

# Convolutional Neural Networks, Backprop and Architectures

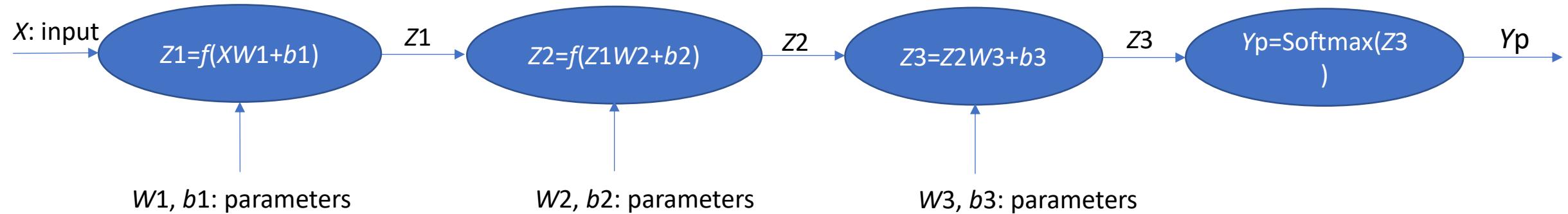
Nilanjan Ray

# Going from fully connected net to CNN

## Review of Fully Connected Networks (FCN):

- Images are flattened into row vectors; a batch becomes a matrix of shape (batch size  $\times$  number of pixels).
- This matrix is multiplied by a parameter matrix, a bias vector is added, and an activation function is applied  $\rightarrow$  the first hidden layer.
- Additional hidden layers follow the same structure, differing only in parameter and bias shapes.
- The last hidden layer usually has no activation.
- Its output goes into the softmax function.
- During training, the final node is the loss function, which is discarded during deployment.

# Going from FCN to CNN...



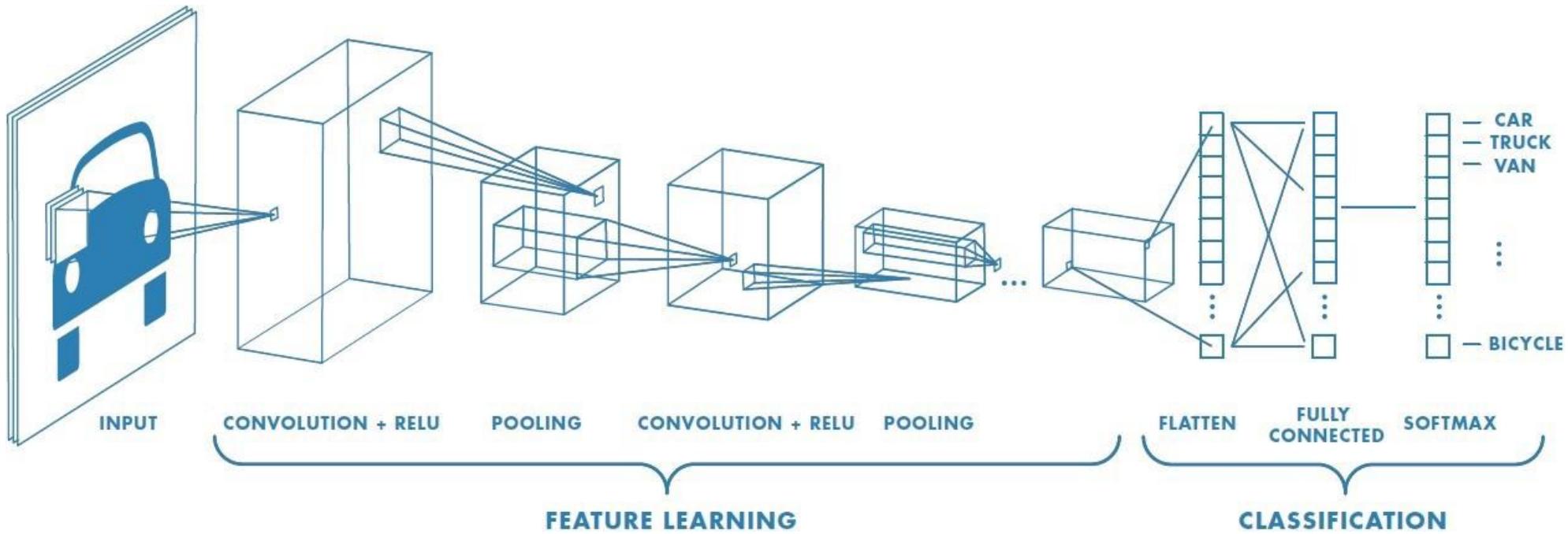
**A fully connected neural net**

Note:  $X$  is of shape (batch size x number of pixels), a batch size = 1 is perfectly admissible

# Going from FCN to CNN...

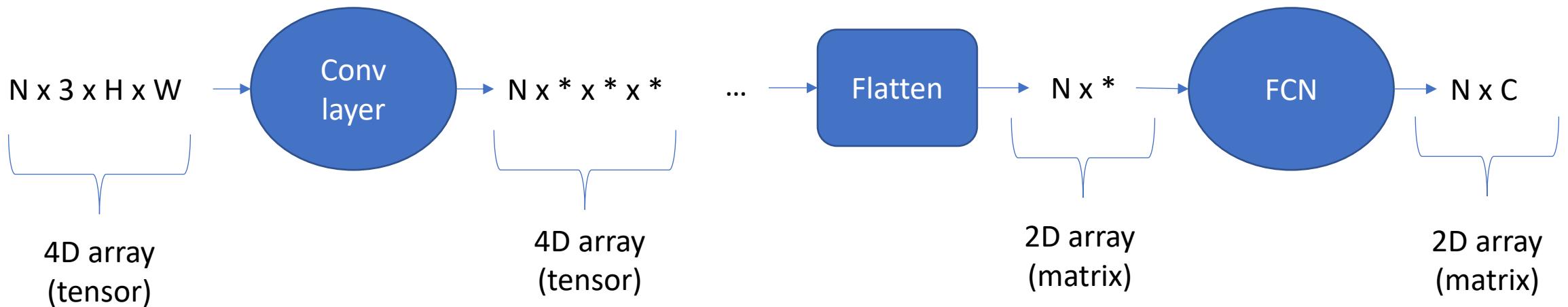
- In a CNN, **we do not flatten the input image at the start** — this is the first difference from an FCN.
- For now, assume batch size = 1 (we will generalize later).
- The **first hidden layer is a convolutional layer**. It takes an input RGB image (3D array), applies convolution (to be defined shortly after), and produces another 3D array. An activation function is applied element-wise.
- CNNs can also include **other layers, such as pooling**, which also map a 3D array to another 3D array.
- **At the end of a CNN, a fully connected network (FCN) may be attached**. Since an FCN expects a matrix/vector, the final 3D array is reshaped (flattened) into a vector before being passed to the FCN.

# Going from FCN to CNN (batch size = 1)...



Picture source: <https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html>

# Going from FCN to CNN (batch size = N)



N: batch size, C: number of classes

# Let's talk about convolution

- What is a convolution operation?
  - Let's work with a small numerical example

$$\begin{array}{|c|c|c|c|c|c|} \hline 1 & 3 & 2 & 4 & 6 & 4 \\ \hline 4 & 8 & 3 & 1 & 0 & 2 \\ \hline 2 & 1 & 4 & 3 & 9 & 1 \\ \hline 4 & 7 & 2 & 3 & 9 & 2 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array} =$$

I (feature map)

K (convolution kernel  
aka filter matrix)

J (output)

- <https://cs231n.github.io/convolutional-networks/>

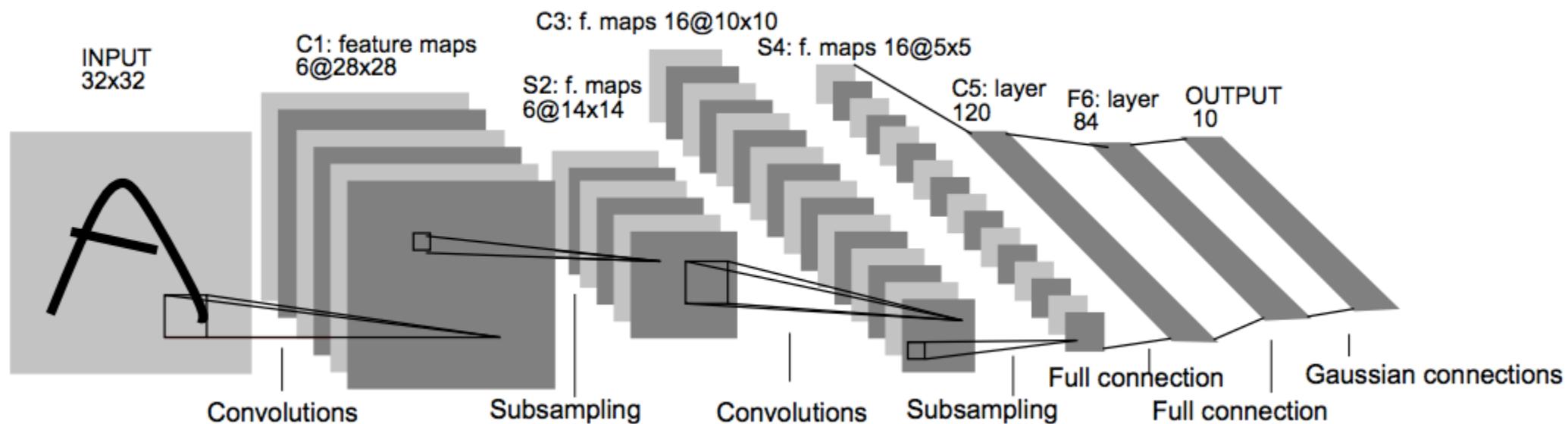
# Practice questions

- Input layer shape:  $1 \times 3 \times 32 \times 32$ , two filters, each of shape:  $3 \times 5 \times 5$ . Assume no padding. Assume stride 1. What is the output shape?
  - $1 \times 2 \times 28 \times 28$
- How many learnable parameters?
  - In each filter we have  $3 \times 5 \times 5 + 1 = 76$  parameters. In two filters we have  $2 \times 76 = 152$  parameters. We added 1 because we assumed a bias term is present in each filter.
- Now assume stride 2 convolution. What would be the output shape?
  - $1 \times 2 \times 14 \times 14$

# Common layers used in a CNN

- Convolution ([https://d2l.ai/chapter\\_convolutional-neural-networks/conv-layer.html](https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html))
- Pooling: average pooling, max pooling  
([https://d2l.ai/chapter\\_convolutional-neural-networks/pooling.html](https://d2l.ai/chapter_convolutional-neural-networks/pooling.html))

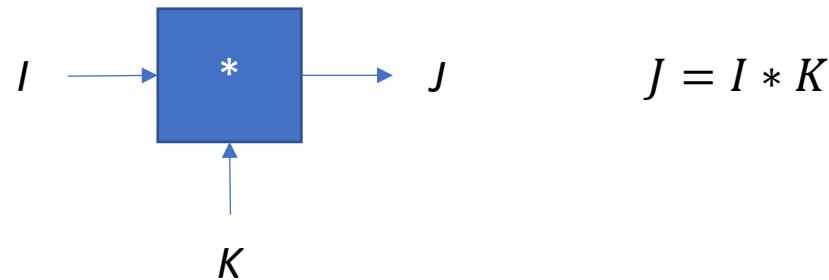
# Let's build our first convnet: LeNet



Note: Here “Gaussian connection” refers to a softmax function,  
subsampling is max pooling

Implementation in MNIST\_LeNet.ipynb

# Backpropagation (BP) for a conv layer



Input to convolution layer:  $I$ , a  $H$ -by- $W$  matrix

Parameter of the layer:  $K$ , a  $h$ -by- $w$  matrix

Output of the layer:  $J$ , a  $(H-h+1)$ -by- $(W-w+1)$  matrix

Assume:  $H \geq h$  and  $W \geq w$

Given the gradient of loss function  $\delta J$  with respect to  $J$ , BP tries to find answers to the following:

- (1) What is the gradient of the loss function with respect to  $K$ ? Denote this gradient by  $\delta K$ .
- (2) What is the gradient of the loss function with respect to  $I$ ? Denote this gradient by  $\delta I$ .

**Why do we need  $\delta K$ ?** Because, we want to adjust the parameter  $K$  by gradient descent:  $K = K - (\text{learning rate})\delta K$

**Why do we need  $\delta I$ ?** Because, we want to apply BP to the layer that precedes this conv layer.

# Derivation of $\delta K$

$$J(i, j) = \sum_{l=1}^h \sum_{m=1}^w I(i + l - 1, j + m - 1) K(l, m) \quad \longrightarrow \quad \frac{\partial J(i, j)}{\partial K(p, q)} = I(i + p - 1, j + q - 1)$$

Using chain rule of derivative:

$$\delta K(p, q) = \sum_{i=1}^{H-h+1} \sum_{j=1}^{W-w+1} \frac{\partial J(i, j)}{\partial K(p, q)} \delta J(i, j) = \sum_{i=1}^{H-h+1} \sum_{j=1}^{W-w+1} I(i + p - 1, j + q - 1) \delta J(i, j)$$

Thus,  $\boxed{\delta K = I * \delta J}$

# Derivation of $\delta I$

$$J(i, j) = \sum_{l=1}^h \sum_{m=1}^w I(i + l - 1, j + m - 1) K(l, m) \quad \longrightarrow$$

$$\frac{\partial J(i, j)}{\partial I(p, q)} = \begin{cases} K(p - i + 1, q - j + 1), & \text{if } 0 \leq p - i \leq h - 1 \text{ and } 0 \leq q - j \leq w - 1, \\ 0, & \text{otherwise.} \end{cases}$$

Using chain rule of derivative:

$$\begin{aligned} \delta I(p, q) &= \sum_{i=1}^{H-h+1} \sum_{j=1}^{W-w+1} \frac{\partial J(i, j)}{\partial I(p, q)} \delta J(i, j) = \sum_{i=\max(1, p-h+1)}^{\min(p, H-h+1)} \sum_{j=\max(1, q-w+1)}^{\min(q, W-w+1)} K(p - i + 1, q - j + 1) \delta J(i, j) \\ &= \sum_{l=\max(1, p+h-H)}^{\min(p, h)} \sum_{m=\max(1, q+w-W)}^{\min(q, w)} K(l, m) \delta J(p - l + 1, q - m + 1) \end{aligned}$$

Thus,  $\delta I = \text{pad}(\delta J) * \text{flip}(K)$

“pad” function adds  $(h-1)$  0 rows at the top and bottom and also adds  $(w-1)$  0 columns at the left and at the right of a matrix.

Size of  $\text{pad}(\delta J)$  is  $(H+h-1)$ -by- $(W+w-1)$ .

$\text{flip}(K)$  is best understood by an example:

$$K = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \quad \text{flip}(K) = \begin{bmatrix} 6 & 4 & 2 \\ 5 & 3 & 1 \end{bmatrix}$$

# BP for a max pooling layer



Note that in case of a tie, only a single index is chosen for the following operation:

$$i = \operatorname{argmax}_k \{x_k\}_{k=1}^n$$

By chain rule:  $\delta x_i = \frac{\partial y}{\partial x_i} \delta y = \begin{cases} \delta y, & \text{if } i = \operatorname{argmax}_k \{x_k\}_{k=1}^n, \\ 0, & \text{otherwise.} \end{cases}$

Example of a 2-by-2, stride 2 max pooling:

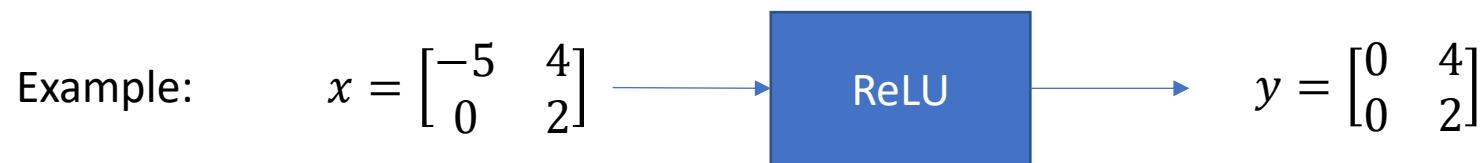
$$x = \begin{bmatrix} 2 & -3 & 6 & 5 \\ 4 & 5 & 6 & -7 \\ 0 & -1 & 3 & -4 \\ -2 & -6 & 6 & 8 \end{bmatrix} \rightarrow \text{Max pool} \rightarrow y = \begin{bmatrix} 5 & 6 \\ 0 & 8 \end{bmatrix}$$

Suppose,  $\delta y = \begin{bmatrix} \delta y_1 & \delta y_3 \\ \delta y_2 & \delta y_4 \end{bmatrix}$ ,  
then,  $\delta x = \begin{bmatrix} 0 & 0 & \delta y_3 & 0 \\ 0 & \delta y_1 & 0 & 0 \\ \delta y_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \delta y_4 \end{bmatrix}$ .

# BP for a ReLU layer

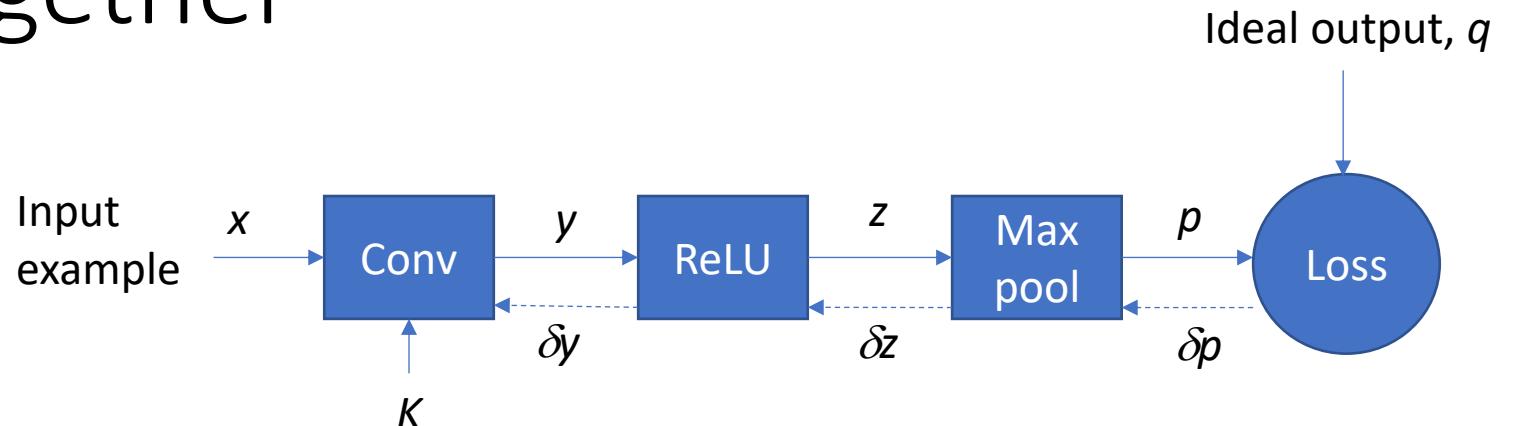


By chain rule:  $\delta x = \frac{\partial y}{\partial x} \delta y = \begin{cases} \delta y, & \text{if } x > 0, \\ 0, & \text{otherwise.} \end{cases}$



Suppose,  $\delta y = \begin{bmatrix} \delta y_1 & \delta y_3 \\ \delta y_2 & \delta y_4 \end{bmatrix}$ , then  $\delta x = \begin{bmatrix} 0 & \delta y_3 \\ 0 & \delta y_4 \end{bmatrix}$ .

# Putting it all together



## Convnet training algorithm:

Initialize parameter  $K$ .

Iterate:

Step 1: (Forward pass)

Step 1a: Randomly choose a training example  $x$  and its corresponding ideal output  $q$ .

Step 1b: Pass  $x$  through "Conv" to get  $y$ ; pass  $y$  through ReLU to get  $z$ ; pass  $z$  through Max pool to get  $p$ .

Step 2: Compute "Loss" function for diagnostic purposes. /\* Loss function measures deviation of  $p$  from  $q$ . \*/

Step 3: (Backward pass aka backpropagation)

Step 3a: Compute gradient of Loss function with respect to  $p$ . Denote this gradient by  $\delta p$ .

Step 3b: Compute  $\delta z$  given  $\delta p$ . /\* Look at "BP for Max pooling." \*/

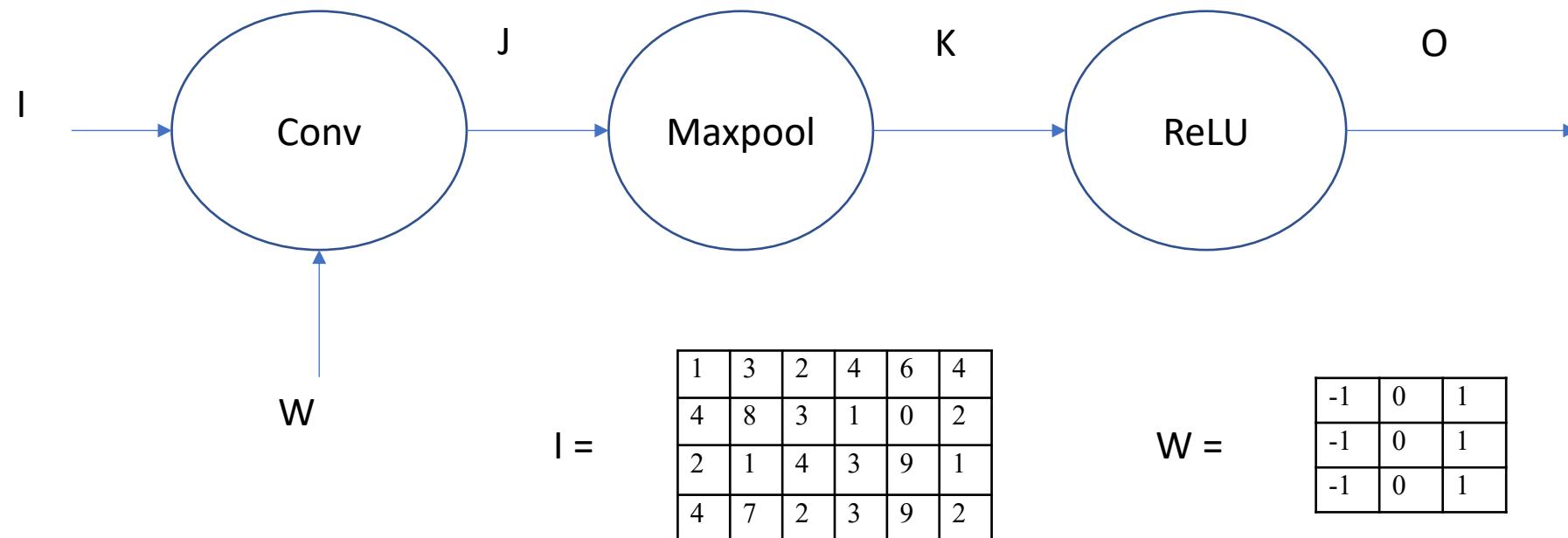
Step 3c: Compute  $\delta y$  given  $\delta z$ . /\* Look at "BP for ReLU." \*/

Step 4c: Compute  $\delta K$  given  $\delta y$ . /\* Look at "BP for Conv." \*/

Step 4: (Update parameter  $K$  by gradient descent)  $K = K - (\text{learning rate}) \delta K$ .

Note: We don't have to compute  $\delta x$ , because there is no layer preceding conv layer in the example above.<sup>16</sup>

# Example w/PyTorch



Compute  $J$ ,  $K$  and  $O$ . You are not doing any zero padding while doing the convolution. Assume stride size 1 for the convolution. Assume a 2-by-2 max pooling with stride 2. Write  $J$ ,  $K$  and  $O$  below.

Now Assume that ideal output  $IO = [7, 10]$ . Also assume the loss as 0.5 times the square of Euclidean distance between  $IO$  and  $O$ . Compute back-propagation for  $O$ ,  $K$ ,  $J$  and  $W$ . Verify using pytorch autograd!

## Example 2

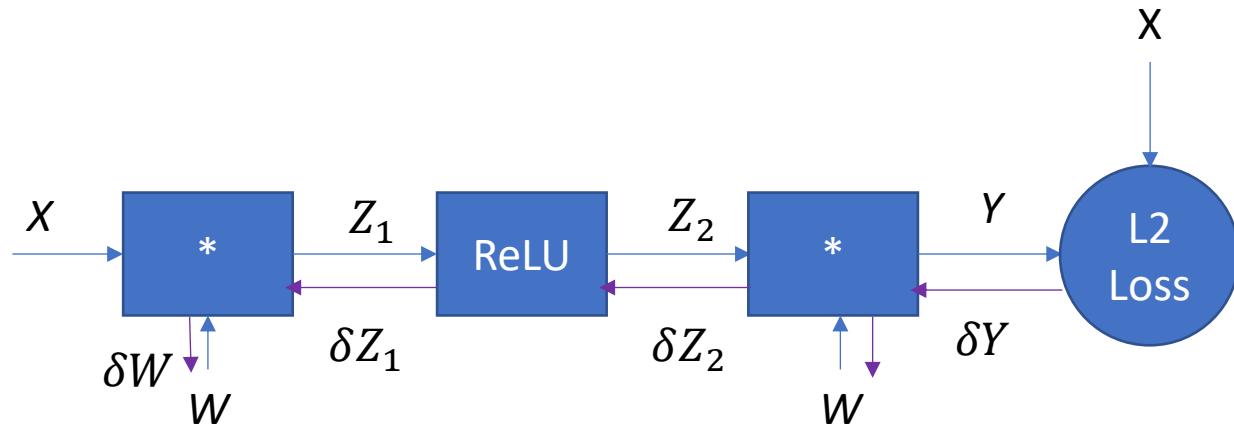
Consider a fully connected neural net:

$$Y = \text{ReLU}(X * W) * W$$

The loss is L2 between Y and X. Compute the gradient of the loss with respect to W. Here '\*' refers to matrix multiplication.

Note that the same parameter matrix W appears in two computational nodes – called shared parameters/weights.

## Example 2 solution



$$\delta Y = Y - X$$

$$\delta Z_2 = \delta Y * W^T$$

$$\delta Z_1 = ReLU'(Z_1) * \delta Z_2$$

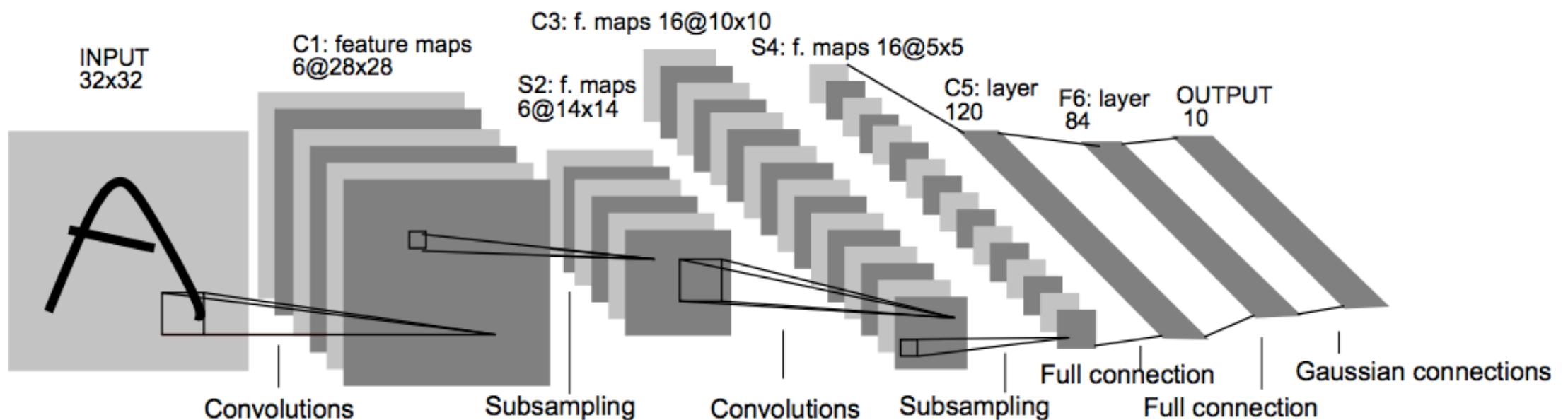
$$\delta W = Z_2^T * \delta Y + X^T * \delta Z_1$$

Let's talk about CNN architectures

# Different types of layers in a CNN

- A convolutional neural network consists of several types of layers / computational nodes.
  - Convolution layer (we have seen this, we will see some variations later)
  - Pooling (we have seen this)
  - Activation functions (we have seen it, but will examine more of them)
  - Fully connected net (we have seen this)
  - Dropout layer (new, we will learn about it)
  - Normalization layers (new, we will examine them)
- We will also learn about a few important types of connections
  - Residual
  - Dense

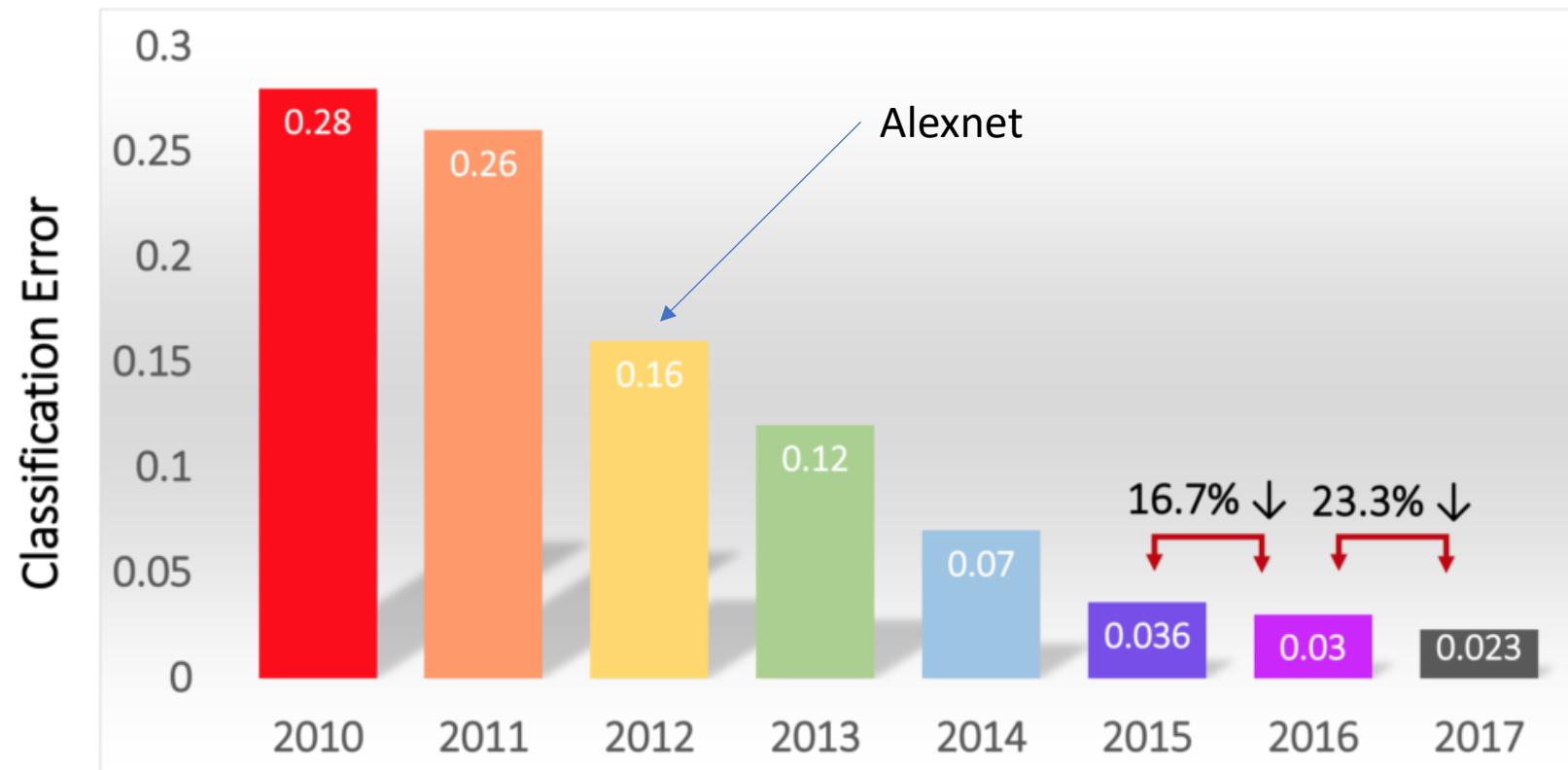
# LeNet: Review



# Large-scale image classification problem

## Classification Results (CLS)

One million images  
One thousand classes



# 2012: AlexNet

## Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

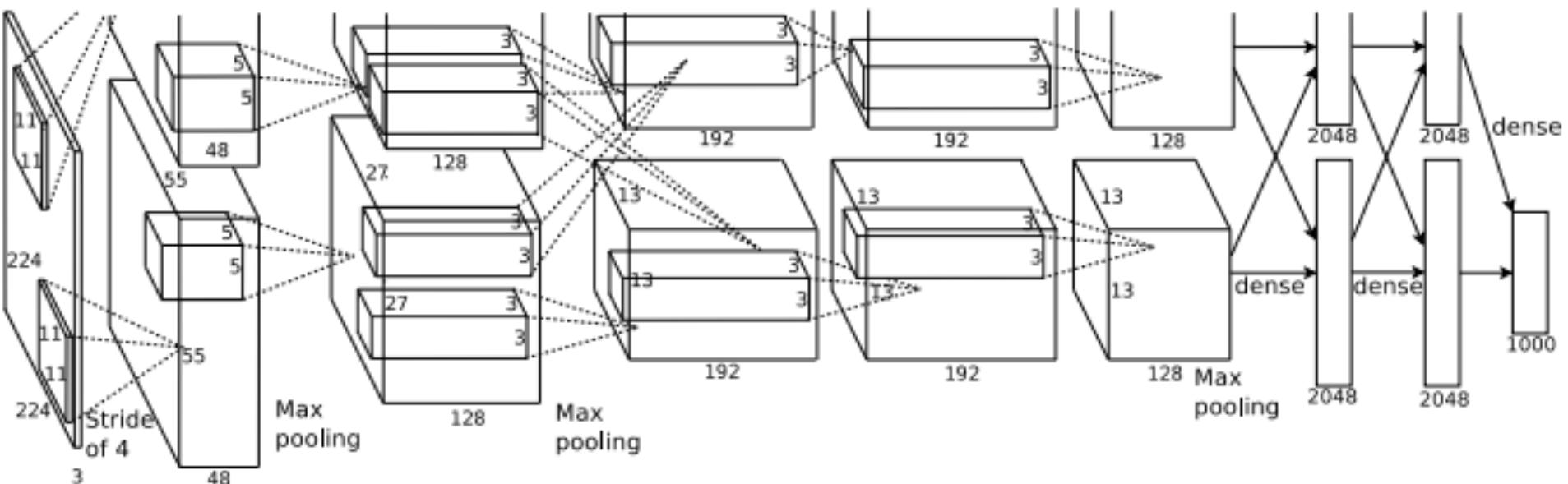
CONV5

Max POOL3

FC6

FC7

FC8



From U Toronto (Jeff Hinton's team), 60M parameters, 8 layers, first use of ReLU as activation function

# AlexNet...

- Introduced several new ideas
  - ReLU activation function
  - Overlapping pooling
  - Data augmentation
  - Dropout
  - Training with multiple GPUs

A simplified implementation:

[https://d2l.ai/chapter\\_convolutional-modern/alexnet.html](https://d2l.ai/chapter_convolutional-modern/alexnet.html)

AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

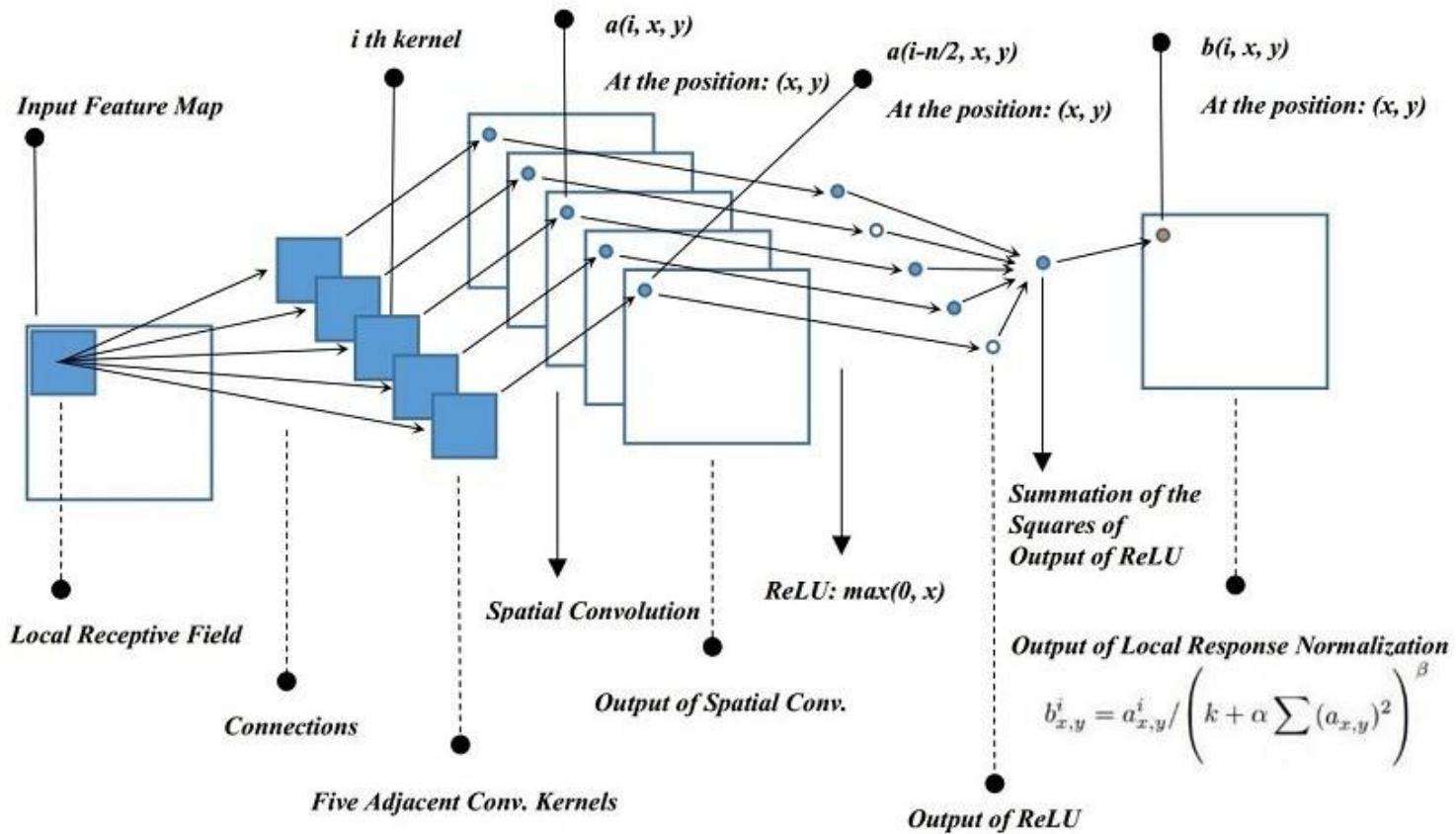
[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

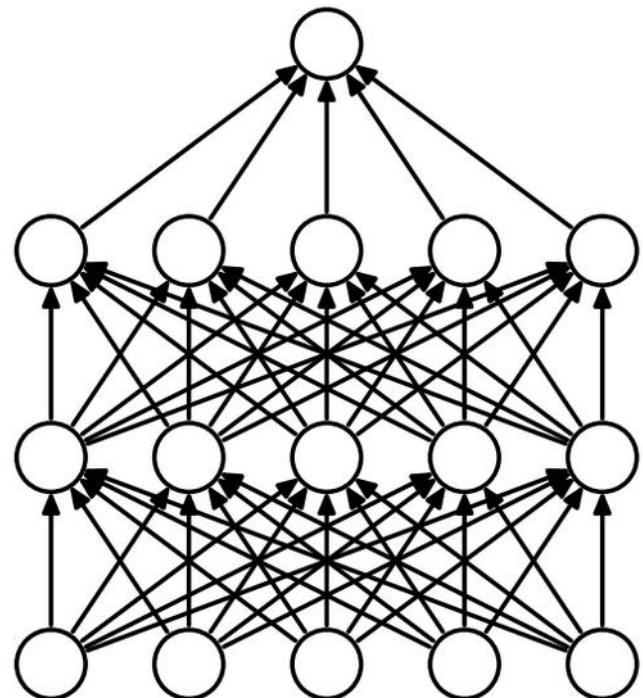
[1000] FC8: 1000 neurons (class scores)

# Normalization layer in AlexNet

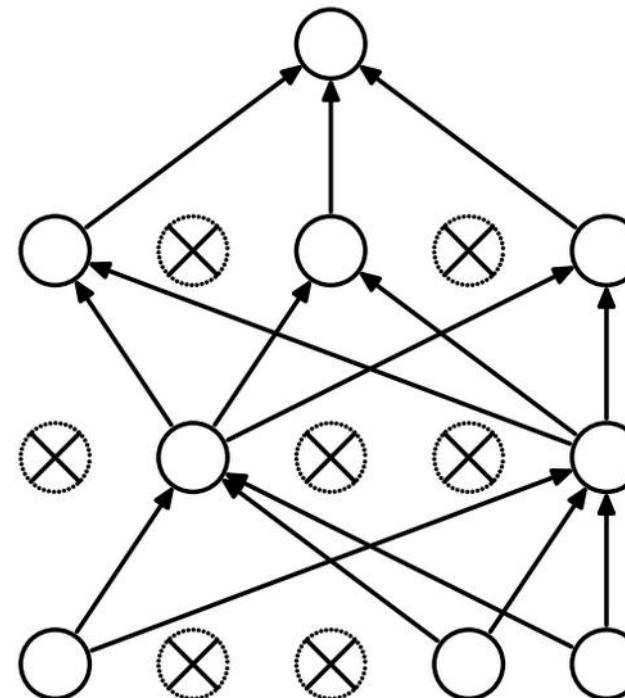


Source: <https://www.quora.com/What-is-Local-Response-Normalization-and-why-does-AlexNet-utilize-that-instead-of-any-other-type-of-normalization>

# Dropout



(a) Standard Neural Net



(b) After applying dropout.

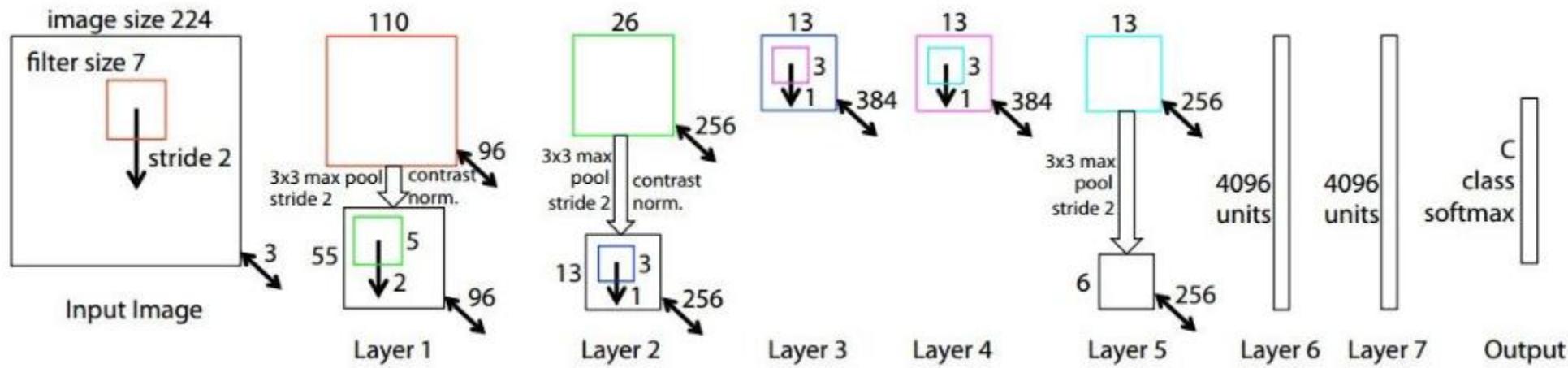
Pytorch tutorial: [https://xuwd11.github.io/Dropout\\_Tutorial\\_in\\_PyTorch/](https://xuwd11.github.io/Dropout_Tutorial_in_PyTorch/)

# Batchnorm layer

[https://d2l.ai/chapter\\_convolutional-modern/batch-norm.html](https://d2l.ai/chapter_convolutional-modern/batch-norm.html)

<https://arxiv.org/pdf/1502.03167.pdf>

# 2013: ZF Net



Changes from AlexNet:

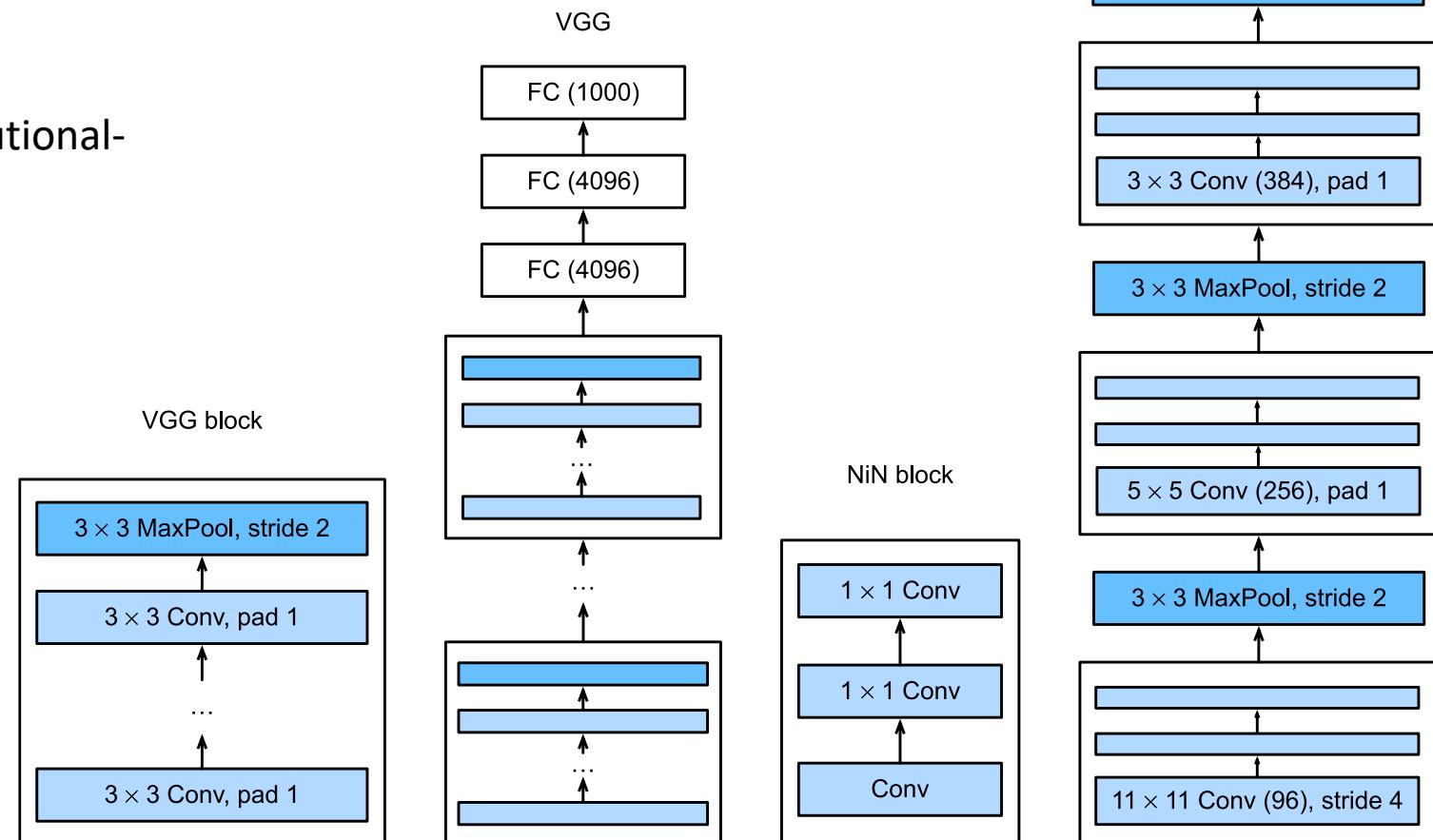
CONV1: change from (11x11 stride 4) to (7x7 stride 2)

CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

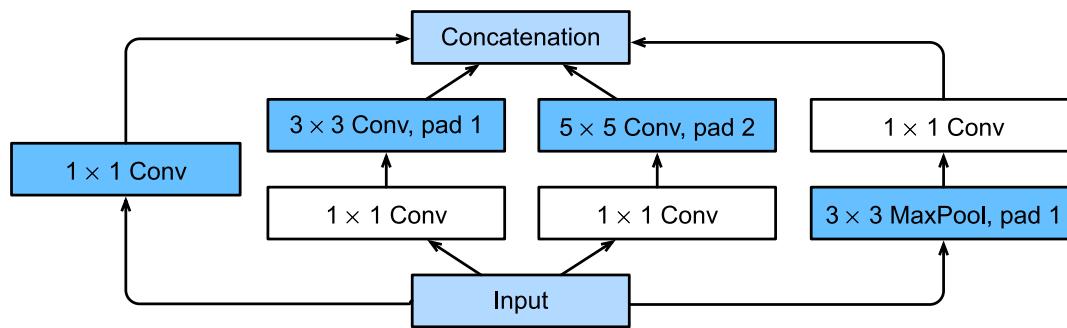
# 2013: Network-In-Network

Implementation:

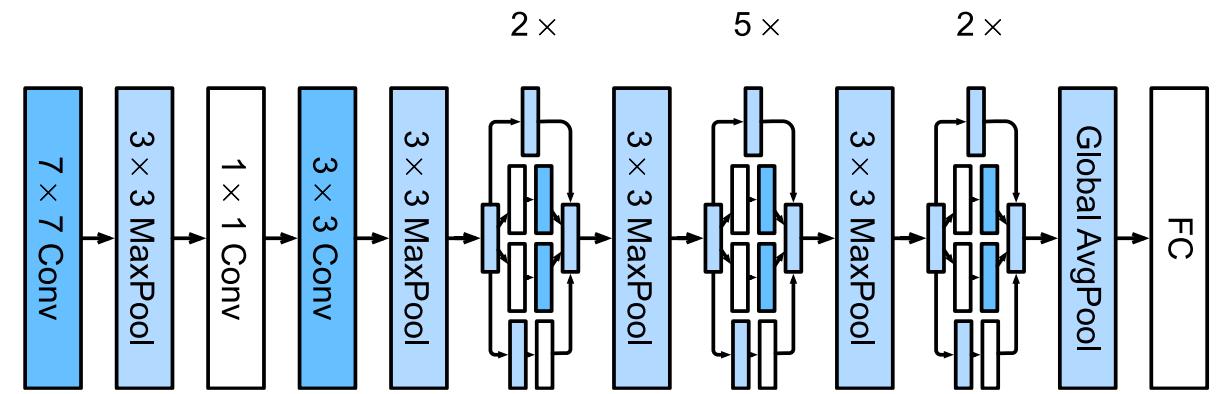
[https://d2l.ai/chapter\\_convolutional-modern/nin.html](https://d2l.ai/chapter_convolutional-modern/nin.html)



# 2014: GoogLeNet - inception block



Inception block



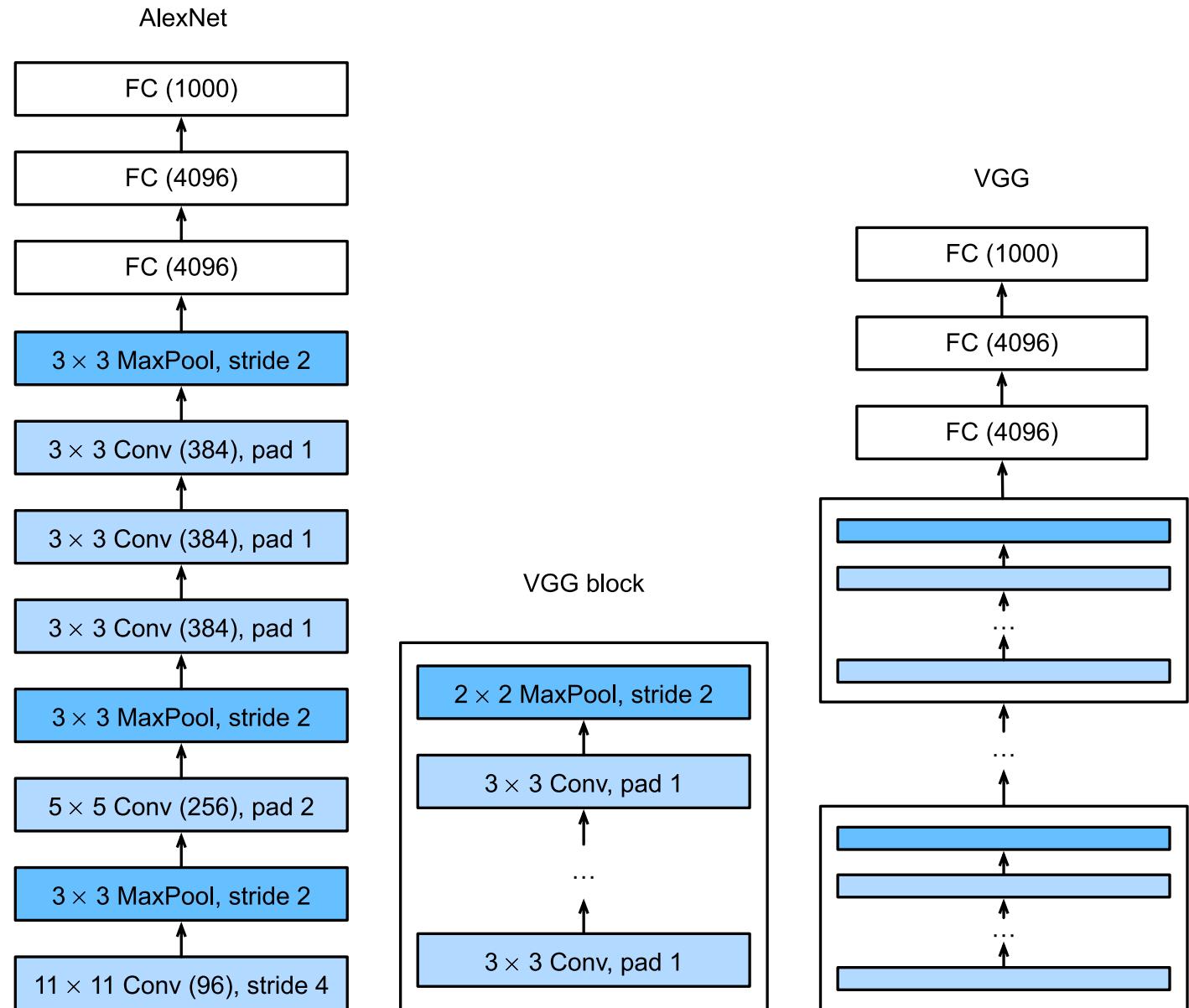
GoogLeNet architecture

Let's take a look at its implementation

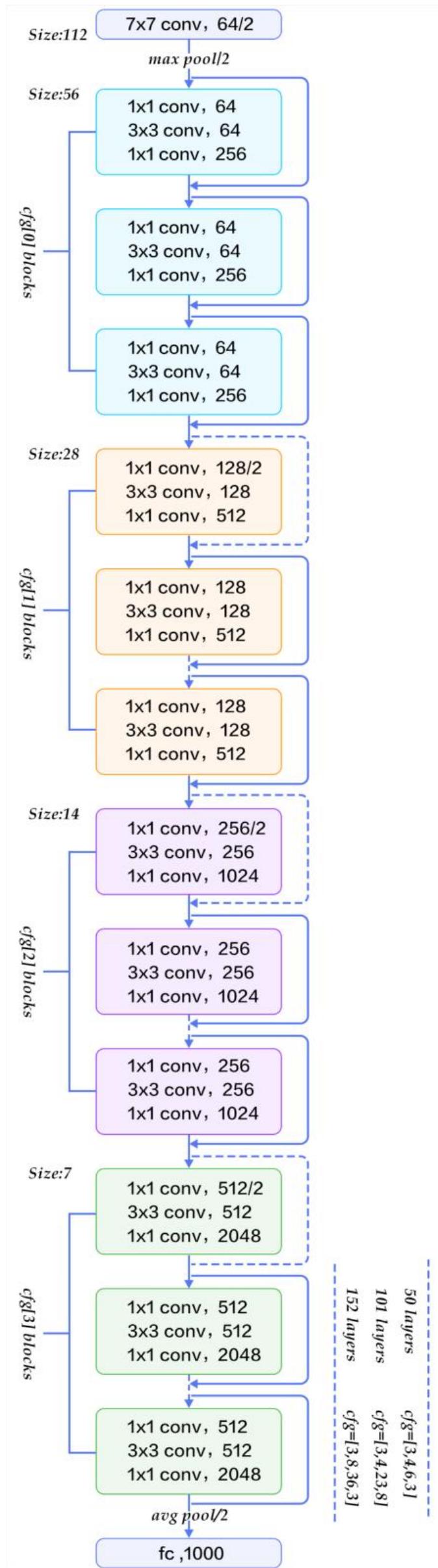
[https://d2l.ai/chapter\\_convolutional-modern/googlenet.html](https://d2l.ai/chapter_convolutional-modern/googlenet.html)

# 2014: VGGNet

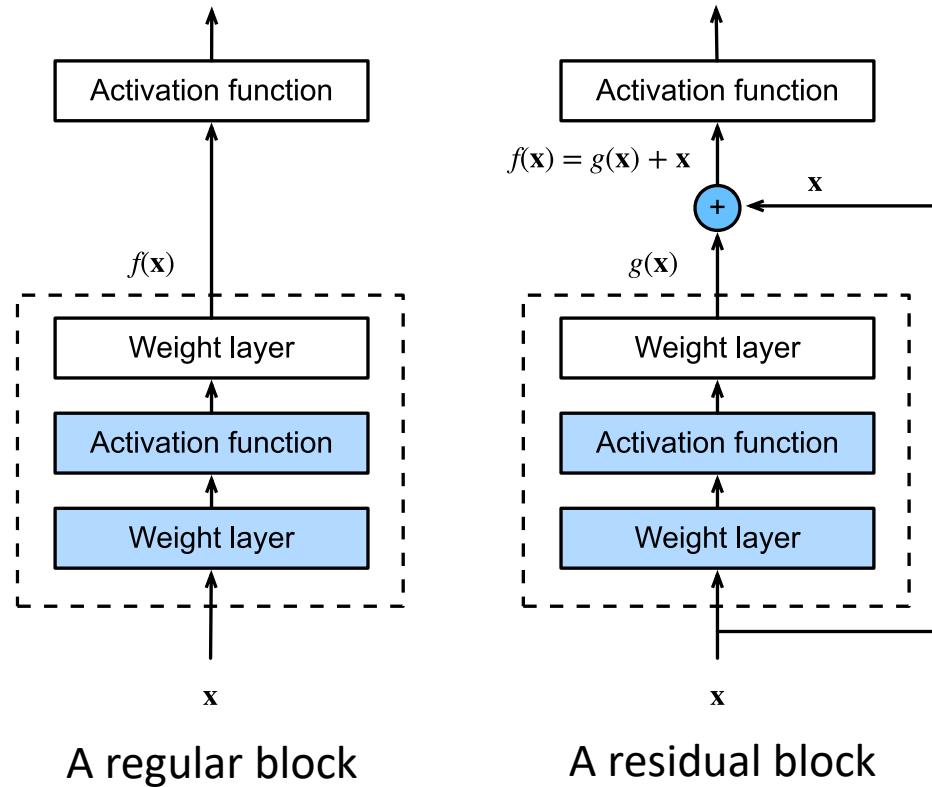
Let's take a look at its implementation:  
[https://d2l.ai/chapter\\_convolutional-modern/vgg.html](https://d2l.ai/chapter_convolutional-modern/vgg.html)



# 2015: ResNet

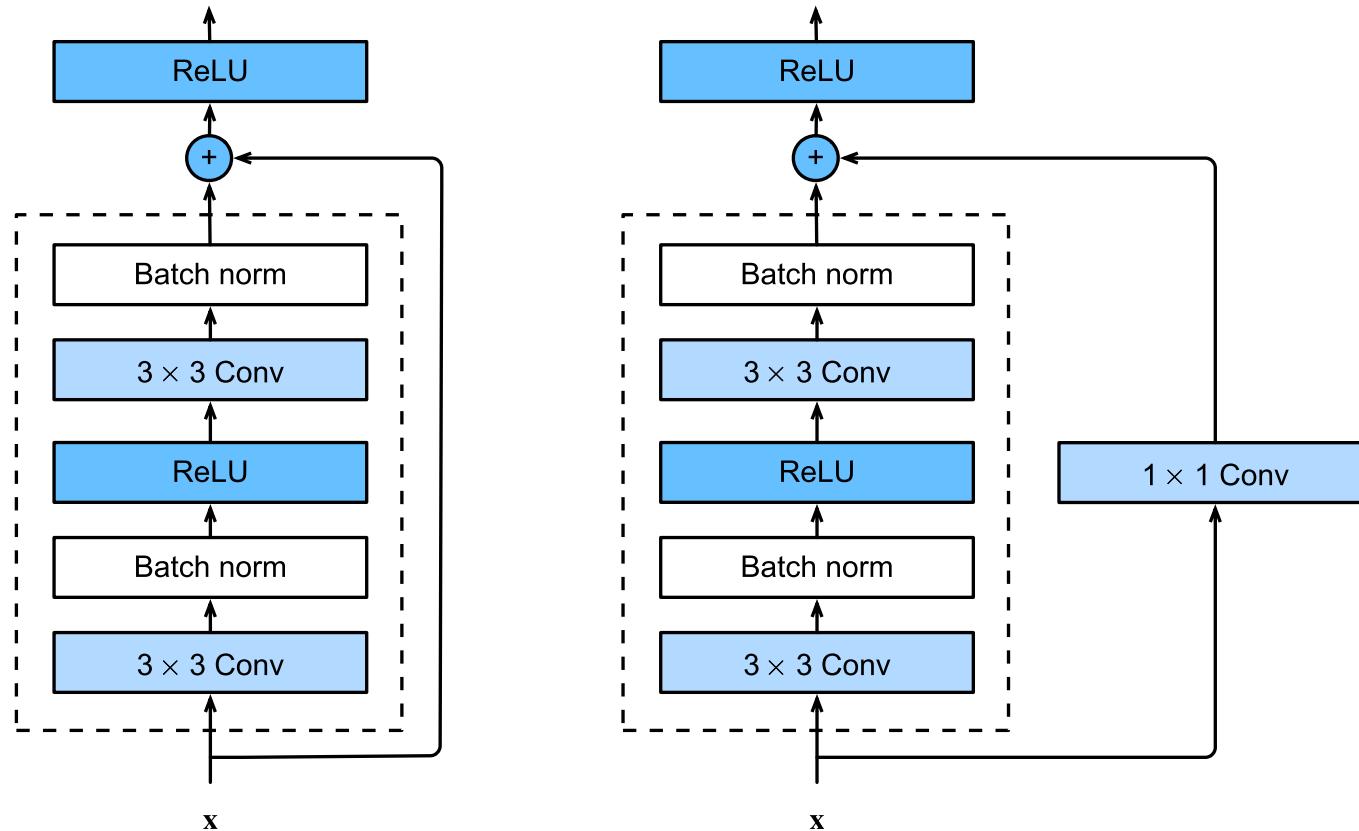


# ResNet: Introduction of residual blocks



Source: [https://d2l.ai/chapter\\_convolutional-modern/resnet.html](https://d2l.ai/chapter_convolutional-modern/resnet.html)

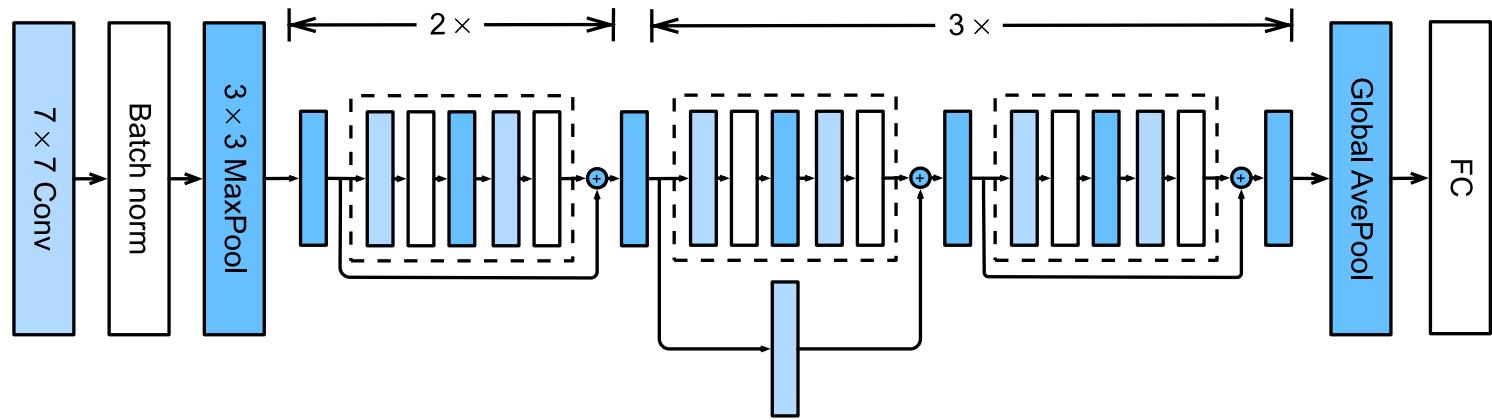
# ResNet...



ResNet block with and without convolution, which transforms the input into the desired shape for the addition operation

Source:  
[https://d2l.ai/chapter\\_convolutional-modern/resnet.html](https://d2l.ai/chapter_convolutional-modern/resnet.html)

# ResNet...



ResNet-18 Architecture

Implementation: [https://d2l.ai/chapter\\_convolutional-modern/resnet.html](https://d2l.ai/chapter_convolutional-modern/resnet.html)

# A Minimal ResNet Block Implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    expansion = 1 # used in deeper ResNets

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                            stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                            stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Downsample is used when dimensions change (e.g., stride > 1 or channels differ)
        self.downsample = downsample

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

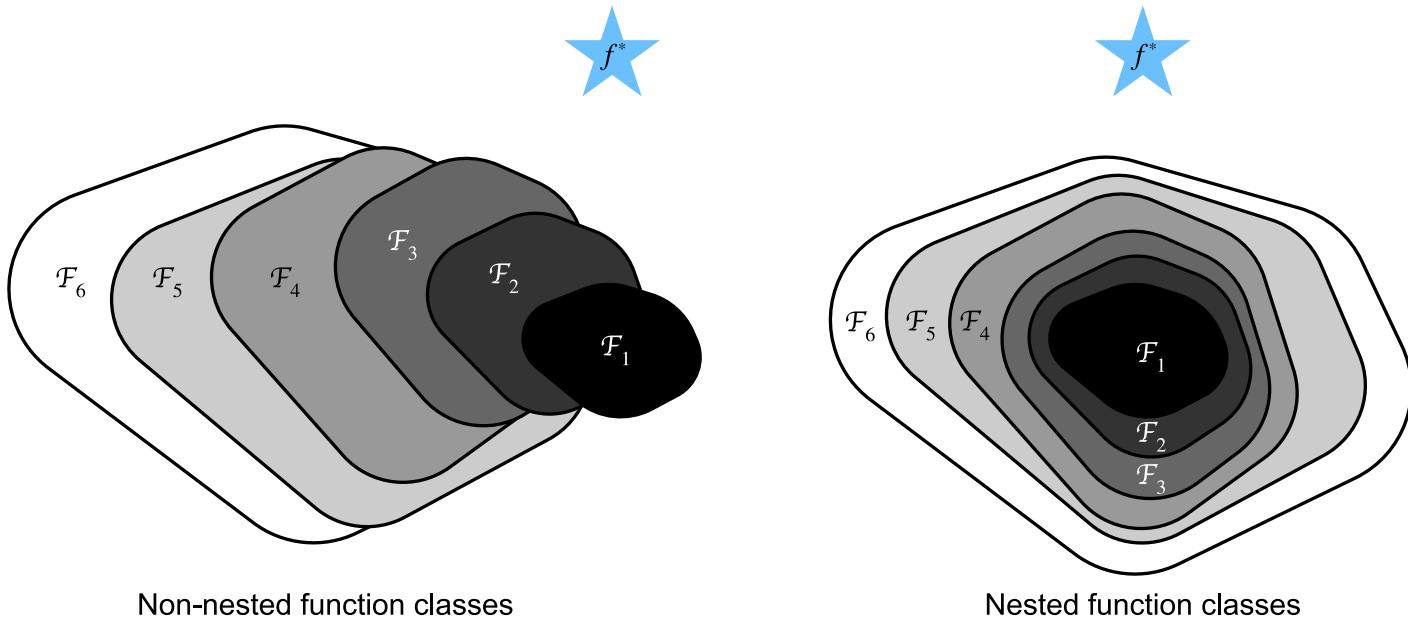
        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity # residual connection
        out = self.relu(out)

    return out
```

 Copy code

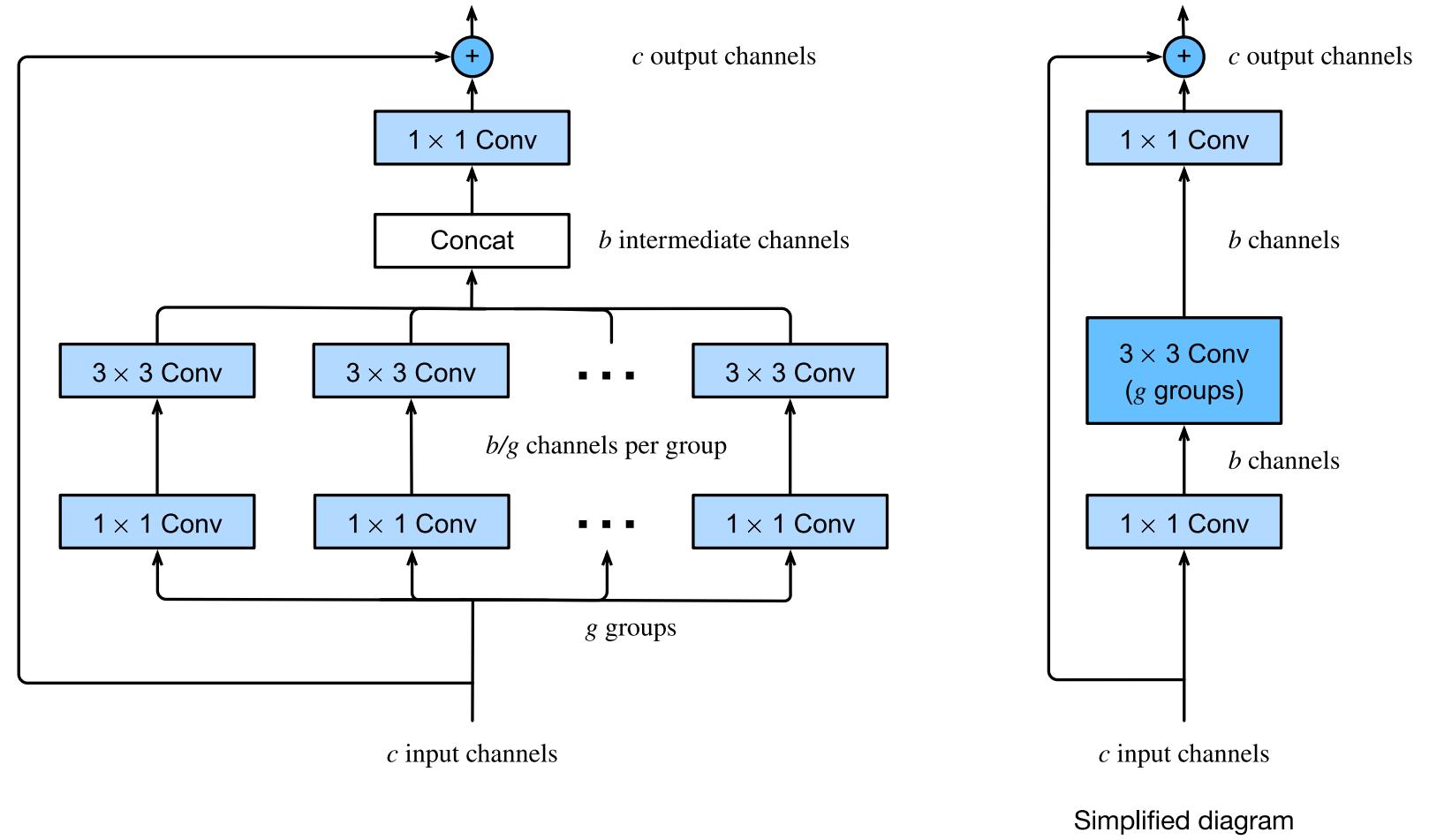
# Why does ResNet work well?



An explanation: [https://d2l.ai/chapter\\_convolutional-modern/resnet.html](https://d2l.ai/chapter_convolutional-modern/resnet.html)

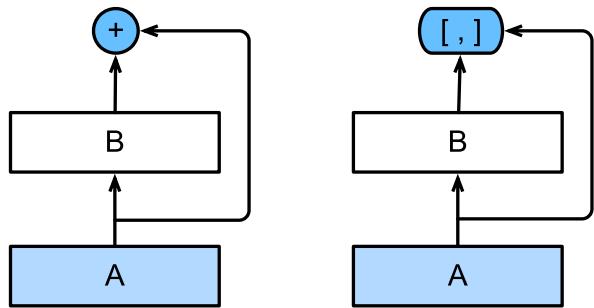
# 2016: ResNeXt

Trade-off between nonlinearity and dimensionality within a residual block

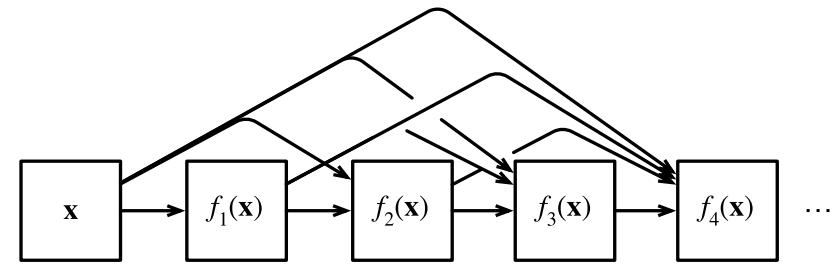


Implementation: [https://d2l.ai/chapter\\_convolutional-modern/resnet.html](https://d2l.ai/chapter_convolutional-modern/resnet.html)

# 2017: DenseNet - dense connections



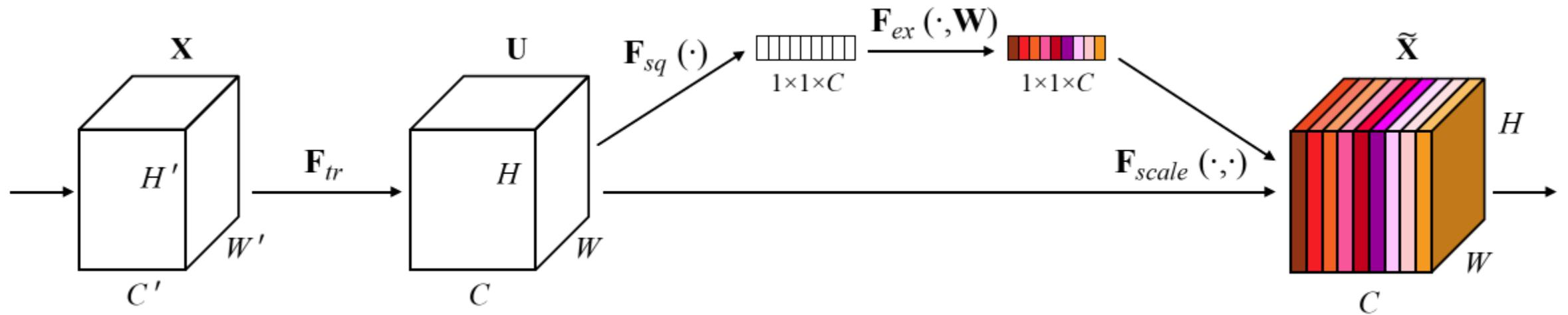
From residual to dense connections



dense connections = concatenations from preceding nodes

Implementation: [https://d2l.ai/chapter\\_convolutional-modern/densenet.html](https://d2l.ai/chapter_convolutional-modern/densenet.html)

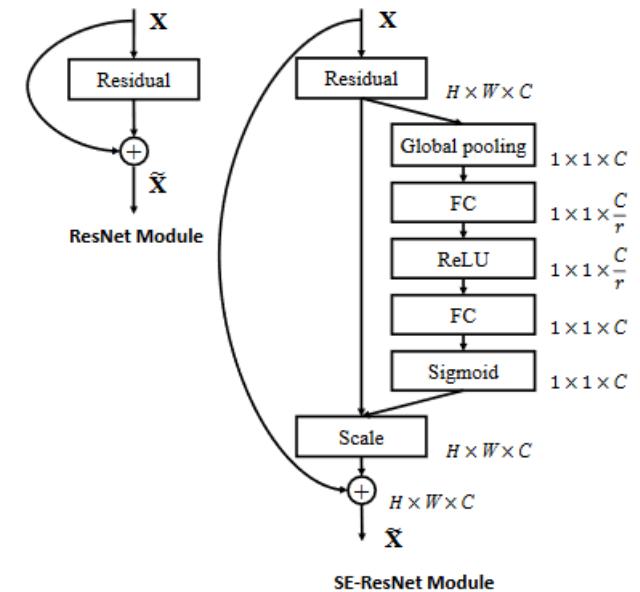
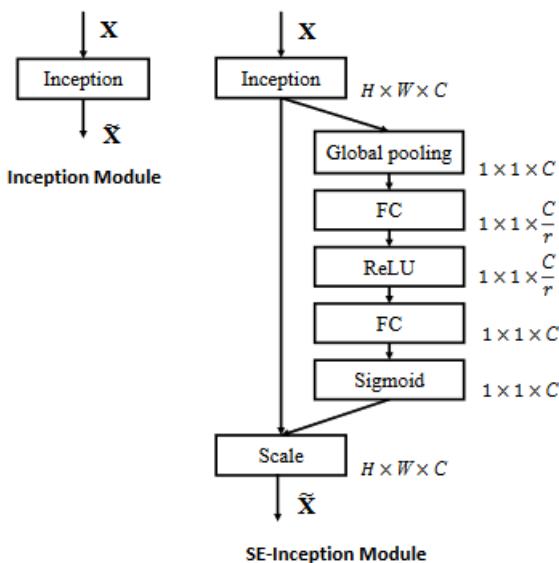
# 2017: SENet



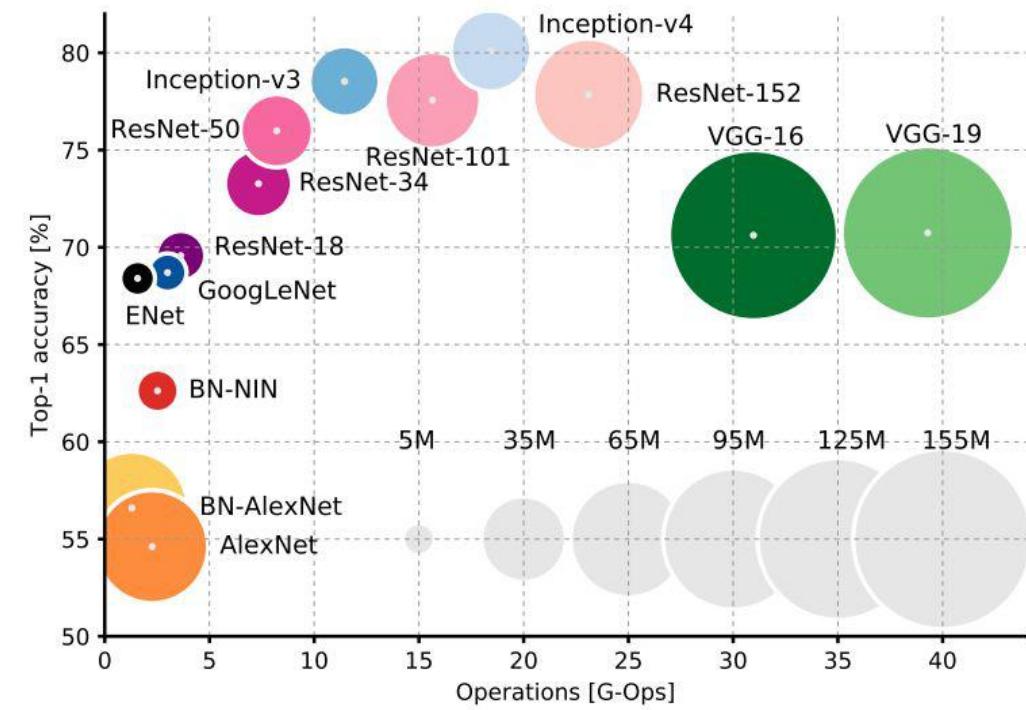
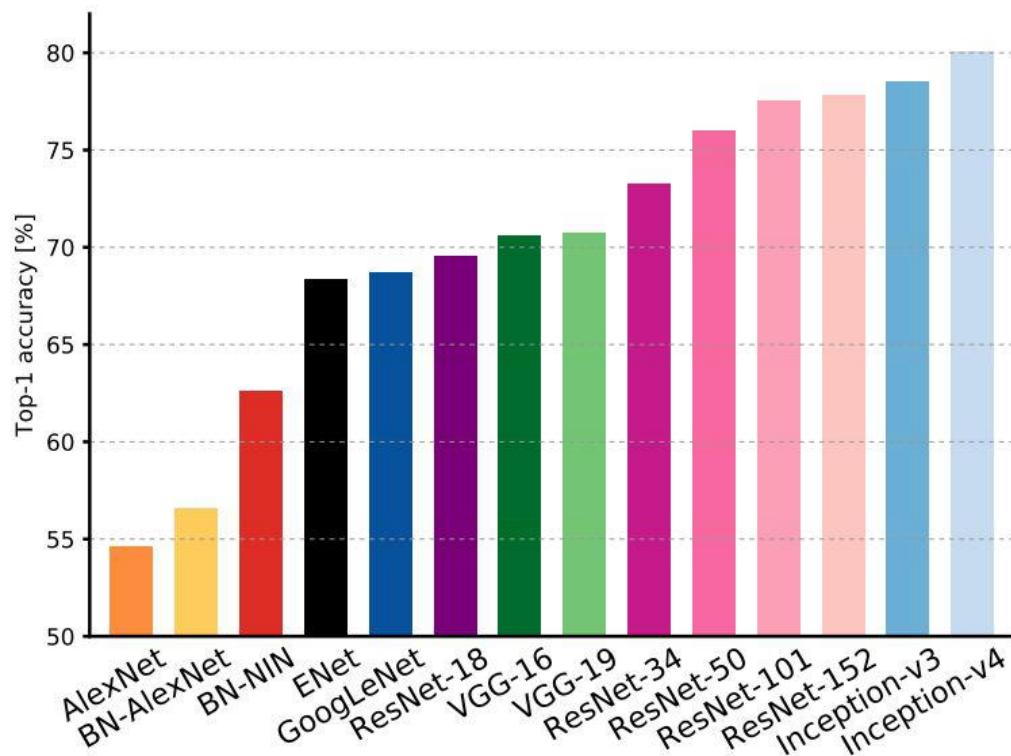
A Squeeze-and-Excitation block: <https://arxiv.org/pdf/1709.01507.pdf>

# SENet...

- Feature recalibration module to learn to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC'17 classification winner (using ResNeXt-152 as a base architecture)

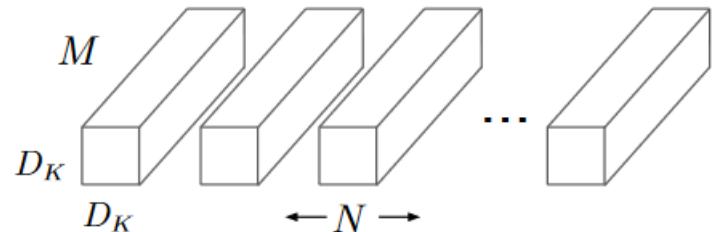


# Network performances

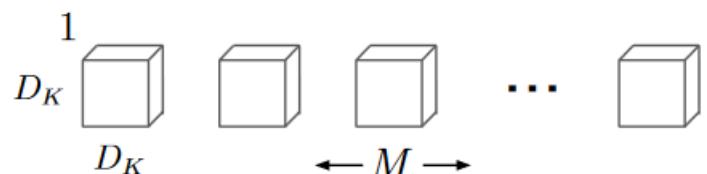


Source: <https://arxiv.org/pdf/1605.07678.pdf>

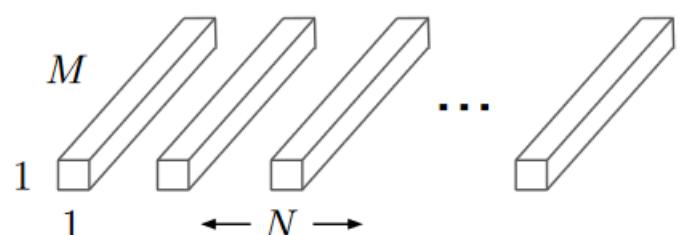
# 2017: MobileNet



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c)  $1 \times 1$  Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

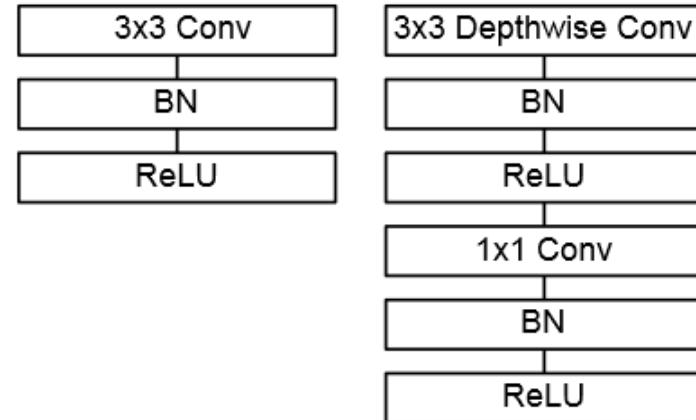


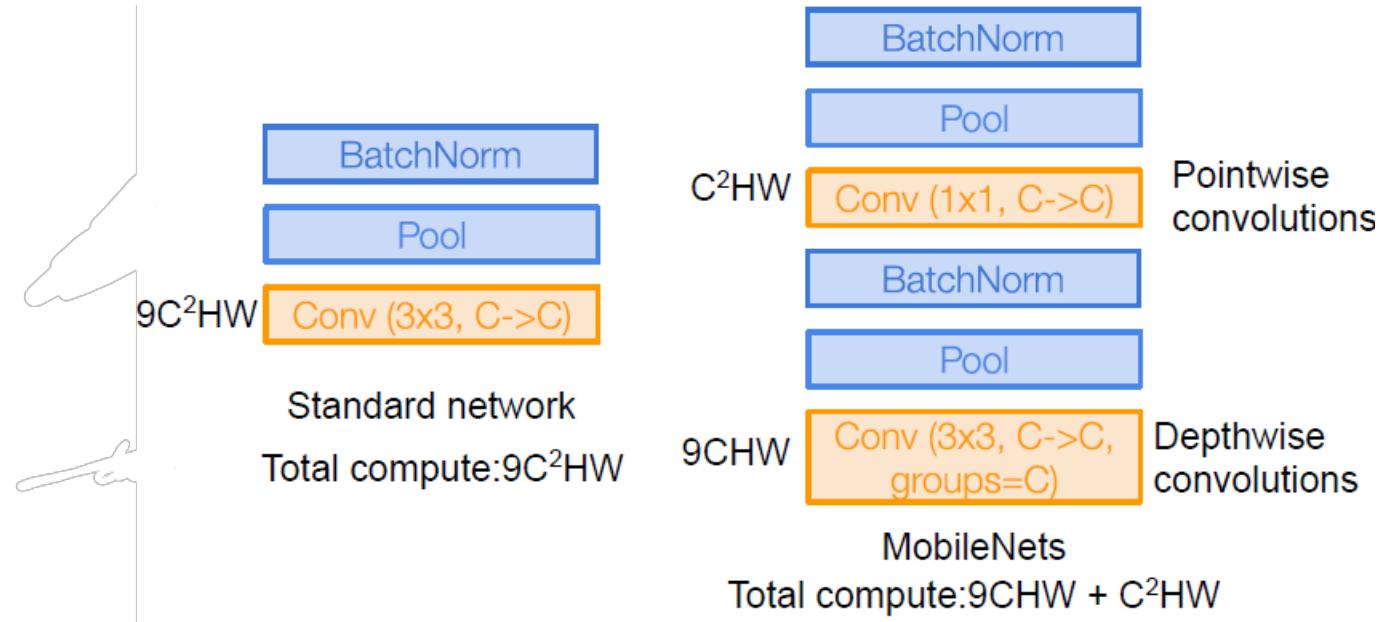
Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

Replace standard convolutions with depthwise convolutions and pointwise (aka  $1 \times 1$ ) convolutions

Why does it save FLOPs?

<https://arxiv.org/pdf/1704.04861.pdf>

# MobileNet: Saving FLOPs



Picture source: <http://cs231n.stanford.edu/>

Also look at:  
<https://arxiv.org/pdf/1704.04861.pdf>

# Neural architecture search (NAS)

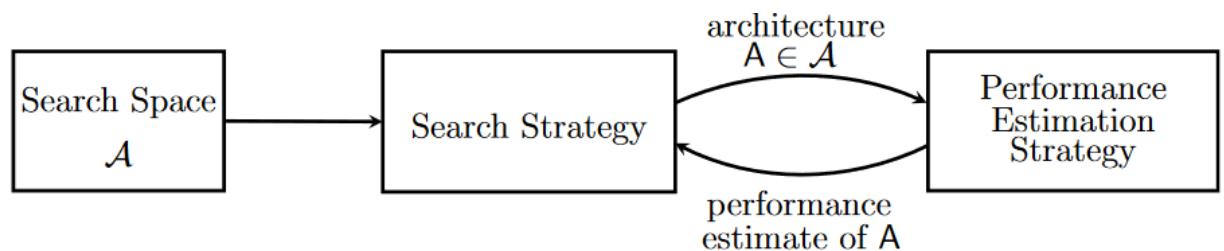


Figure 1: Abstract illustration of Neural Architecture Search methods. A search strategy selects an architecture  $A$  from a predefined search space  $\mathcal{A}$ . The architecture is passed to a performance estimation strategy, which returns the estimated performance of  $A$  to the search strategy.

<https://jmlr.org/papers/v20/18-598.html>

An comprehensive blog on NAS:

<https://lilianweng.github.io/posts/2020-08-06-nas/>

**Search space:** The NAS search space defines a set of operations (e.g. convolution, fully-connected, pooling) and how operations can be connected to form valid network architectures. The design of search space usually involves human expertise, as well as unavoidably human biases.

**Search algorithm:** A NAS search algorithm samples a population of network architecture candidates. It receives the child model performance metrics as rewards (e.g. high accuracy, low latency) and optimizes to generate high-performance architecture candidates.

**Evaluation strategy:** We need to measure, estimate, or predict the performance of a large number of proposed child models in order to obtain feedback for the search algorithm to learn. The process of candidate evaluation could be very expensive and many new methods have been proposed to save time or computation resources

# NAS...

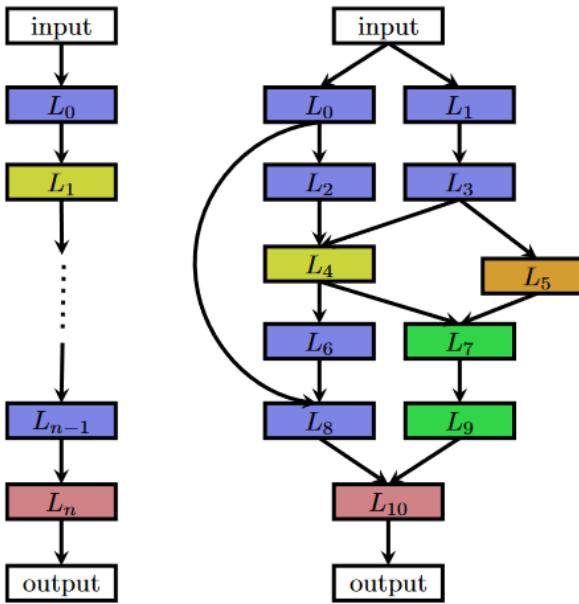


Figure 2: An illustration of different architecture spaces. Each node in the graphs corresponds to a layer in a neural network, e.g., a convolutional or pooling layer. Different layer types are visualized by different colors. An edge from layer  $L_i$  to layer  $L_j$  denotes that  $L_j$  receives the output of  $L_i$  as input. Left: an element of a chain-structured space. Right: an element of a more complex search space with additional layer types and multiple branches and skip connections.

# 2020: AnyNet – designing network design spaces

Instead of focusing on designing such individual instances, an alternative approach is to *design network design spaces* that characterize populations of networks

Implementation: [https://d2l.ai/chapter\\_convolutional-modern/cnn-design.html](https://d2l.ai/chapter_convolutional-modern/cnn-design.html)

[https://openaccess.thecvf.com/content\\_CVPR\\_2020/papers/Radosavovic\\_Designing\\_Network\\_Design\\_Spaces\\_CVPR\\_2020\\_paper.pdf](https://openaccess.thecvf.com/content_CVPR_2020/papers/Radosavovic_Designing_Network_Design_Spaces_CVPR_2020_paper.pdf)

# CNN Architectures: Summary

- Depth in a network helps
- Residual connections are significant
- ReLU is very important
- Batch norm layers are helpful
- FLOPs can be reduced by using depth-wise and  $1 \times 1$  convolutions

# Tricks and techniques for training CNNs

- Activation functions
- Initialization of weights
- Data augmentation (will be in assignment 3)
- Optimizations
- Batch normalization (seen earlier)
- Transfer learning (also in assignment 2)

# Different activation functions

[https://en.wikipedia.org/wiki/Activation\\_function](https://en.wikipedia.org/wiki/Activation_function)

# Weight initialization

- Vanishing and exploding gradient problem in a deep network
  - Small initial weights lead to vanishing gradients
  - Large initial weights lead to exploding gradients
- Just right type of initial weights should keep mean of activations close to 0, while keeping variance same across layers
- Xavier initialization

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}})$$
$$b^{[l]} = 0$$

<https://www.deeplearning.ai/ai-notes/initialization/index.html>

# Data augmentation

- Inflate the size of training data by performing various random transforms to the images:
  - Geometric transforms, such as rotation, scaling, reflection
  - Photometric transforms, such as slight color jitter
- Why does data augmentation help?
  - It is a form of regularizer
- Pytorch has several transforms available that can work with data loading module: <https://pytorch.org/vision/main/transforms.html>

# Optimizations

- The basic version is SGD (aka mini batch gradient descent)
- There are several variations  
([https://d2l.ai/chapter\\_optimization/index.html](https://d2l.ai/chapter_optimization/index.html))
- My go to choice is ADAM

# Revisiting dropout

Let's take a look at the tutorial:

[https://xuwd11.github.io/Dropout\\_Tutorial\\_in\\_PyTorch/](https://xuwd11.github.io/Dropout_Tutorial_in_PyTorch/)

<https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>

# Transfer learning

- How to adapt a large model to your own (typically) small dataset?
- PyTorch transfer learning tutorial:  
[https://pytorch.org/tutorials/beginner/transfer\\_learning\\_tutorial.html](https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html)