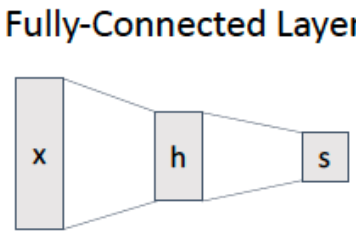
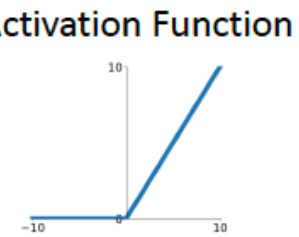
Lecture 7- Convolutional Networks

1. Components of a Network

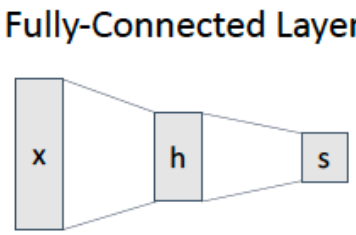
- **Components of a Fully-connected Network**

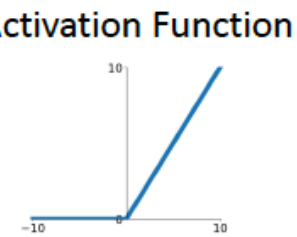


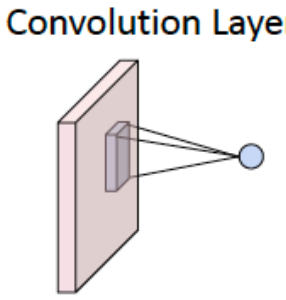


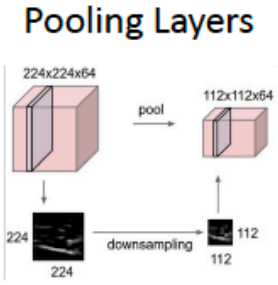
- **Components of a Convolutional Network**

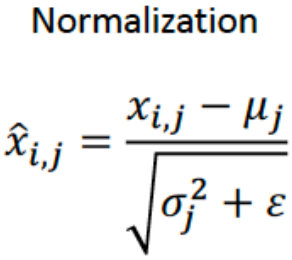
  --> Define new computational nodes that operate on images --> respect the spatial structure of images









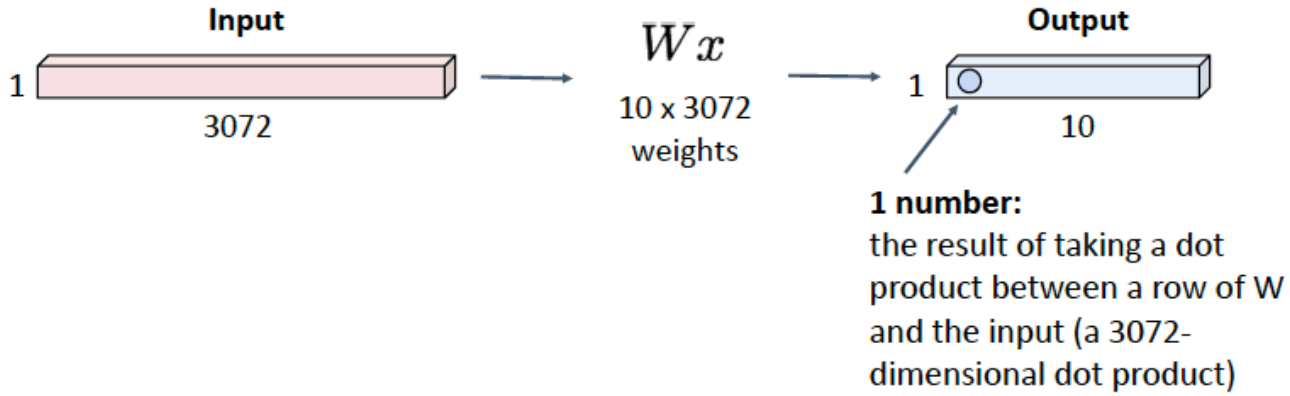


2. Components of a Convolutional Network

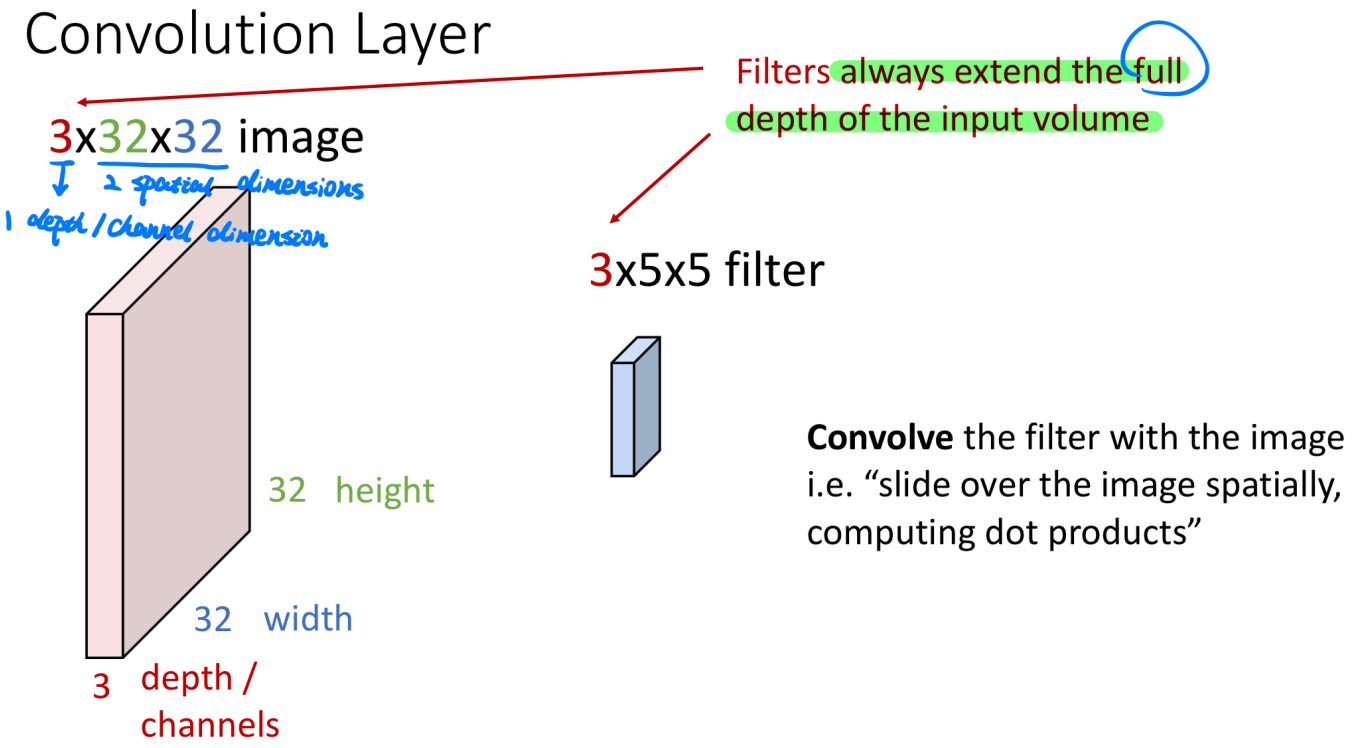
- **Fully-Connected Layer**

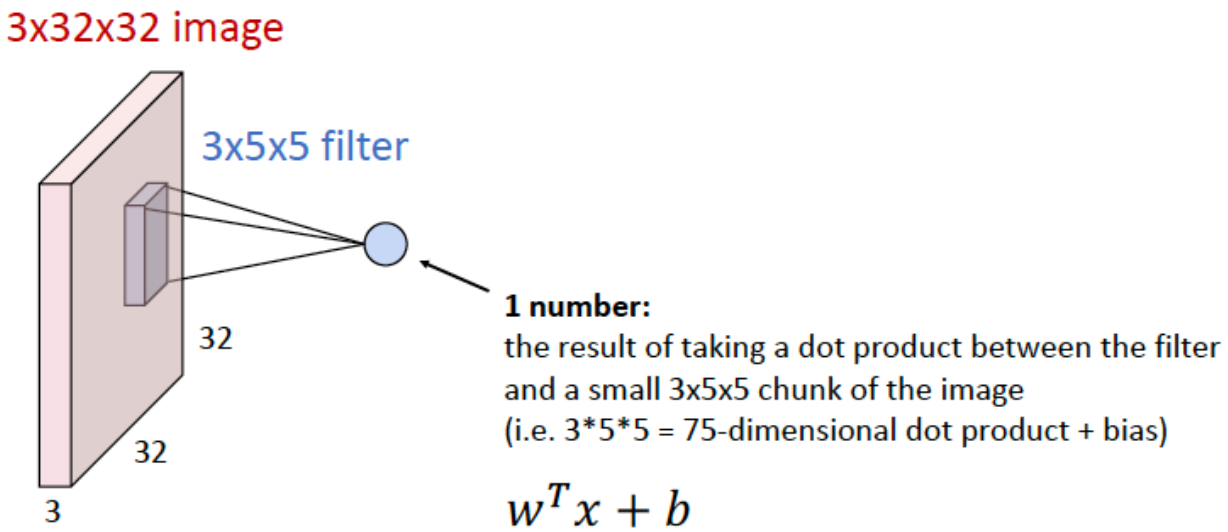
  Input: Image height × Image width × Channel number

32x32x3 image -> stretch to 3072 x 1

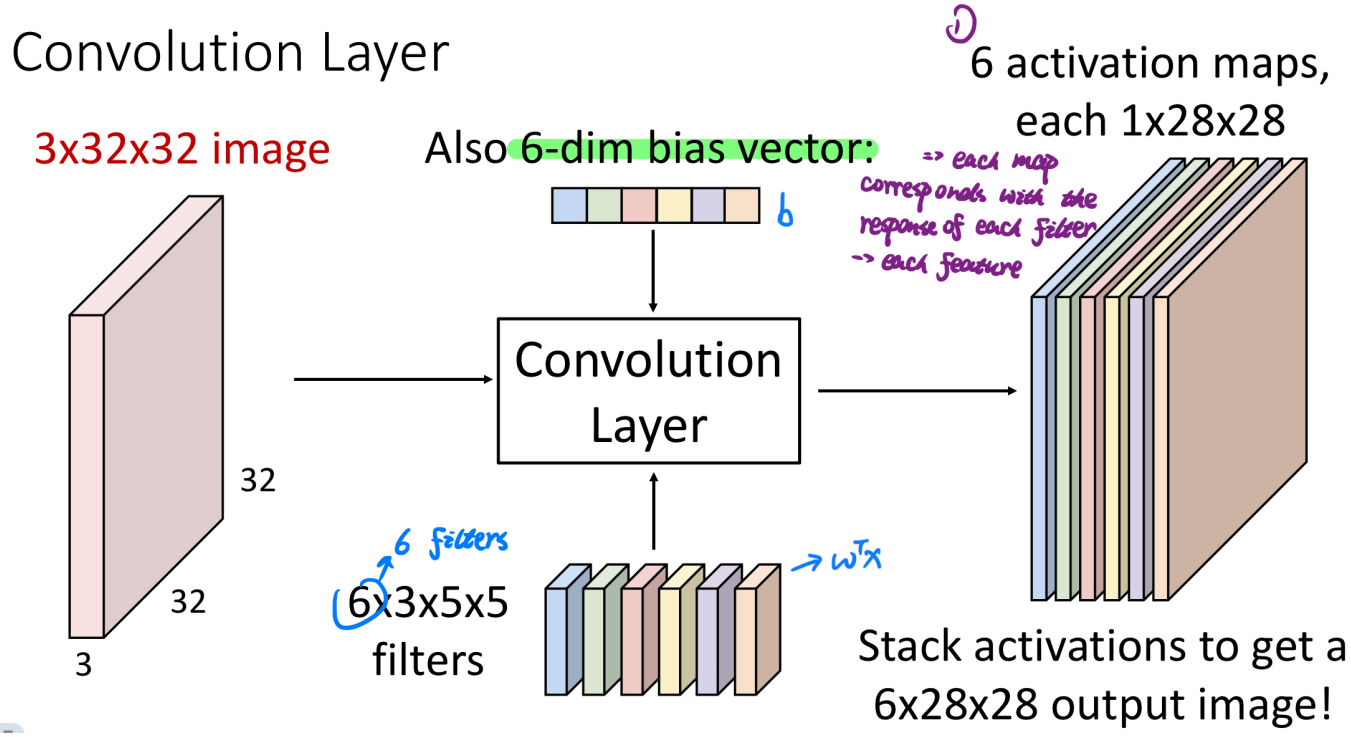


- **Convolutional Layer**



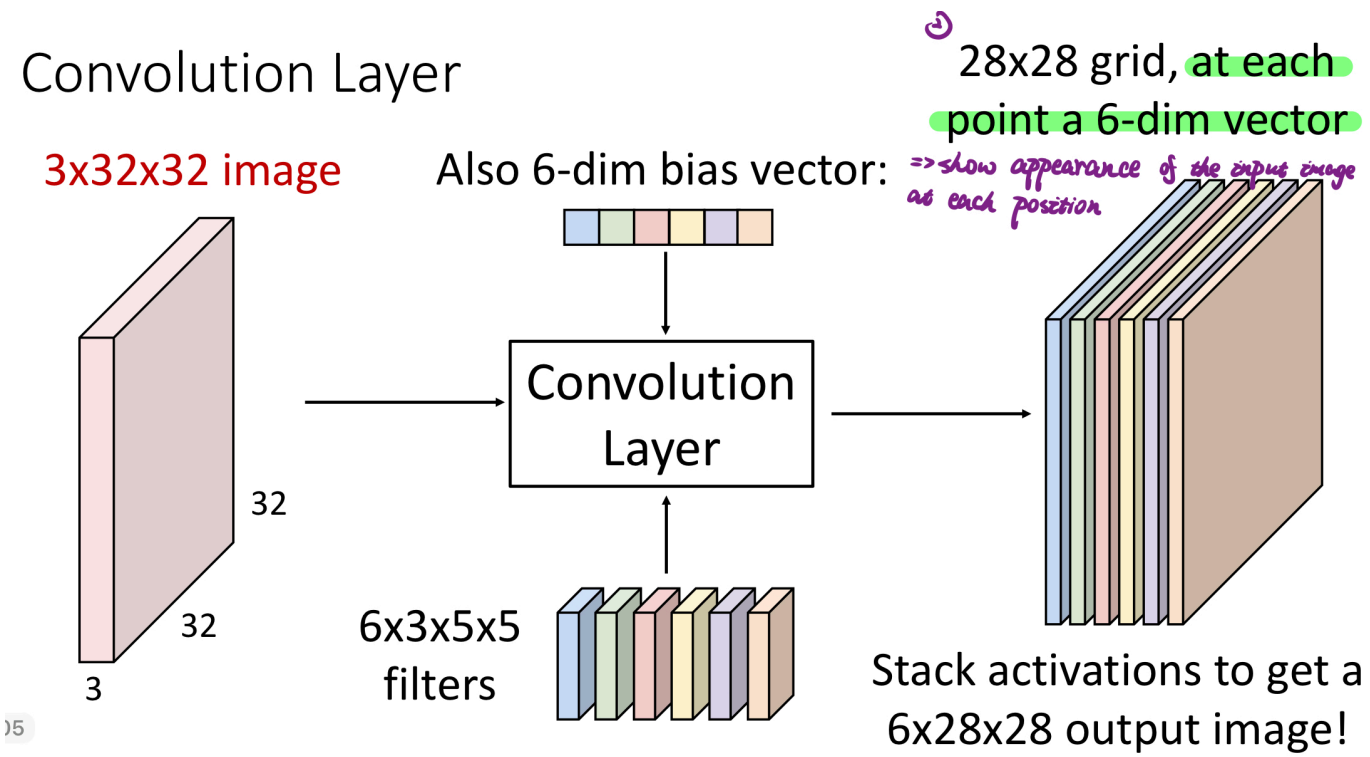


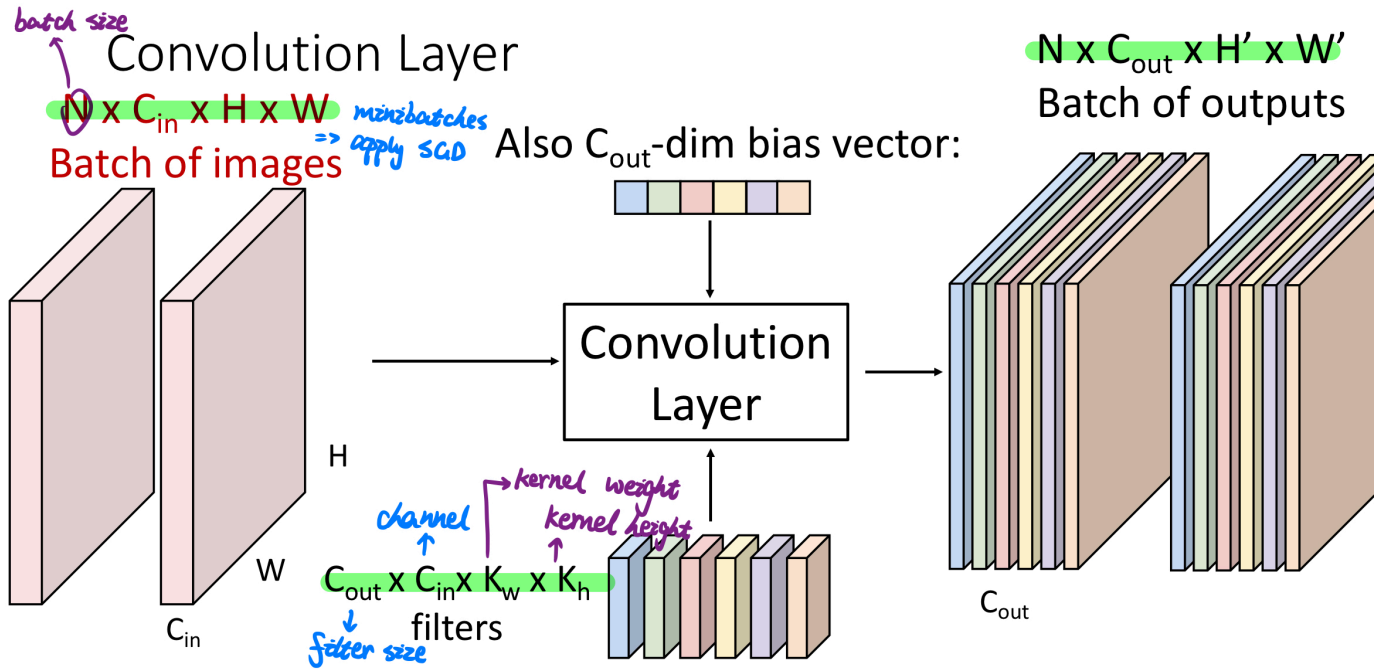
Convolution Layer

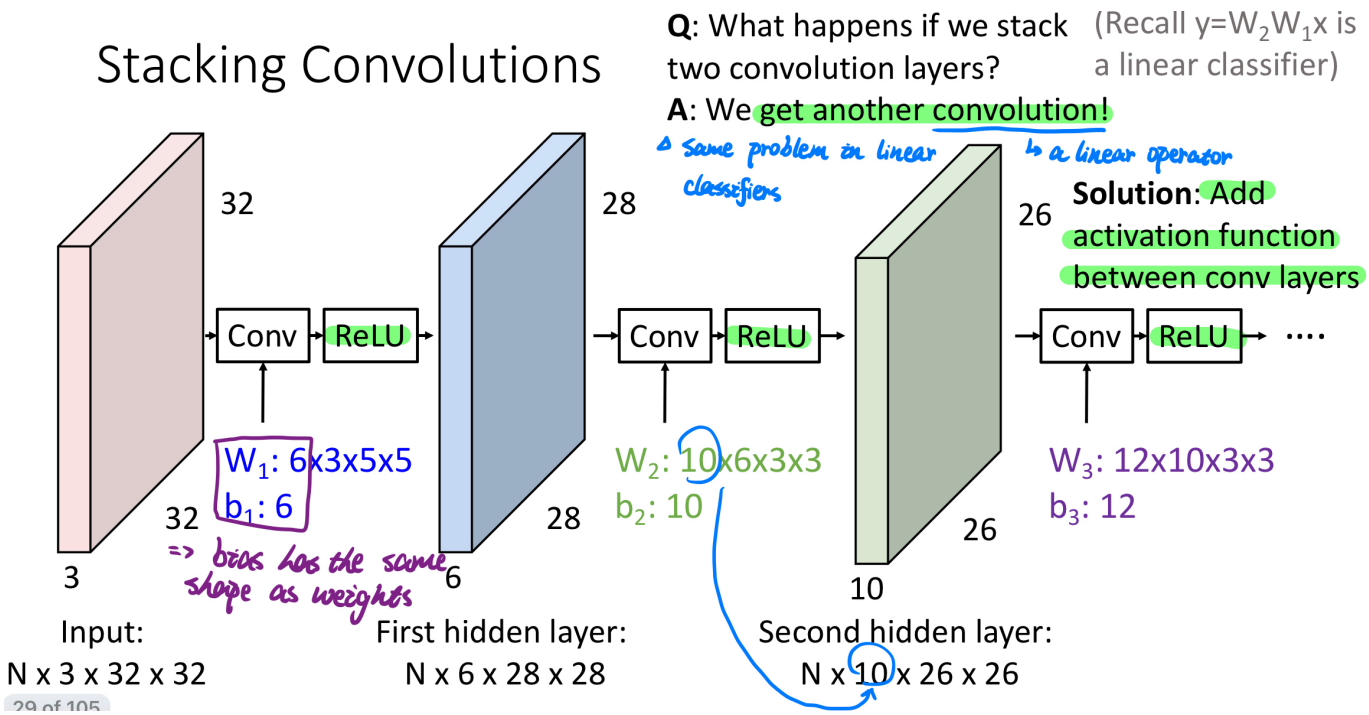


--> One filter is a template for extracting one feature

Convolution Layer

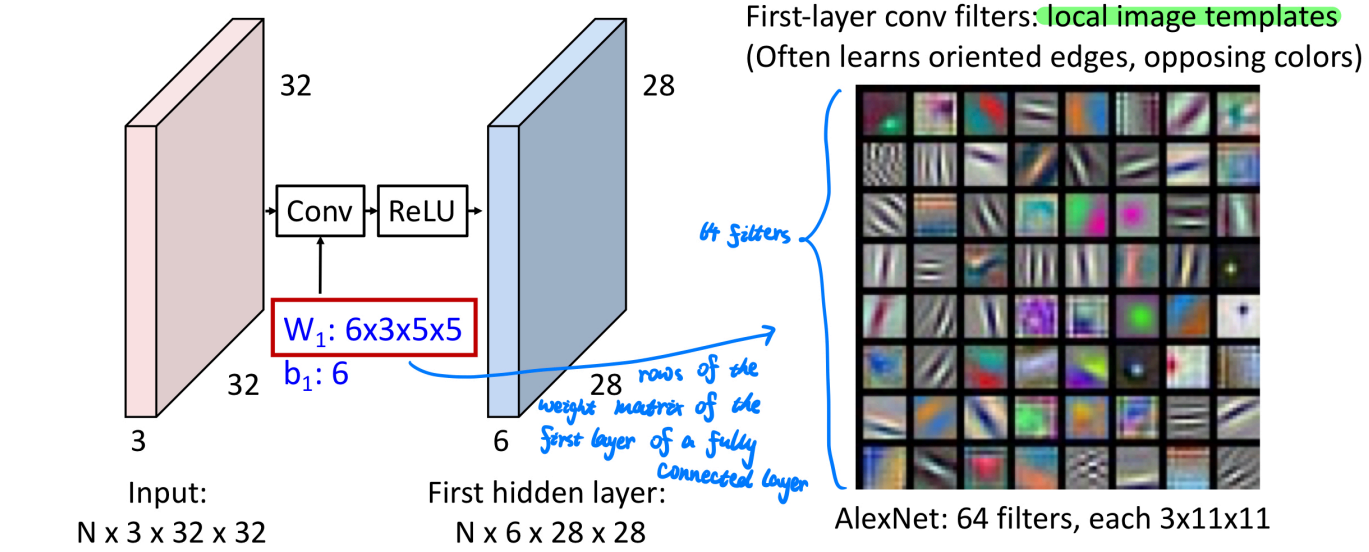






29 of 105

- Convolutional Filters



- Spatial Dimensions

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input: 7x7
Filter: 3x3
Output: 5x5

In general:
Input: W
Filter: K
Padding: P
Output: W – K + 1 + 2P

Very common:
Set P = (K – 1) / 2 to make output have same size as input!
[Same padding]

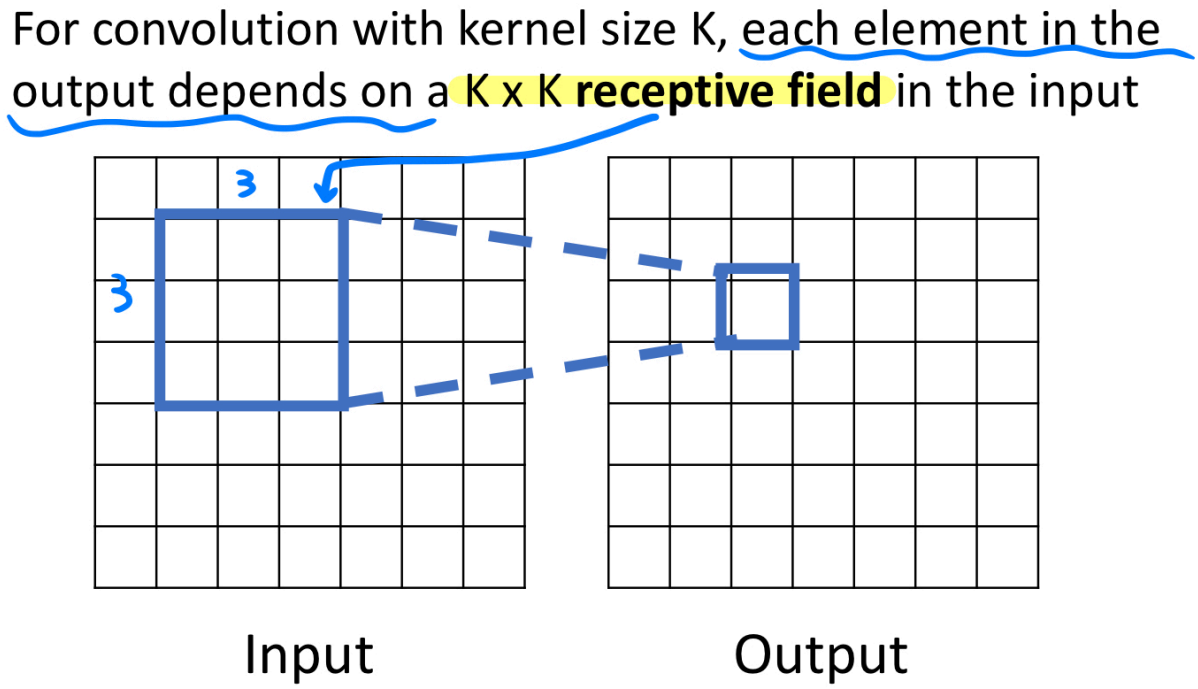
(W, K, P are all sizes)

- Receptive Fields
  - Definition:

    When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, we will **connect each neuron to only a local region of the input volume**.

    The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron (equivalently this is the **filter size**).

  - In the input:

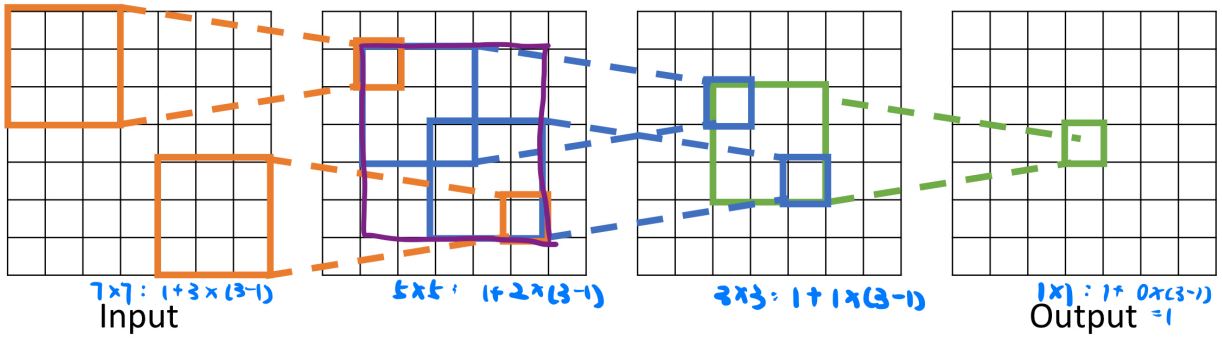


  - In the previous layers:

Receptive Fields

k: filter size, in this case k=3

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)

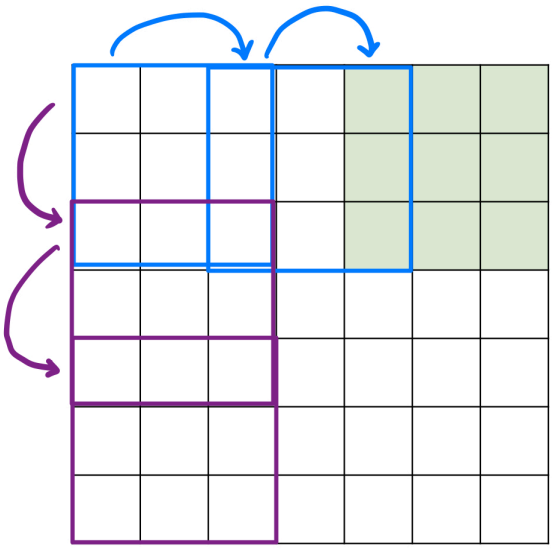


--> Given kernel width k and stride s, for m adjacent pixels in the activation output, **cummulative receptive field n** with respect to layer input is $n = k + s(m-1)$

- Problem

  For large images, many layer for each output to "see" the whole image image --> result in computational inefficiency --> Solution: **Downsample** inside the network --> *Strided Convolution*

- Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

Output: 3x3

In general:
Input: W
Filter: K
Padding: P
Stride: S
Output: (W – K + 2P) / S + 1

Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

(W-K+2P)/S+1

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
10 x 32 x 32

Number of learnable parameters: 760

Number of learnable parameters: **760**

Parameters per filter: 3*5*5 + 1 (for bias) = 76

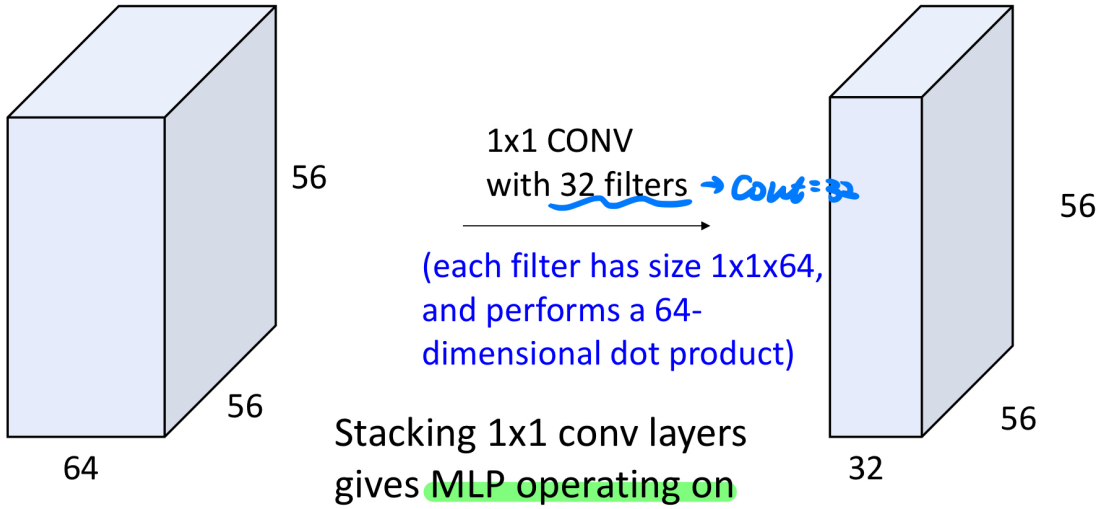
10 filters, so total is 10 * 76 = **760**

Number of multiply-add operations: **768,000**

10*32*32 = 10,240 outputs; each output is the inner product of two 3x5x5 tensors (75 elems); total = 75*10240 = **768K**

- 1x1 convolution

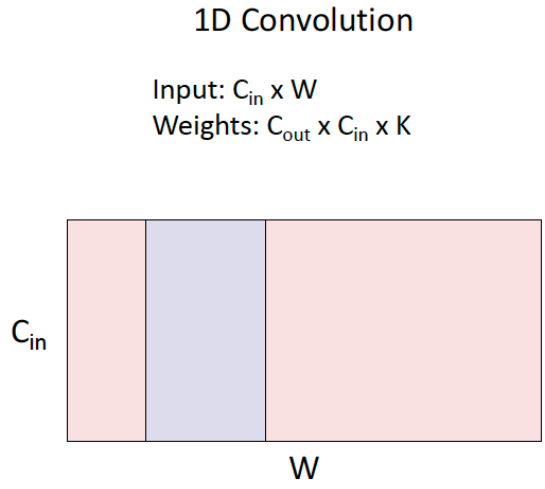
  --> implement 3-dimensional dot products

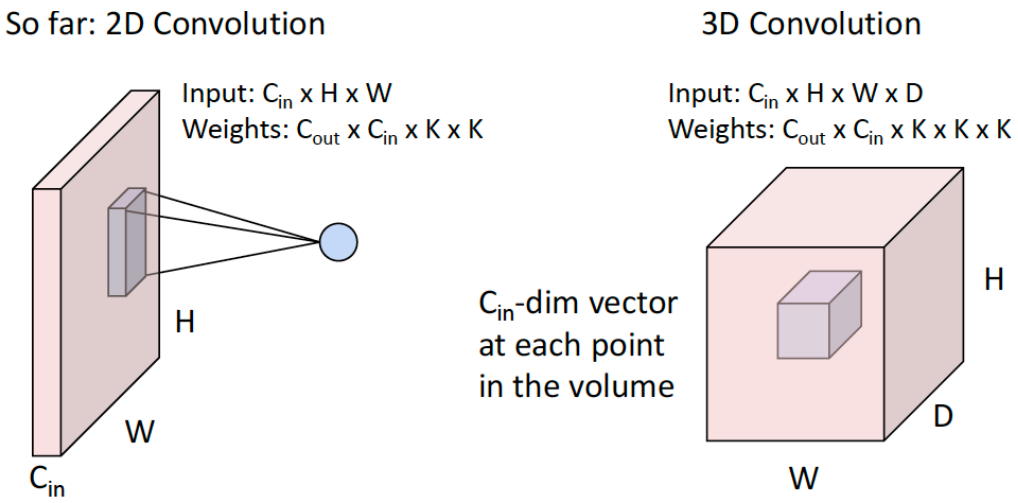


Stacking 1x1 conv layers gives MLP operating on each input position

Lin et al, "Network in Network", ICLR 2014

- 1D / 3D convolution





- Summary

Input: C_{in} x H x W
Hyperparameters:
- **Kernel size**: K_H x K_W
- **Number filters**: C_{out}
- **Padding**: P
- **Stride**: S

Weight matrix: C_{out} x C_{in} x K_H x K_W
giving C_{out} filters of size C_{in} x K_H x K_W
Bias vector: C_{out}
Output size: C_{out} x H' x W' where:
- H' = (H – K + 2P) / S + 1
- W' = (W – K + 2P) / S + 1

Common settings:
$K_H = K_W$ (Small square filters)
P = (K – 1) / 2 ("Same" padding)
C_{in}, C_{out} = 32, 64, 128, 256 (powers of 2)
K = 3, P = 1, S = 1 (3x3 conv)
K = 5, P = 2, S = 1 (5x5 conv)
K = 1, P = 0, S = 1 (1x1 conv)
K = 3, P = 1, S = 2 (Downsample by 2)

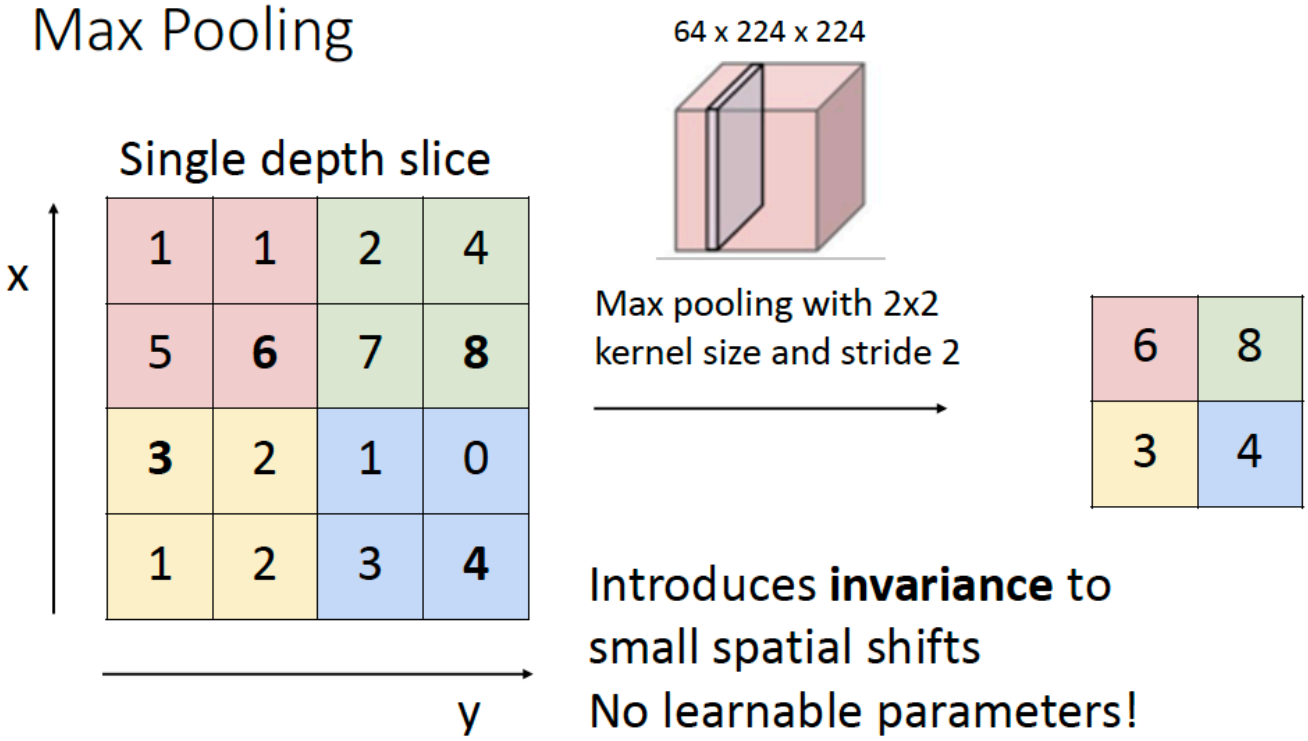
In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

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

- **Pooling Layers: Another way to downsample**
  - Properties:
    - No learnable parameters
    - Make the image smaller --> reduce computational complexity
    - Hyperparameters: **Kernel size, Stride, Pooling function**
    - Downsampling methods:
      - Pooling: cheap functions, easy to implement
      - Strided convolution: learnable downsample

- Strided convolution: learnable downsample

Max Pooling



Introduces **invariance** to small spatial shifts
No learnable parameters!

Pooling Summary

△ Downsample:
- pooling: deep functions, easy to implement
- strided convolution: learnable downsample

Input: C x H x W
Hyperparameters:
- Kernel size: K
- Stride: S
- Pooling function (max, avg)

Output: C x H' x W' where
- H' = (H – K) / S + 1
- W' = (W – K) / S + 1

Learnable parameters: None!

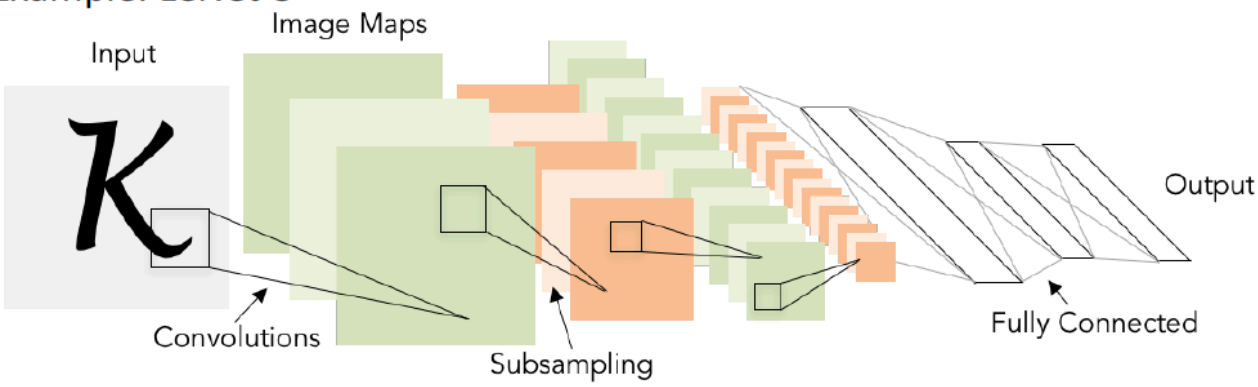
☆ Common settings:
max, K = 2, S = 2 (k = s)
max, K = 3, S = 2 (AlexNet)

△ average pooling (linear classifier)
-> a certain convolutional layer
=> can implement average pooling via convolution

- Convolutional Networks

Classic architecture: [Conv, ReLU, Pool] x N, flatten, [FC, ReLU] x N, FC

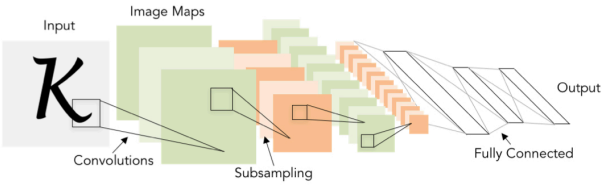
Example: LeNet-5



Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10

Lecun et al, "Gradient-based learning applied to document recognition", 1998



As we go through the network:

Spatial size **decreases** (using pooling or strided conv)

Number of channels **increases** (total "volume" is preserved!)
=> more learnable parameters
=> learn more complex patterns

Some modern architectures break this trend -- stay tuned!

- **Batch Normalization**
  - Problem: deep networks very hard to train! --> as input matrix can have large values as well as neglible values --> want to process more smoothly

Batch Normalization

Idea: "Normalize" the outputs of a layer so they have zero mean and unit variance

Why? Helps reduce "internal covariate shift", improves optimization

=> add an extra normalization layer

We can normalize a batch of activations like this:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Problem: Estimates depend on minibatch; can't do this at test-time!

=> can't calculate the parameters

=> may lose information across mini-batch

[At training time]

Input: $x \in \mathbb{R}^{N \times D}$ (D: channel)

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^D$$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$ Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$ Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

$\mu_j^{test} = 0$

For each training iteration:

$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$

$\mu_j^{test} = 0.99\,\mu_j^{test} + 0.01\,\mu_j$

(Similar for σ)

μ_j: (Running) average of values seen during training

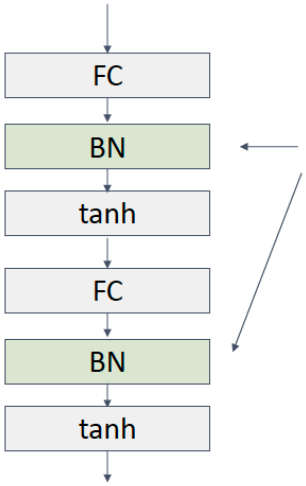
σ_j^2: (Running) average of values seen during training

- Batch Normalization during test-time

- Batch Normalization during test time

  --> become a **linear operator** and can be fused with the previous fully-connected or conv layer --> no runtime during test time

- Properties

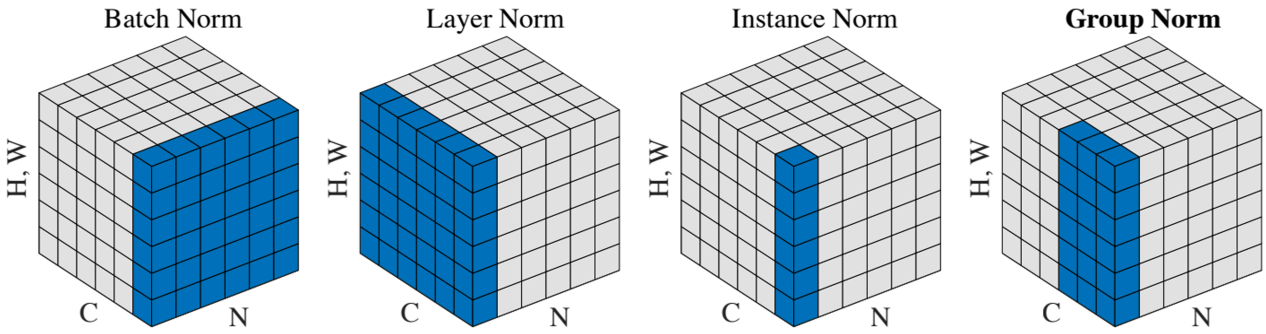


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a very common source of bugs!

- Comparison of Normalization Layers



Layer Normalization

Layer Normalization

Batch Normalization for **fully-connected** networks

no interaction along the elements of the mini-batch

Layer Normalization for fully-connected networks

Same behavior at train and test!

Used in RNNs, Transformers

cannot fuse into FC/Conv

$$x : N \times D$$

Normalize ↓

$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

$$x : N \times D$$

Normalize ↓

$$\mu, \sigma : N \times 1$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Batch Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize ↓ ↓ ↓

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$

Instance Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize ↓ ↓ ↓

$$\mu, \sigma : N \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma}\gamma + \beta$$