
128
96+32
56
DL4
160
128+32
56
DL5
192
160+32
56
DL5
224
192+32
56
224+32
256
56
128
1
128
56
S=2
K=2
128
28
64
56
P=0
S=1
128
1
128
56
P=1
S=1
32
3
3
56
Concatenation

Convolution Layer



Layers	Output Size	DenseNet-121
Convolution	112 × 112	7 × 7 conv, stride 2
Pooling	56 × 56	3 × 3 max pool, stride 2

```
# 1. Convolution
x = layers.Conv2D(k * 2, (7, 7), strides=2, padding='same', input_shape=(224, 224, 3))(x)
# 2. Pooling
x = layers.MaxPool2D((3, 3), 2, padding='same')(x)  # 56x56x64
```

CNN의 feature map들이 많아질수록 Parameter 수도 증가하게 되는데, 이 경우 overfitting이 발생할 가능성이 있다.

때문에, 일반적으로는 max pooling을 적용하여 크기를 줄이게 되는데, kernel의 max 값이 그 kernel의 feature를 가장 잘 나타내기 때문이다.

하지만 Layer가 많아질수록(깊어질수록) feature들이 포함하는 정보가 더욱 중요해지는데,

Max pooling을 사용할 경우 이 미세한 feature들의 정보가 사라질 수 있다.

따라서, Conv의 stride를 늘리는 방식으로 크기를 축소함으로써 feature가 사라지는 것을 방지한다.

여기서는 stride 2, 7*7 Conv & stride 2, 3*3 Max poolng을 적용하였다.

Dense Block

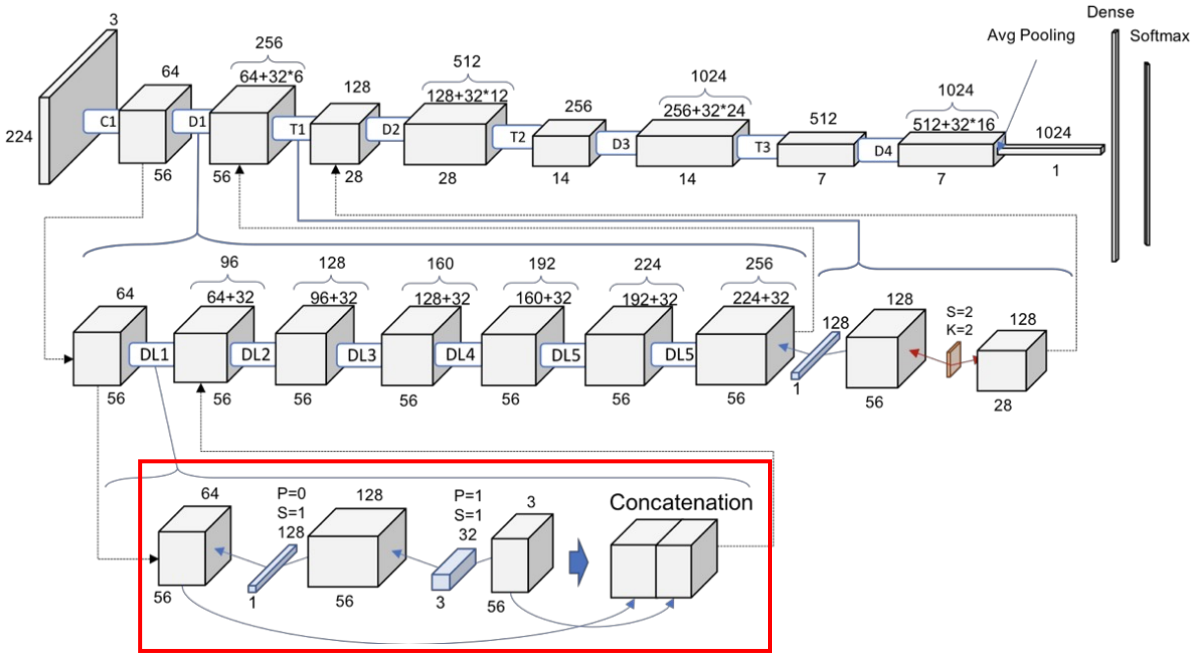
$$X_l = H_l([X_0, X_1, \ldots, X_{l-1}])$$

```python
# 3. Dense Block (1)
for i in range(6) :
    x_l = layers.Conv2D(k * 4, (1, 1), strides=1, padding='same')(x)    # 56x56x128
    x_l = layers.BatchNormalization()(x_l)
    x_l = layers.Activation('relu')(x_l)

    x_l = layers.Conv2D(k, (3, 3), strides=1, padding='same')(x_l)  # 56x56x32
    x_l = layers.BatchNormalization()(x_l)
    x_l = layers.Activation('relu')(x_l)

    x = layers.Concatenate()([x, x_l])  # 96 -> 128 -> 160 -> 192 -> 224 -> 256
```

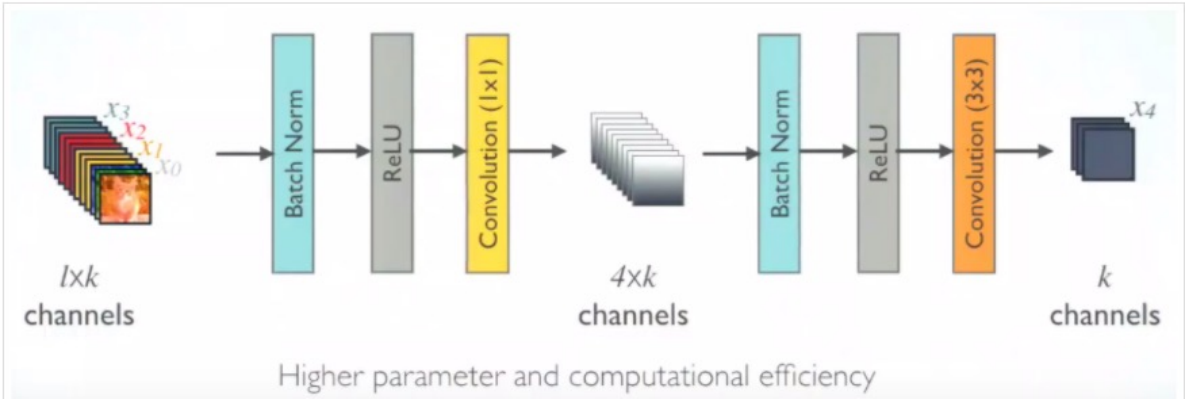
Layers	Output Size	DenseNet-121
Dense Block (1)	56 × 56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$

$X_l = H_l([X_0, X_1, \ldots, X_{l-1}])$

Bottleneck Layer

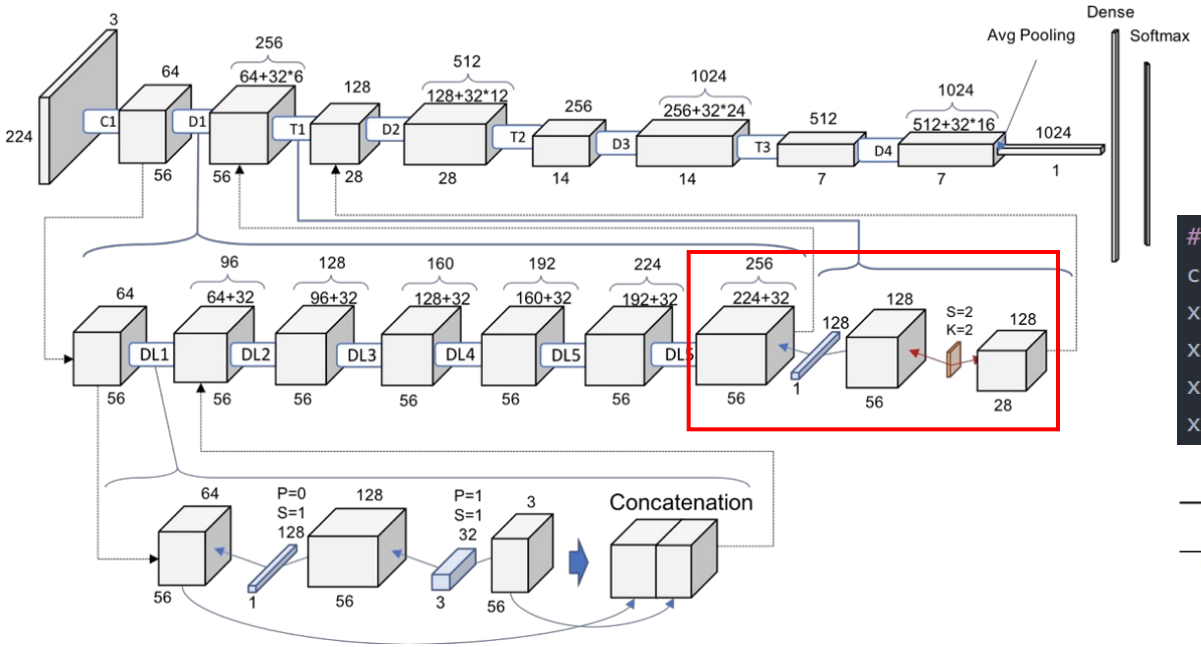



```python
# 3. Dense Block (1)
for i in range(6) :
    x_l = layers.Conv2D(k * 4, (1, 1), strides=1, padding='same')(x)    # 56x56x128
    x_l = layers.BatchNormalization()(x_l)
    x_l = layers.Activation('relu')(x_l)

    x_l = layers.Conv2D(k, (3, 3), strides=1, padding='same')(x_l)  # 56x56x32
    x_l = layers.BatchNormalization()(x_l)
    x_l = layers.Activation('relu')(x_l)

    x = layers.Concatenate()([x, x_l])  # 96 -> 128 -> 160 -> 192 -> 224 -> 256
```



Layers	Output Size	DenseNet-121
Dense Block (1)	56 × 56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$

Transition Layer



$$X_l = H_l([X_0, X_1, \ldots, X_{l-1}])$$

```
# 4. Transition Layer (1)
current_shape = int(x.shape[-1]) # 56x56x256
x = layers.Conv2D(int(current_shape * compression), (1, 1), strides=1, padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.AveragePooling2D((2, 2), strides=2, padding='same')(x)   # 28x28
```

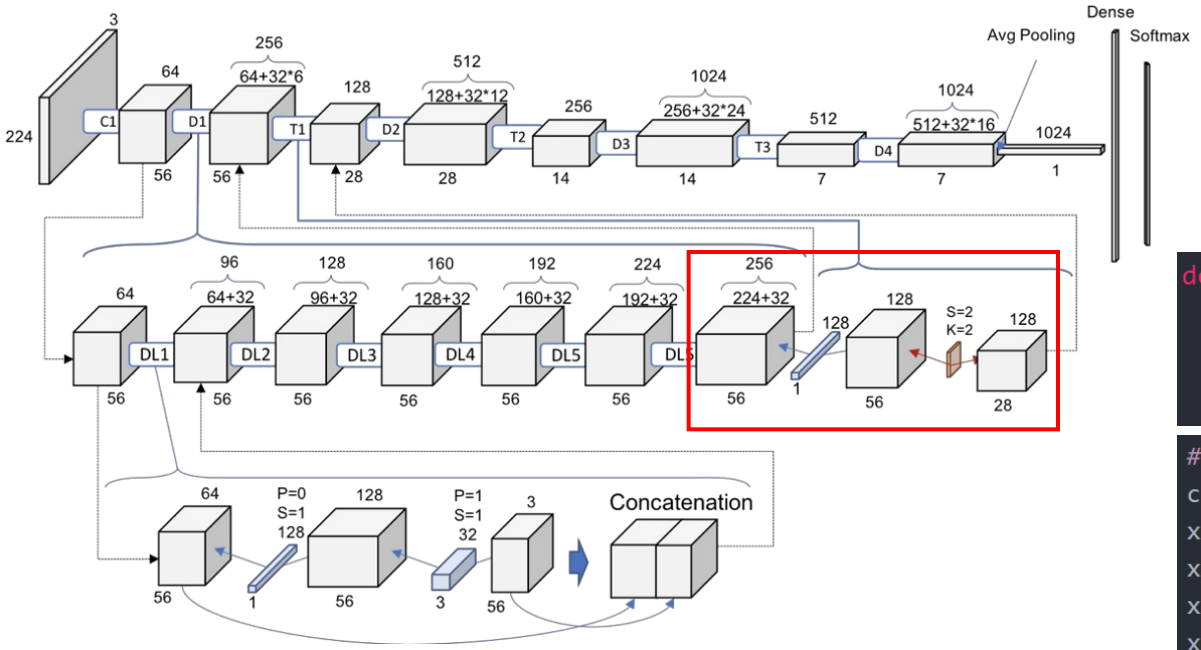
Layers	Output Size	DenseNet-121
Transition Layer (1)	56 × 56	1 × 1 conv
	28 × 28	2 × 2 average pool, stride 2

선행 Dense Block의 output과 후행 Dense Block의 input 크기를 맞춰주는 down-sampling layer

Batch Normalization Layer & 1*1 Convolution Layer & 2*2 Average pooling Layer

그 결과 Feature Map의 size 및 전체 Feature Map 개수는 반으로 감소, ex) 56*56*256 >> 28*28*128

Transition Layer - Compression



$$X_l = H_l([X_0, X_1, \ldots, X_{l-1}])$$

```
def DenseNet(x):
    # input = 224 x 224 x 3
    k = 32  # Grow Rate
    compression = 0.5  # compression factor
```

```
# 4. Transition Layer (1)
current_shape = int(x.shape[-1]) # 56x56x256
x = layers.Conv2D(int(current_shape * compression), (1, 1), strides=1, padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)
x = layers.AveragePooling2D((2, 2), strides=2, padding='same')(x)   # 28x28
```

Model의 조밀함(compactness)을 개선하기 위해, Transition Layer에서 Feature Map의 개수를 조절(감소 or 유지)

선행 Dense Block의 Output이 m개의 Feature Map을 갖고 있다면,

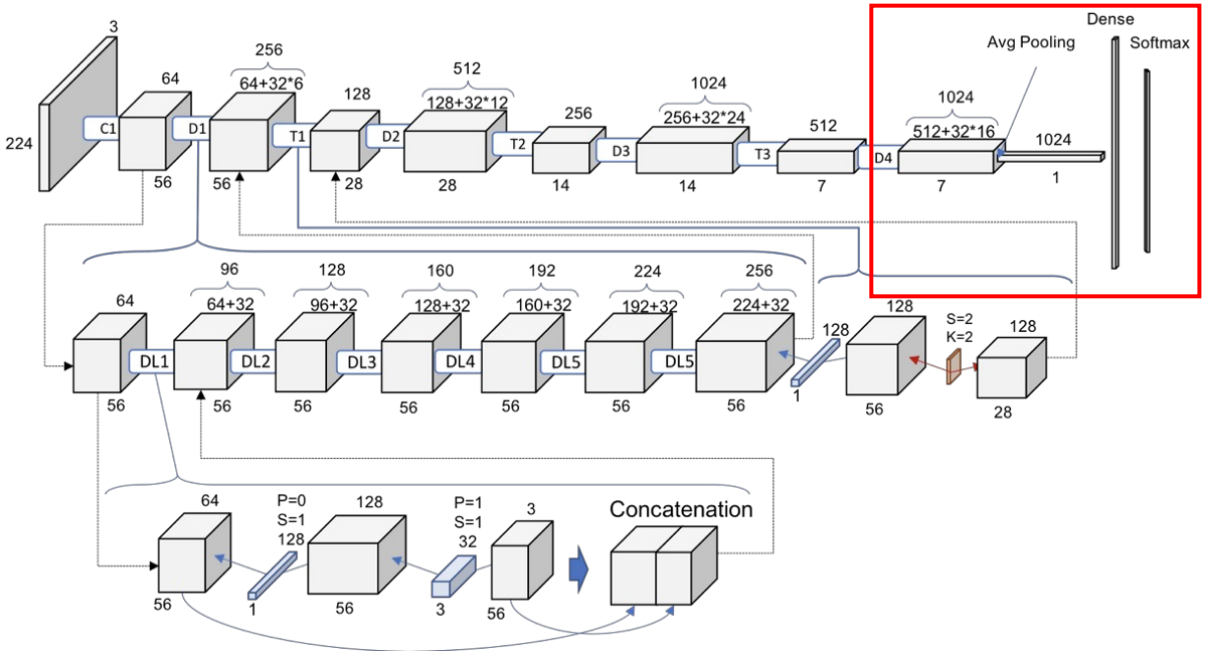
Transition Layer를 통과했을 때는 θm개의 Feature Map을 갖게 됨(0<θ≤1)

θ=1이면, 현상유지

0<θ<1이면, 이를 DenseNet-C라고 하며, 본 논문에서 실험에 사용된 DenseNet-C는 θ=0.5를 사용(위 그림의 경우 256->128)

Bottleneck Layer가 적용된 DenseNet-C의 경우, DenseNet-BC라고 칭함

Classification Layer



Layers	Output Size	DenseNet-121
Classification Layer	1×1	7×7 global average pool
		1000D fully-connected, softmax

```
# 10. Classification Layer
x = layers.GlobalAveragePooling2D()(x)
# classes = 2 (softmax)
x = layers.Dense(2, activation='softmax')(x)
```

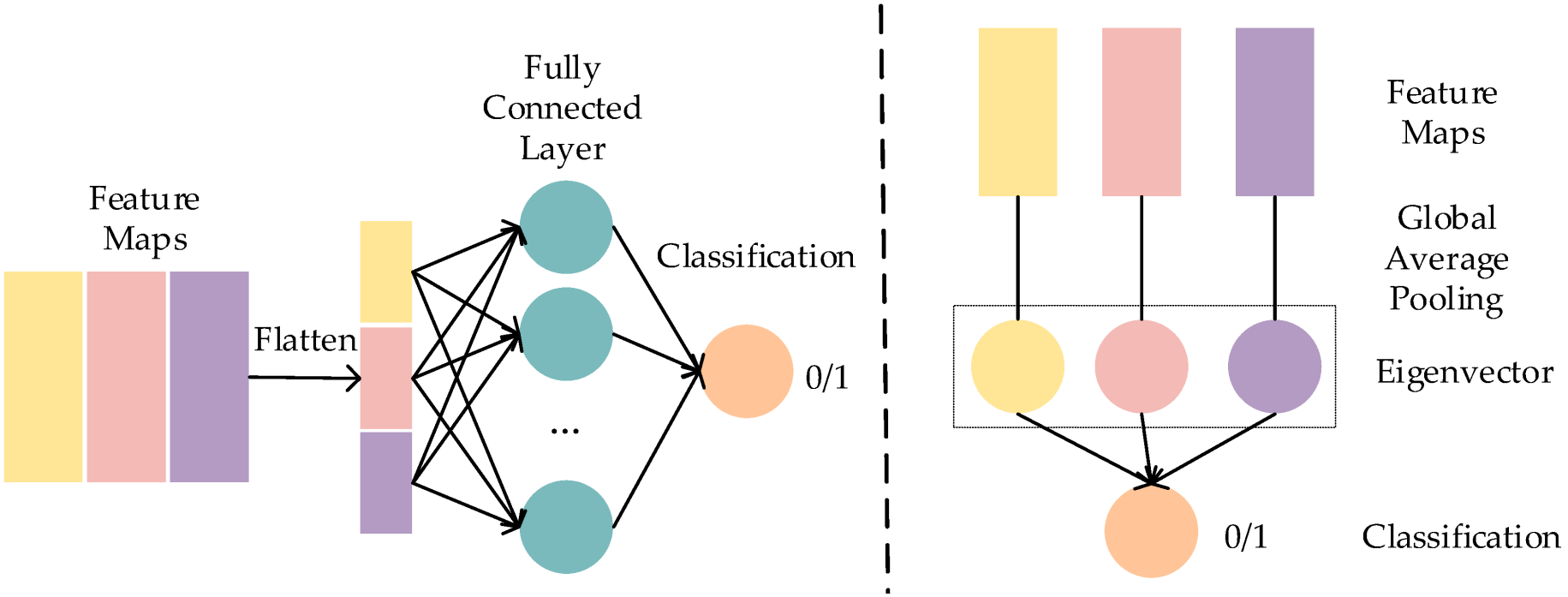
Feature를 1차원 벡터로 만들기 위해 Global Average Pooling 적용한다.

기존에는 CNN + Fully Connected Layer를 사용하였으나 대량의 Parameter와 가중치 요구, 그로 인한 압도적인 계산량, Feature의 위치 정보 손실 등의 문제가 발생한다.

반면 GAP를 사용할 경우 단순히 하나의 feature map을 하나의 출력 노드에 매핑하기 때문에, Parameter와 가중치를 훨씬 적게 사용한다.

따라서 계산량이 줄고 모델이 가벼워져 overfitting을 방지하는 효과도 가져온다.

또한 FC Layer를 사용했을 때에 비해 손실되는 정보도 더 적기에, 최신의 모델들은 GAP 방식을 주로 사용한다고 한다.


Feature
Maps
Flatten
Fully
Connected
Layer
Classification
0/1
...
Feature
Maps
Global
Average
Pooling
Eigenvector
0/1
Classification

Dataset & Training

Dataset

- CIFAR – CIFAR10, CIFAR100을 사용하여 각각 10개의 클래스 100개의 클래스를 사용
- SVHN – 거리 뷰 집 번호 데이터 셋
- ImageNet – ILSVRC 2012 분류 데이터셋 사용

Training

- SGD사용
- CIFAR10 – epoch 300, batch size 64, lr 0.1, epoch가 50%일 때 lr*0.1
- SVHN – epoch 40, batch size 64, lr0.1, epoch가 75%일 때 lr*0.1
- ImageNet – epoch 90, batch size 256, lr 0.1, epoch가 30, 60일 때 lr*0.1
- 10~4 weight decay와 0,9 Nesteroy momentum을 사용
- Dropout rate = 0.2

Classification Result on CIFAR and SVHN

Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [32]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [34]	-	-	-	7.72	-	32.39	-
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73	1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110	1.7M	11.66	5.23	37.80	24.58	1.75
	1202	10.2M	-	4.91	-	-	-
Wide ResNet [42]	16	11.0M	-	4.81	-	22.07	-
	28	36.5M	-	4.17	-	20.50	-
with Dropout	16	2.7M	-	-	-	-	1.64
ResNet (pre-activation) [12]	164	1.7M	11.26*	5.46	35.58*	24.33	-
	1001	10.2M	10.56*	4.62	33.47*	22.71	-
DenseNet ($k = 12$)	40	1.0M	**7.00**	5.24	**27.55**	24.42	1.79
DenseNet ($k = 12$)	100	7.0M	**5.77**	**4.10**	**23.79**	**20.20**	1.67
DenseNet ($k = 24$)	100	27.2M	**5.83**	**3.74**	**23.42**	**19.25**	**1.59**
DenseNet-BC ($k = 12$)	100	0.8M	**5.92**	4.51	**24.15**	22.27	1.76
DenseNet-BC ($k = 24$)	250	15.3M	**5.19**	**3.62**	**19.64**	**17.60**	1.74
DenseNet-BC ($k = 40$)	190	25.6M	-	**3.46**	-	**17.18**	-

Table 2: Error rates (%) on CIFAR and SVHN datasets. k denotes network's growth rate. Results that surpass all competing methods are **bold** and the overall best results are **blue**. "+" indicates standard data augmentation (translation and/or mirroring). $*$ indicates results run by ourselves. All the results of DenseNets without data augmentation (C10, C100, SVHN) are obtained using Dropout. DenseNets achieve lower error rates while using fewer parameters than ResNet. Without data augmentation, DenseNet performs better by a large margin.

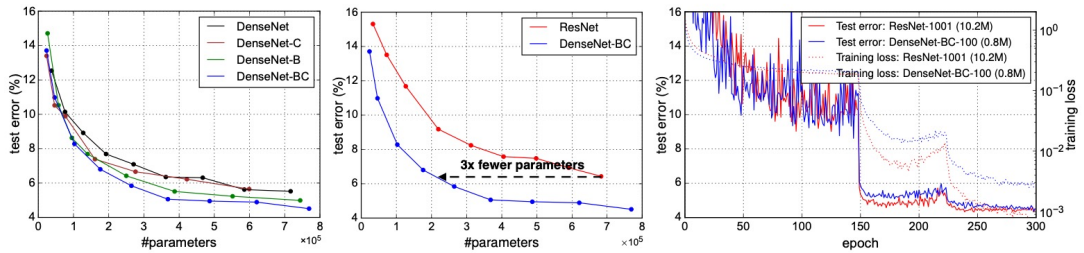


Figure 4: *Left:* Comparison of the parameter efficiency on C10+ between DenseNet variations. *Middle:* Comparison of the parameter efficiency between DenseNet-BC and (pre-activation) ResNets. DenseNet-BC requires about 1/3 of the parameters as ResNet to achieve comparable accuracy. *Right:* Training and testing curves of the 1001-layer pre-activation ResNet [12] with more than 10M parameters and a 100-layer DenseNet with only 0.8M parameters.

Classification Result on ImageNet

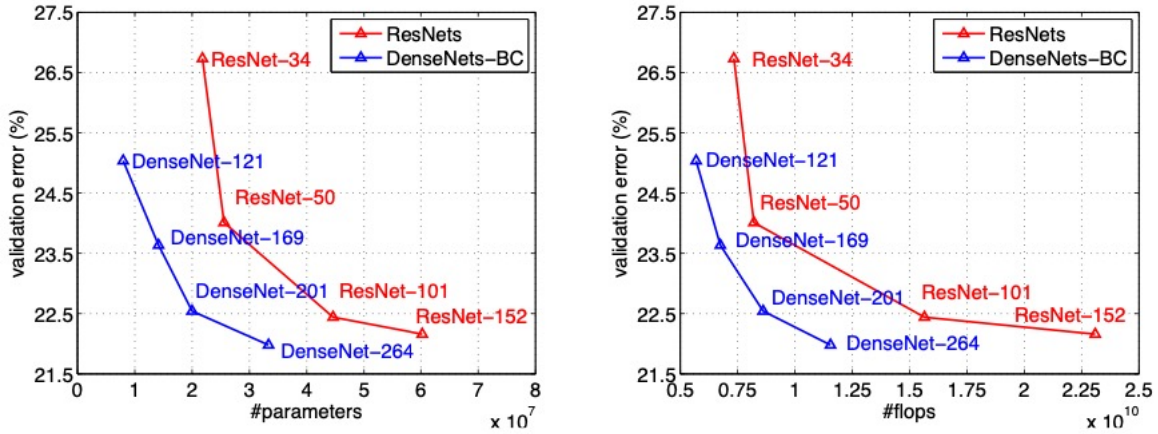


Figure 3: Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*) and FLOPs during test-time (*right*).

플롭스(**FLOPS**, **FL**oating point **O**perations **P**er **S**econd)는 컴퓨터의 성능을 수치로 나타낼 때 주로 사용되는 단위이다.

초당 부동소수점 연산이라는 의미로 컴퓨터가 1초동안 수행할 수 있는 부동소수점 연산의 횟수를 기준으로 삼는다. 상위 단위와 하위 단위로 국제단위계의 표준 접두어를 사용하며, 슈퍼 컴퓨터의 성능을 나타낼 경우에는 테라플롭스(1×10^{12} 플롭스)가 주로 쓰인다.

개인용 컴퓨터의 CPU 성능의 척도로 클럭의 속도 단위인 헤르츠를 주로 사용하는데, 이는 마이크로프로세서의 아키텍처의 구조에 따라 클럭당 연산 속도가 다르기 때문에 객관적인 성능을 비교할 때에는 플롭스를 사용한다.

Conclusion

- 실험결과 DenseNet은 수백개의 layer로 자연스럽게 확장하면서도 최적화 문제를 일으키지 않음
- 성능 저하나 overfitting의 징후 없이 매개변수 수가 증가함에 따라 정확도가 지속적으로 향상 됨
- 적은 수의 파라미터와 컴퓨팅만으로도 최신 성능에 도달할 수 있음