



Lecture 02: Convolutional Neural Networks and Variants

Course Information

- Course website: <https://www.saiqianzhang.com/COURSE/>
- I use Brightspace to post announcements and grades
- I provide an [online zoom meeting](#) option for people interested in auditing the class. However, enrolled students are required to attend in person unless special condition.
- Discussion groups has been created in the Brightspace
- Course email: efficientaiaccelerator@gmail.com

Course Information

- The course will involve 13 lectures, 3 coding assignments, 1 final project, 1 midterm exam and in-class quiz.
 - In-class quiz (15%)
 - In-course presentation (5%)
 - Assignments (30%): total three of them, each counts 10%
 - Midterm (25%)
 - Final project (25%)
 - Project Proposal (5%) (1 page)
 - Final Presentation (10%)
 - Final Report (10%)
- Readings:
 - Course notes and papers (optional)
 - (reference) Goodfellow, Ian. "Deep learning." (2016). <https://www.deeplearningbook.org/>
- Lecture time:
 - Friday: 5:00pm-7:30pm
- Office hour:
 - Friday: 1:30pm-2:30pm, or by appointment ([Zoom](#))

In-Course Presentation General Policy

- Please use Google Slides to create your presentation slides.
- Sign up here:
<https://docs.google.com/spreadsheets/d/1QL7qBQnMluk-uTniPeH0H6i4ij40hVbuOo0ACXBCMV4/edit?usp=sharing>
- There is limit on the number of slides (10 pages for algorithmic paper, 14 pages for architectural paper), make sure to stay within the presentation time limit (15 mins for algorithm paper, 25 mins for arch paper).
- Submission Deadline: Please send the link to your Google Slides presentation by Friday before 2:00 PM each week to Shawn Yin (xy2053@nyu.edu)
- Please ensure that the Google Slides link is set to 'Anyone with the link can view' so that it is accessible to us.

Recap

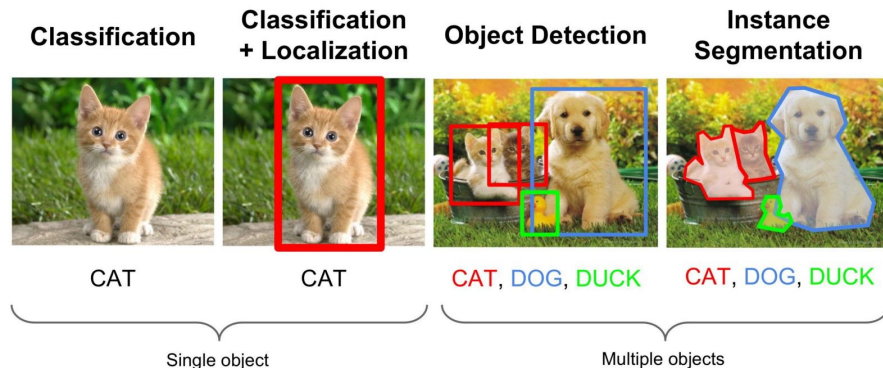
- DNN basics
 - Multilayer perceptron
 - Linear layer, activation function, softmax layer
 - Loss functions
 - Weights decay
 - Dropout
 - Optimizer
 - Learning rate scheduler
 - Weight Initialization

Topics

- Convolutional Neural Network
 - Basic building blocks
 - Popular CNN architectures
 - VGG
 - ResNet
 - MobileNet
 - ShuffleNet
 - SqueezeNet
 - DenseNet
 - EfficientNet
 - ConvNext
 - CNN architectures for other vision tasks
 - Image Segmentation, Object Detection

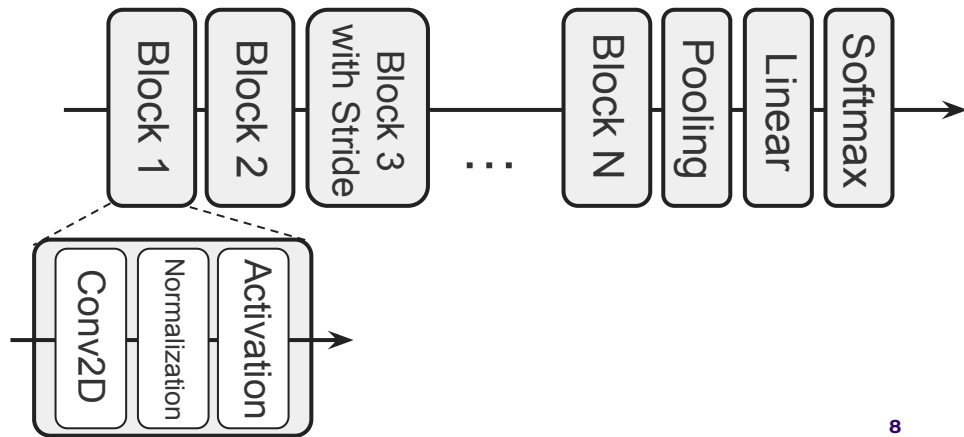
Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) are a type of artificial neural network designed for processing structured grid data, such as images. They're particularly effective in tasks like image recognition, object detection and segmentation.
- The building blocks of a CNN includes:
 - Convolutional layer
 - Activation layer
 - Normalization layer
 - Pooling layer
 - Softmax layer

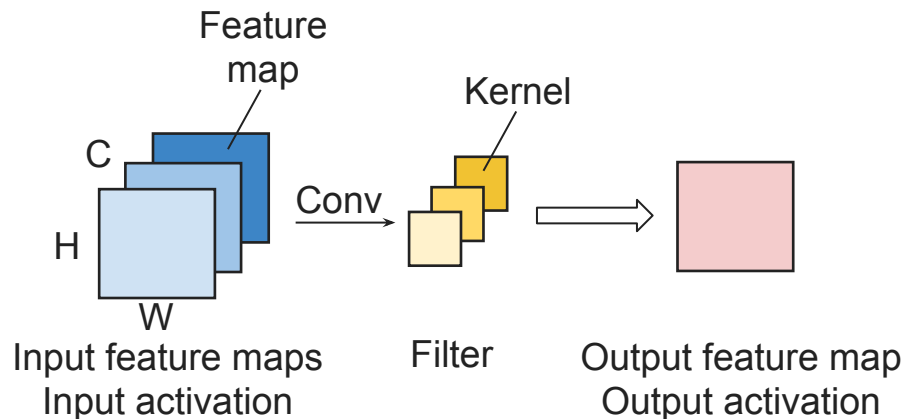


Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) are a type of artificial neural network designed for processing structured grid data, such as images. They're particularly effective in tasks like image recognition, object detection and segmentation.
- The building blocks of a CNN includes:
 - Convolutional layer
 - Activation layer
 - Normalization layer
 - Pooling layer
 - Softmax layer

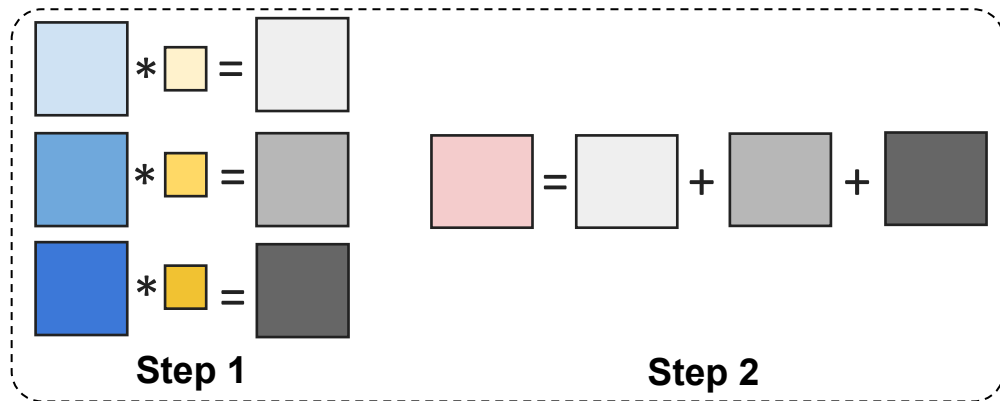
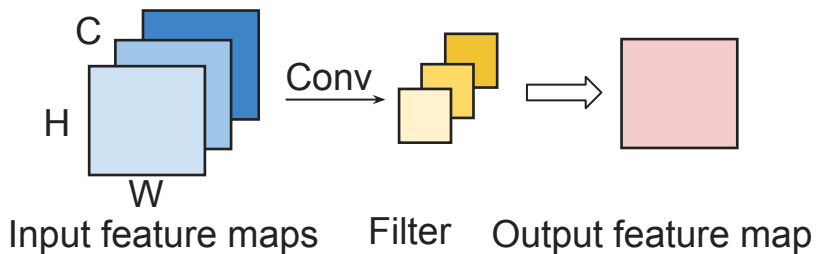


Convolutional Layers: Terminology



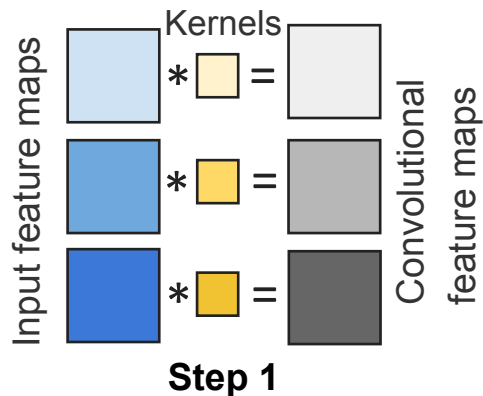
- Core building block of a CNN, it is also the most computational intensive layer.

Convolutional Layers

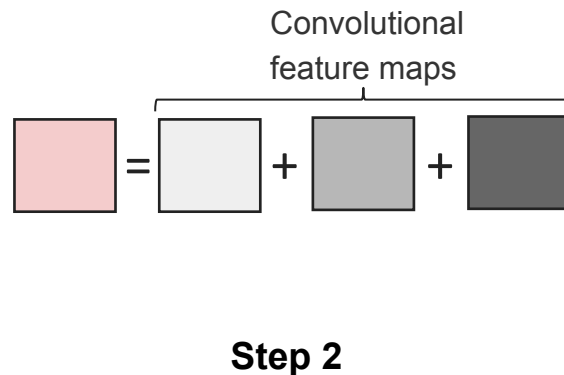


- Core building block of a CNN, it is also the most computational intensive layer.

Convolutional Layers

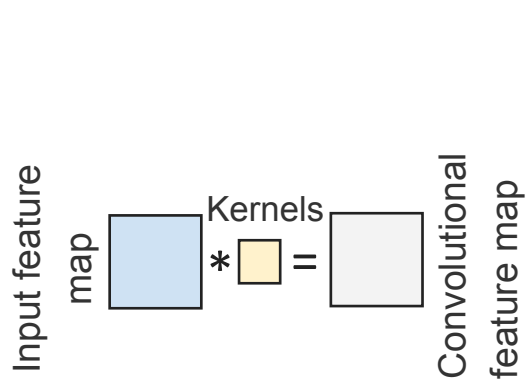


- Each kernel moves across the spatial dimensions of feature maps in the input activations, analyzing the information within those spatial dimensions.



- The information from each feature maps are then aggregated by summing the Convolutional feature maps together.
- A bias may be introduced.

2D Convolution: An Example



Input feature map

1	0	5
3	-1	-2
4	0	-3

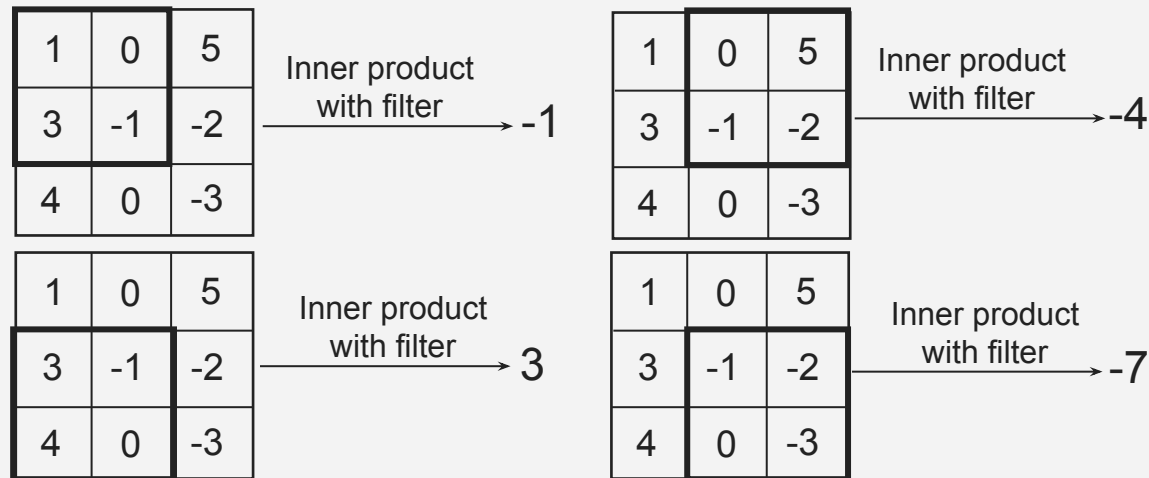
Kernels

1	0
0	2

=

-1	-4
3	-7

Convolutional feature map



Padding

Input feature map

1	0	5
3	-1	-2
4	0	-3

*

Kernel

1	0	3
0	2	-2
1	3	-1

= 25

- Padding is used to preserve the spatial size of the output features.

Padding of 1

0	0	0	0	0
0	1	0	→	0
0	3	-1	-2	0
0	↓	0	-3	0
0	0	0	0	0

*

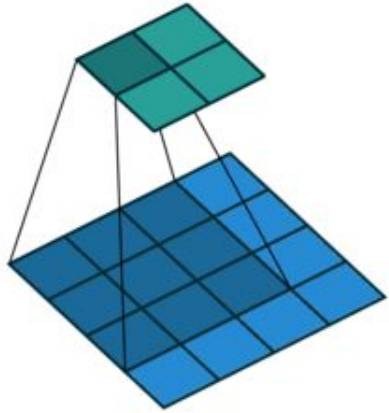
Filter

1	0	3
0	2	-2
1	3	-1

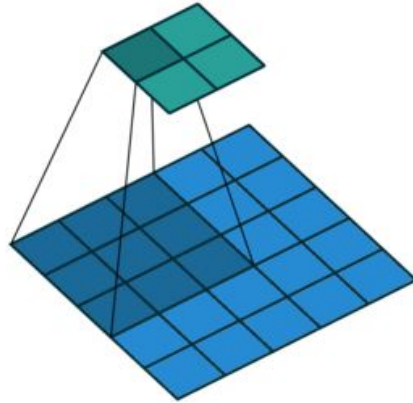
=

12	20	3
20	25	-10
5	3	-7

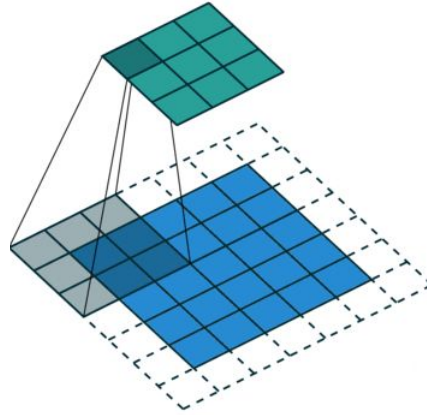
Stride



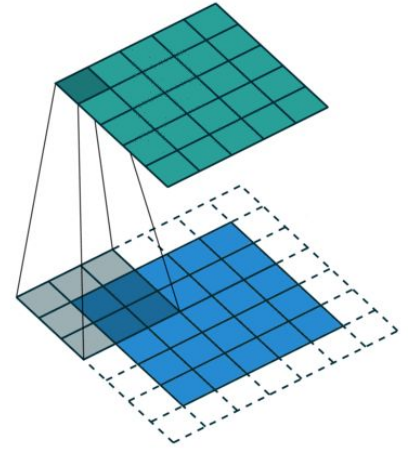
padding = 0, stride = 1



padding = 0, stride = 2



padding = 1, stride = 2



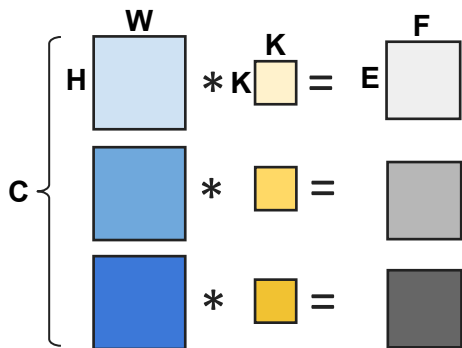
padding = 1, stride = 1

Summary

- $H_{out} = (H_{in} - K + 2P) / S + 1$
- H_{in} and H_{out} are the spatial sizes of the input and convolutional feature maps.
- K is the weight kernel size
- P is the padding size
- S is the stride
- For example:
 - For input size of $224 \times 224 \times 3$, weight kernel size is 3×3 , padding size is 1 and stride size is 1, then the output size is $(224 - 3 + 2) / 1 + 1 = 224$.

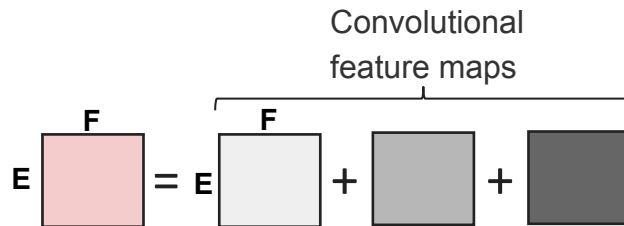
The diagram shows the convolution operation as a visual equation: $H_{in} \times K = H_{out}$. On the left, a blue square represents the input feature map, with H_{in} written above it and H_{in} to its left. This is followed by a multiplication symbol $*$ and a yellow square representing the kernel, with K written above it and K to its left. An equals sign $=$ follows, then a gray square representing the output feature map, with H_{out} written above it and H_{out} to its left.

Computational Cost



Step 1

- Each kernel moves across the spatial dimensions of feature maps in the input activations, analyzing the information within those spatial dimensions.

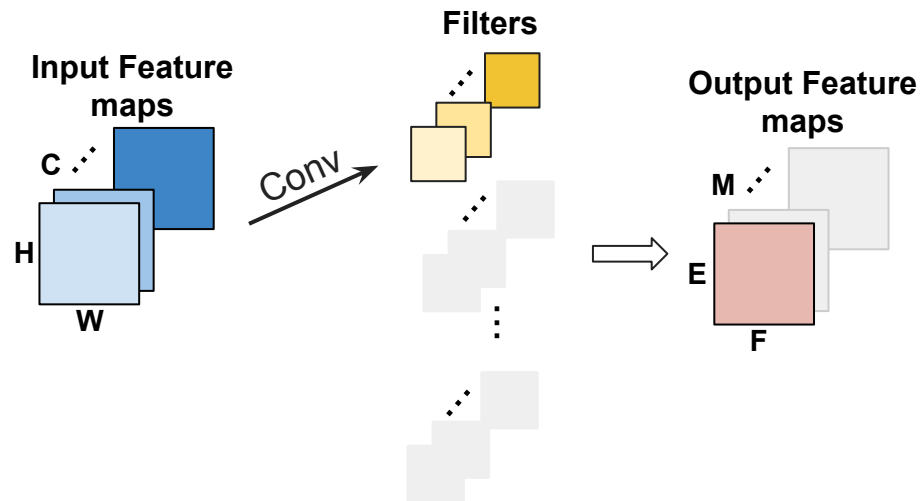


Step 2

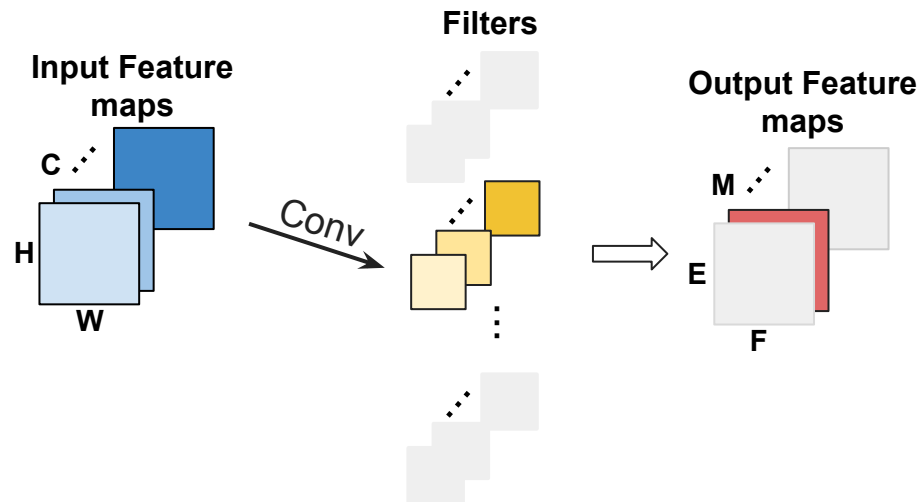
- The information from each feature maps are then aggregated by summing the Convolutional feature maps together.
- A bias may be introduced.

Computational cost in Multiply–accumulate operations (MAC): $E \times F \times K \times K \times C$

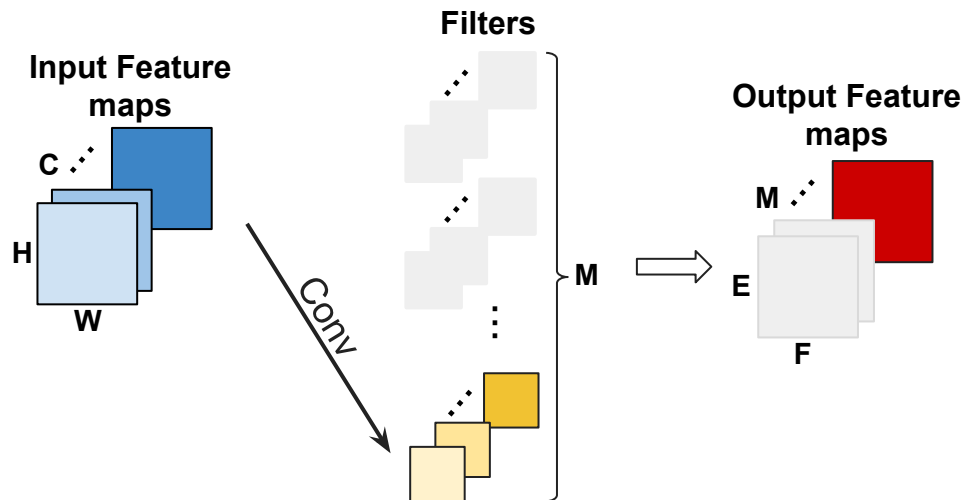
Convolution



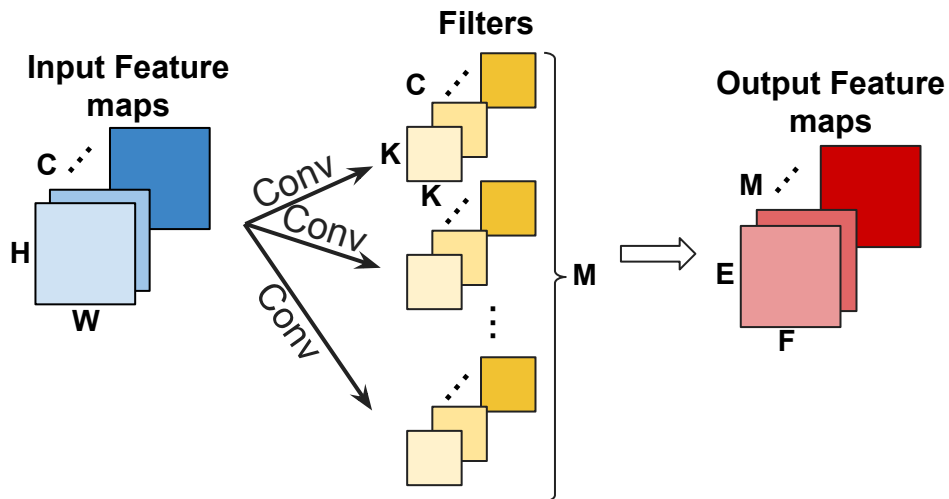
Convolution



Convolution



Convolution



- Number of MACs: $M \times K \times K \times C \times E \times F$
- Storage cost:
 $32 \times (M \times C \times K \times K + C \times H \times W + M \times E \times F)$

C: number of input channels

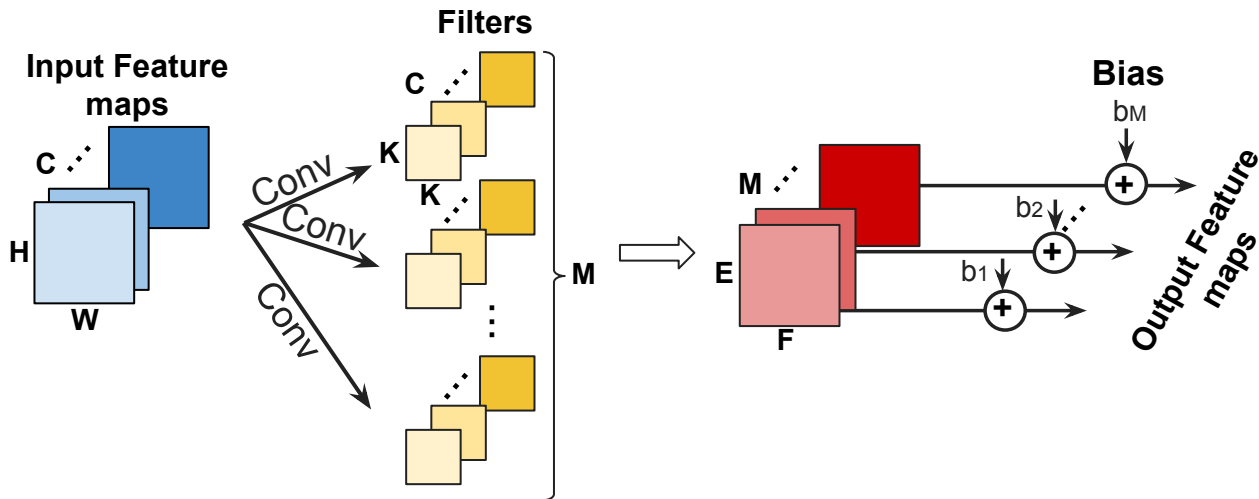
H,W: size of the input feature maps

M: number of weight filters

K: weight kernel size

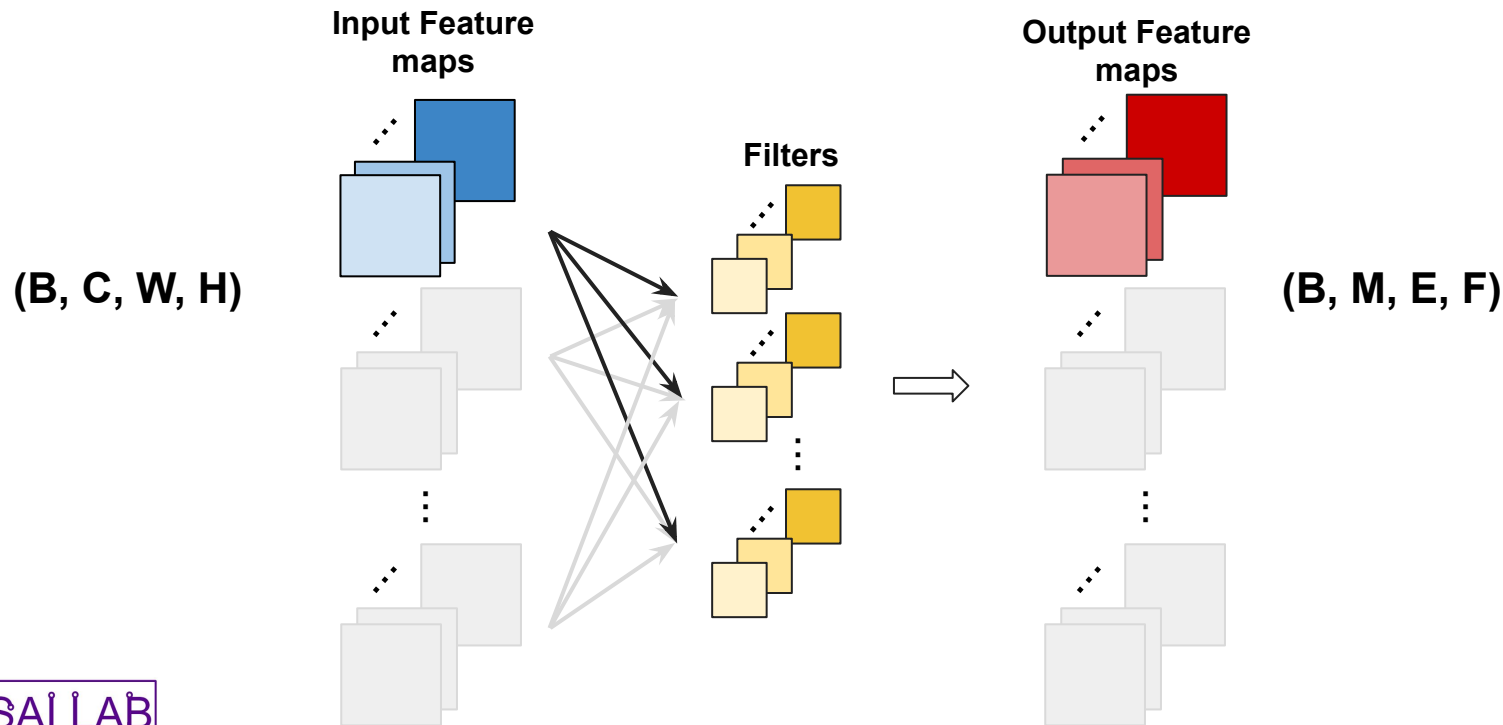
E,F: size of the output feature maps

Convolution

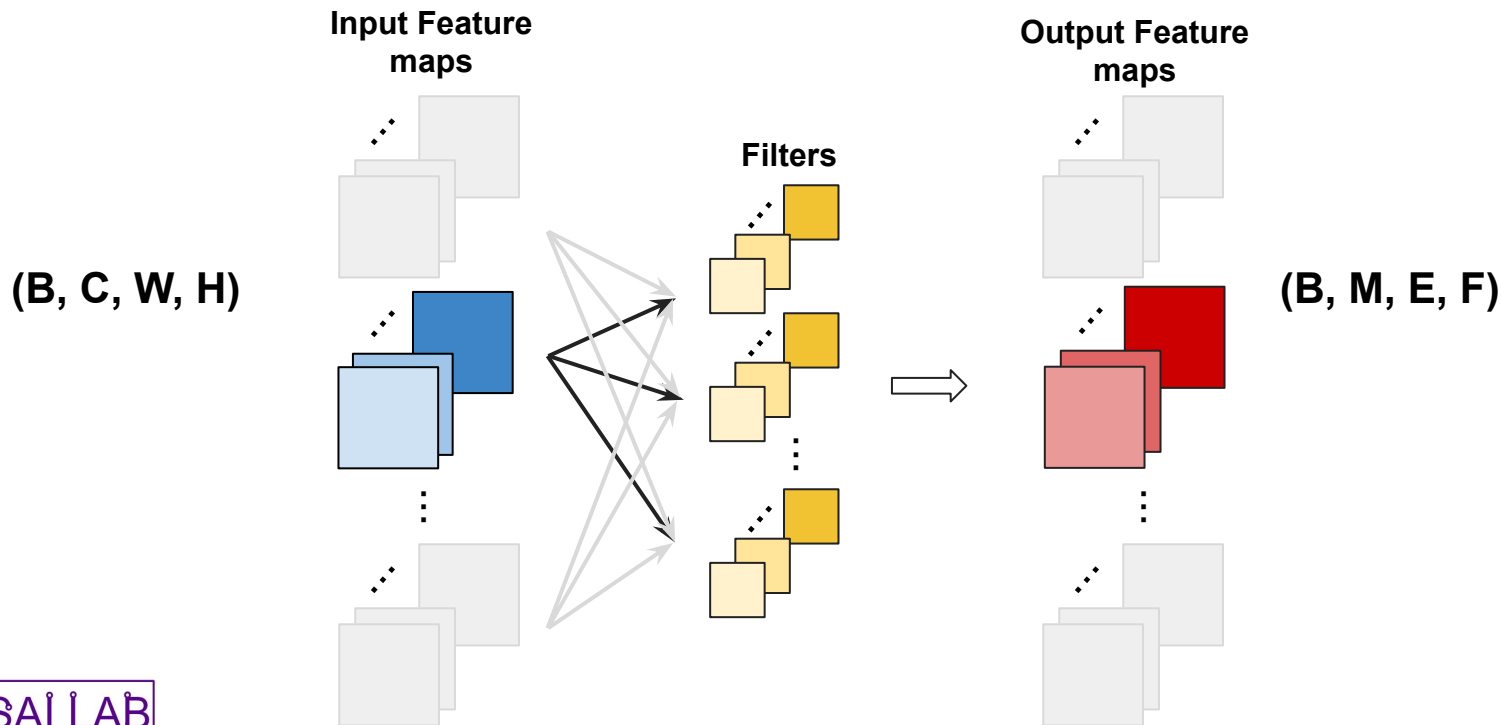


- A bias term may be added to the convolutional output, applied across each output channel.

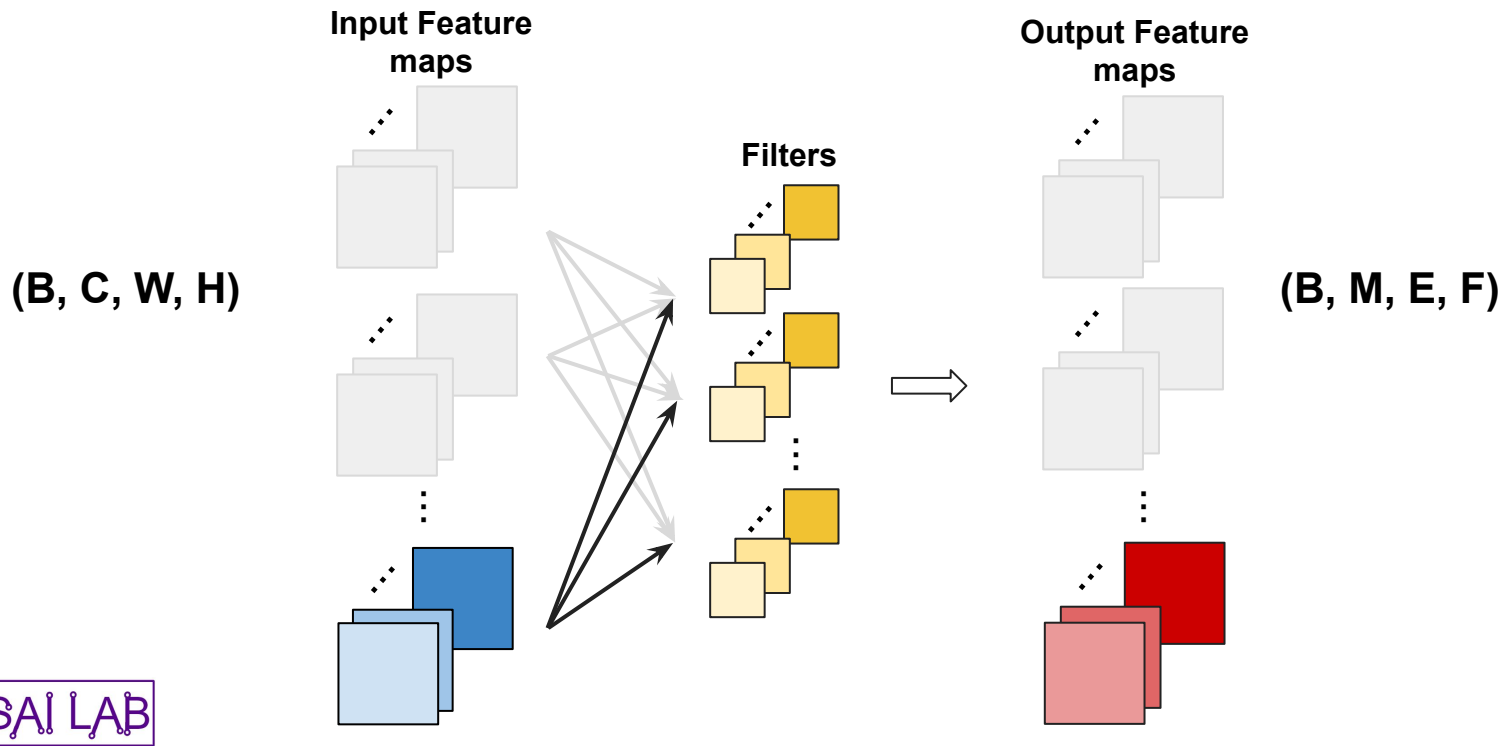
Convolution



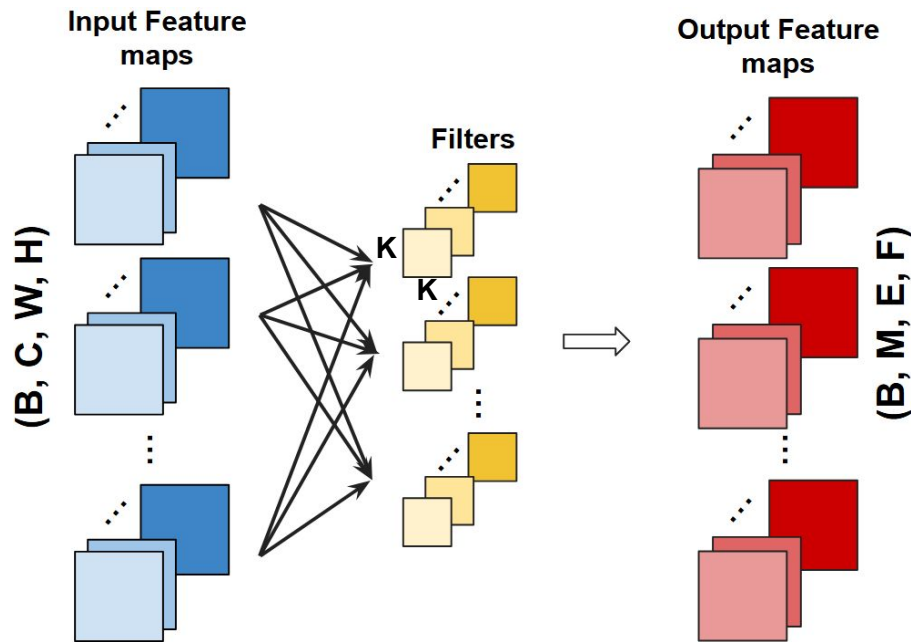
Convolution



Convolution



Computational Cost: Standard Convolution



- Number of MACs: $B \times M \times K \times K \times C \times E \times F$
- Storage cost:
 $32 \times (M \times C \times K \times K + B \times C \times H \times W + B \times M \times E \times F)$

B: batch size

C: number of input channels

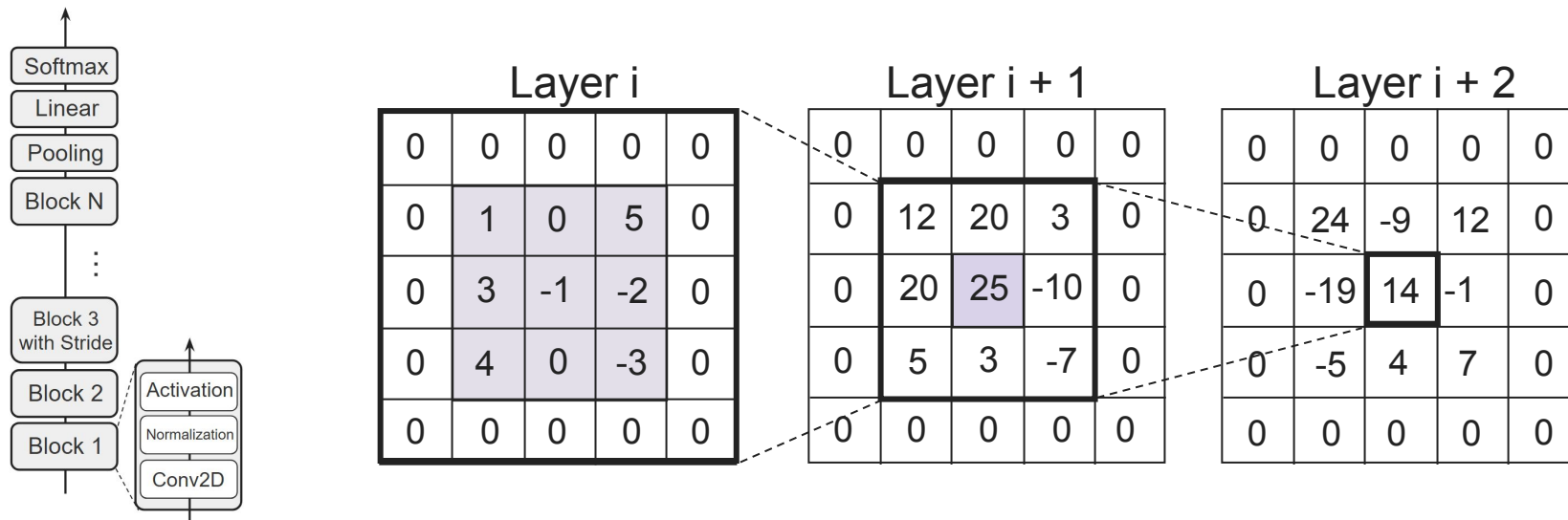
H, W: size of the input feature maps

M: number of weight filters

K: weight kernel size

E, F: size of the output feature maps

Receptive Field of CNN across Layers



- Assume a kernel size of 3 by 3.
- Every elements at layer i is a function of the entire receptive fields of the previous layers.

Activation Functions: ReLU

- Rectifier linear operation (ReLU) applies an elementwise activation function to the output feature maps.
- This leaves the size of the output feature maps unchanged.
- $f(x) = x$ if $x > 0$, $f(x) = 0$ otherwise.

1	0	5	ReLU	1	0	5
3	-1	-2		3	0	0
4	0	-3		4	0	0

Activation Functions: GeLU

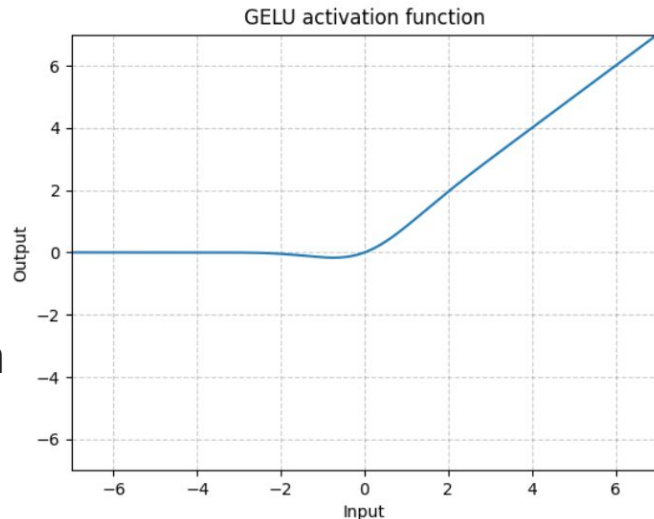
- Gaussian error linear unit (GeLU):

$$\text{GeLU}(x) = x\Phi(x)$$

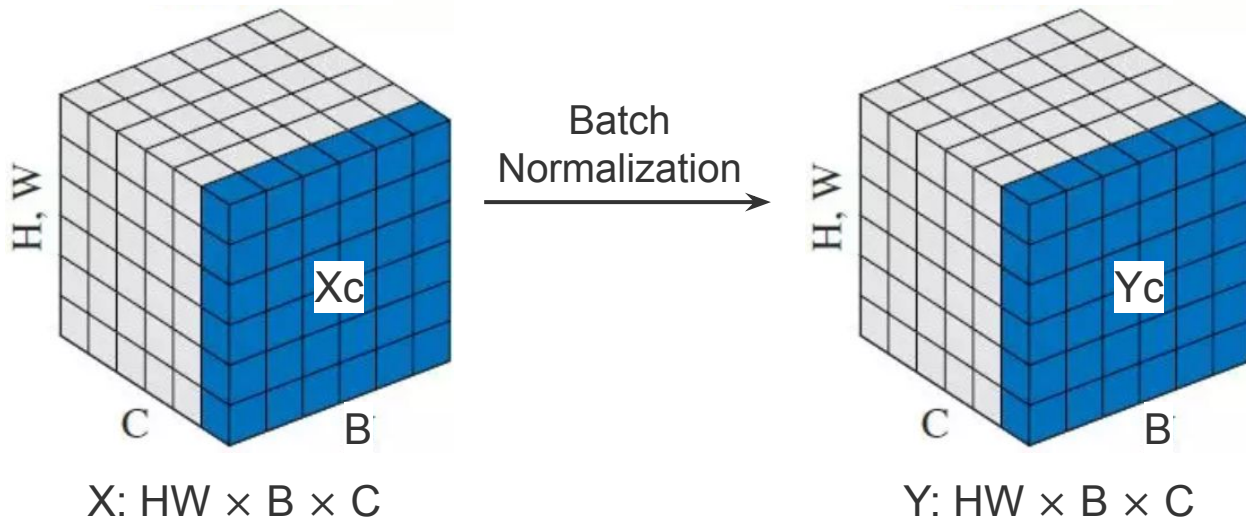
$$\Phi(x) = P(y \leq x), \text{ where } Y \sim N(0, 1)$$

$$0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

- GeLU is increasingly being adopted in transformers and CNNs today.

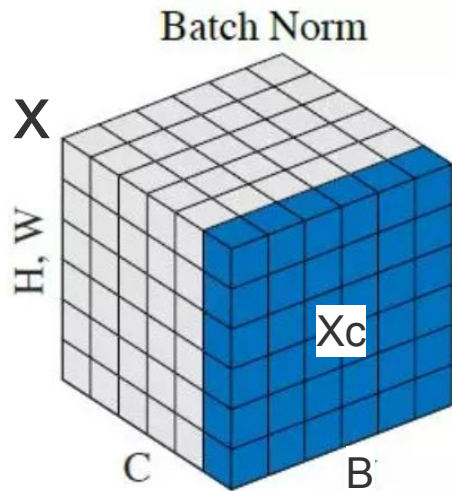


Batch Normalization



- **Batch Normalization (BatchNorm)** is a technique used in deep learning to improve the training stability and performance of neural networks.

Batch Normalization



$X: HW \times B \times C$

$$Y_c = \alpha_c \frac{X_c - \mu_c}{\sigma_c} + \beta_c \quad \text{For each } c \in C$$

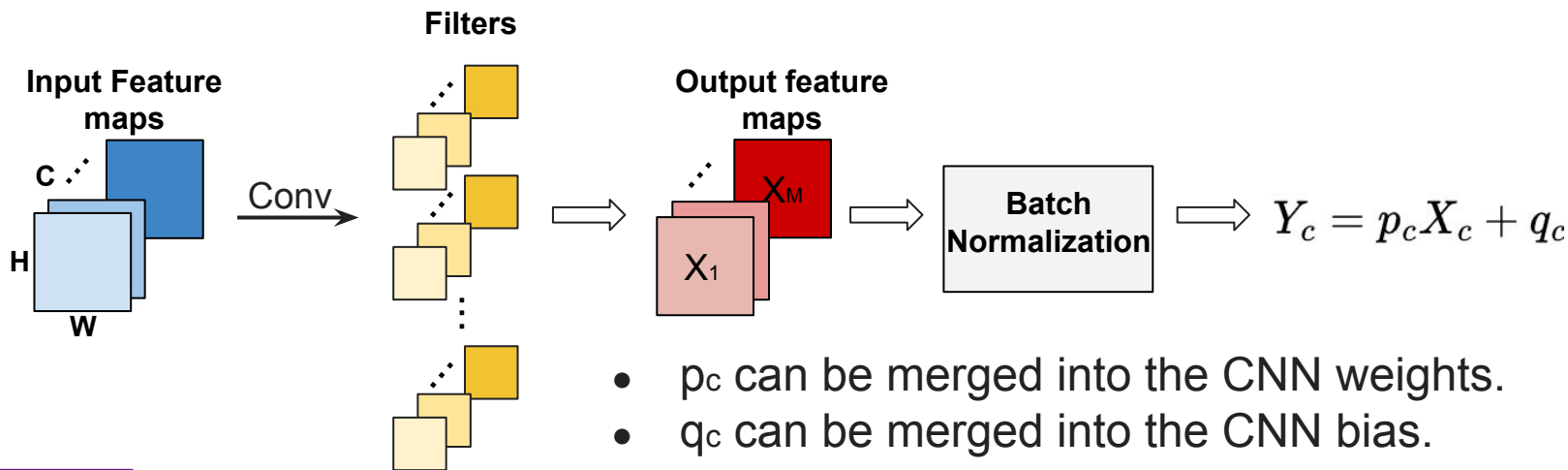
$$\alpha = \{\alpha_c\}, \beta = \{\beta_c\}, \mu = \{\mu_c\}, \sigma = \{\sigma_c\}$$

- For each channel c , we have:
 - X_c : ($HW \times B$)
 - μ_c and σ_c are the mean and standard deviation of X_c .
 - α_c and β_c are learnable parameters
 - $\alpha_c, \beta_c, \mu_c, \sigma_c$ are scalars
- Overall, we have:
 - μ, σ, α and β all have a length of C
 - μ, σ, α and β are all fixed during the inference
 - μ, σ are statistics based on the training dataset

Batch Normalization: During Inference

- Given all the parameters are fixed, for each channel c , we have:

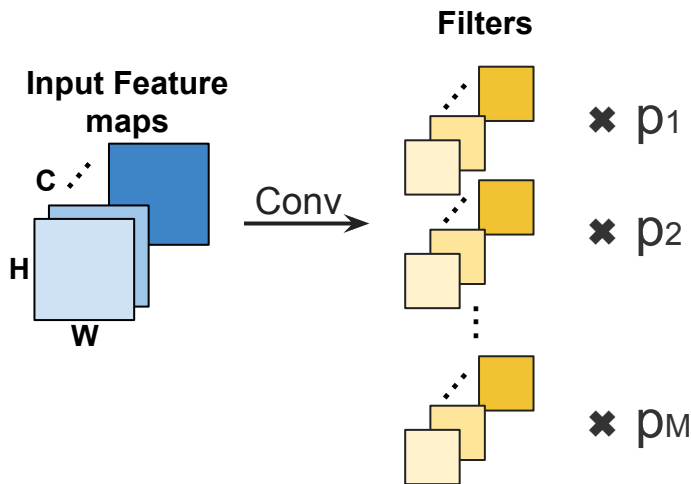
$$Y_c = \alpha_c \frac{X_c - \mu_c}{\sigma_c} + \beta_c = \frac{\alpha_c}{\sigma_c} X_c + \left(\beta_c - \frac{\alpha_c \mu_c}{\sigma_c} \right) \implies Y_c = p_c X_c + q_c$$



Batch Normalization

- For each channel c , we have:

$$Y_c = \alpha_c \frac{X_c - \mu_c}{\sigma_c} + \beta_c = \frac{\alpha_c}{\sigma_c} X_c + \left(\beta_c - \frac{\alpha_c \mu_c}{\sigma_c} \right) \Rightarrow Y_c = p_c X_c + q_c$$

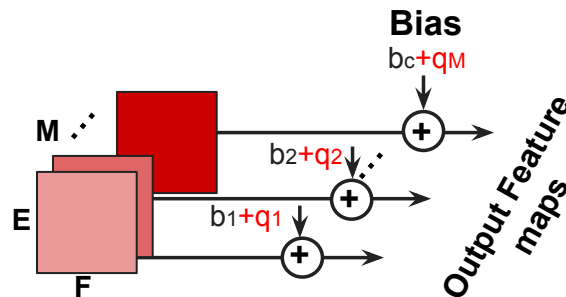
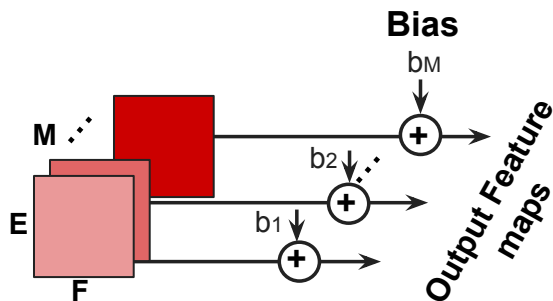


- We can fold in the p and q to the weights and bias of convolutional layer during inference and reduce the online computational cost.

Batch Normalization

- For each channel c , we have:

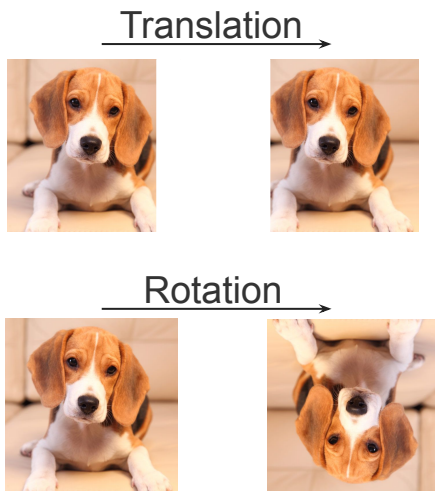
$$Y_c = \alpha_c \frac{X_c - \mu_c}{\sigma_c} + \beta_c = \frac{\alpha_c}{\sigma_c} X_c + \left(\beta_c - \frac{\alpha_c \mu_c}{\sigma_c} \right) \Rightarrow Y_c = p_c X_c + q_c$$



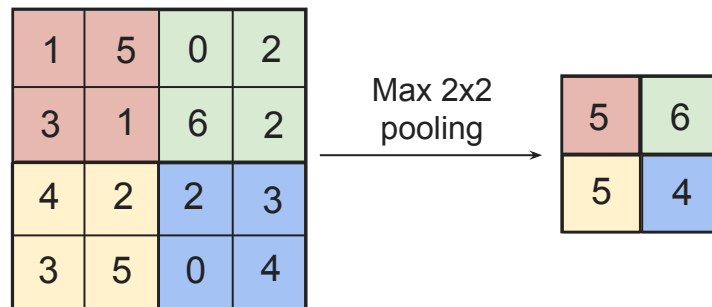
- We can fold in the p and q to the weights and bias of convolutional layer during inference and reduce the online computational cost.

Pooling

- Enhance the model invariance to spatial transformations such as translation and rotation, thereby reducing the risk of overfitting.
- Reduce the spatial size of the representation and reduce the amount of parameters and computation in the CNN.



Should produce the same prediction result



Strided Convolution As Pooling Layer

- Recent Neural Networks replace pooling layers with Strided Convolution

Diagram illustrating a 1D strided convolution operation with a kernel size of 1 and a stride of 2. The input is a 4x4 matrix, and the output is a 2x2 matrix.

1	5	0	2
3	1	6	2
4	2	2	3
3	5	0	4

Kernels

1

=

1	0
4	2

Stride = 2

Diagram illustrating a 1D strided convolution operation with a kernel size of 1 and a stride of 2. The input is a 4x4 matrix, and the output is a 2x2 matrix.

1	5	0	2
3	1	6	2
4	2	2	3
3	5	0	4

Kernels

w

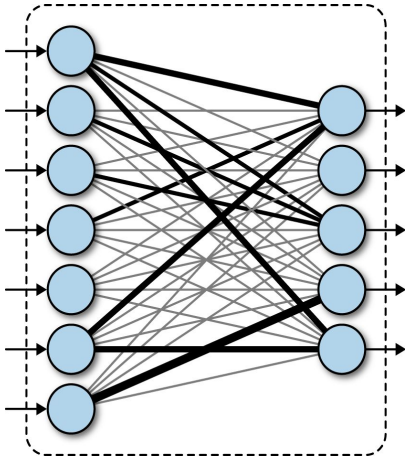
=

w	0
4w	2w

Stride = 2

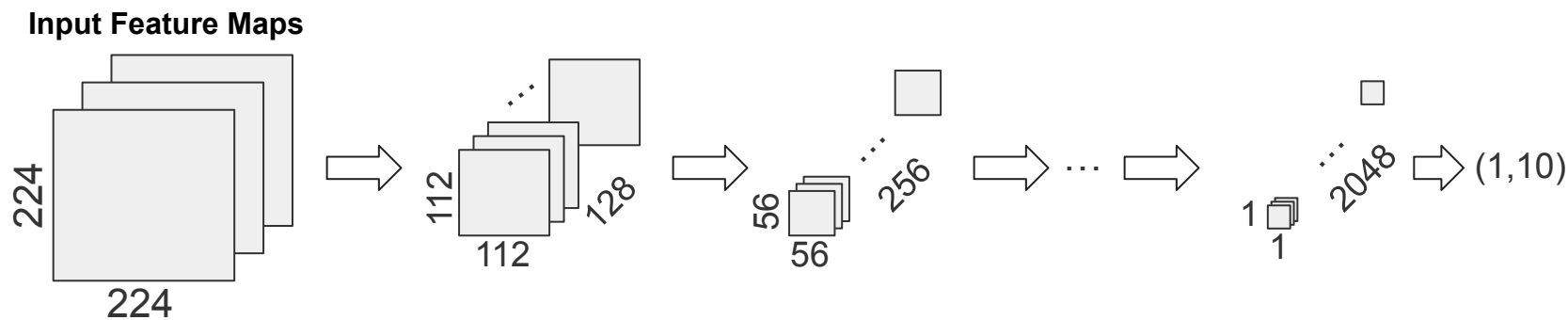
Fully Connected Layers

- Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular neural networks.



- Normally used in the last several layers to generate the classification results.

CNN Architecture for Image Classification Task

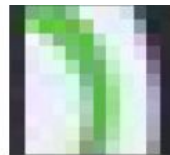


- For image classification task, during the forward propagation of CNN, the spatial size reduces while the number of channels increases.

Interpretation of Convolutional Features

- Each layer progressively extracts higher level features of the input image, until the last layer which aggregates all the high-level abstraction and makes a final decision.
- Early CNN layers tend to focus on detecting the local features (e.g., edge or corner in the image), whereas later layers usually look for the high-level abstractions (e.g., shapes of the object in the image)

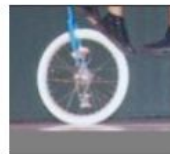
First
layer



Second
layer



Third
layer



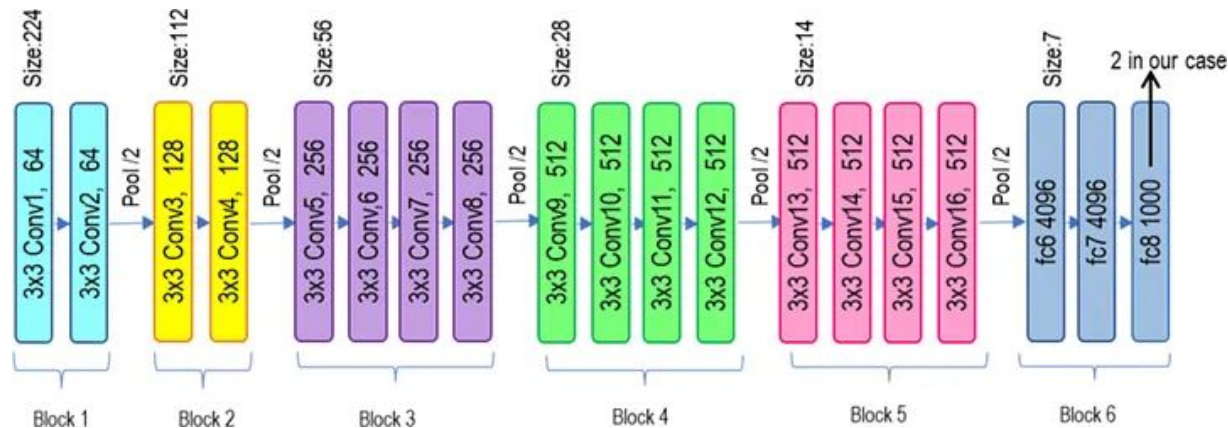
Fifth
layer



Topics

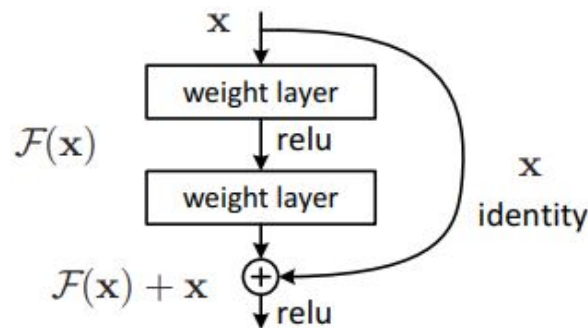
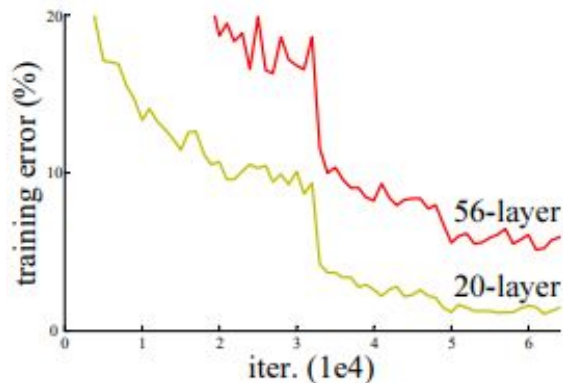
- Convolutional Neural Network
 - Basic building blocks
 - Popular CNN architectures
 - VGG
 - ResNet
 - MobileNet
 - ShuffleNet
 - SqueezeNet
 - DenseNet
 - EfficientNet
 - ConvNext
 - CNN architectures for other vision tasks
 - Image Segmentation, Object Detection

VGG



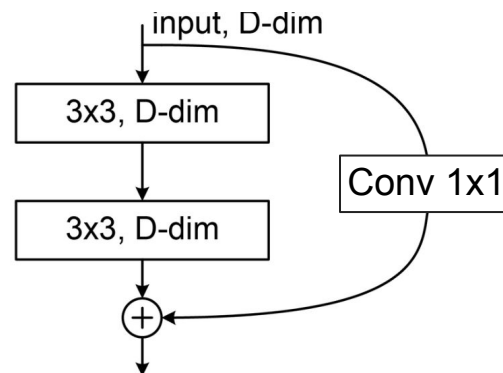
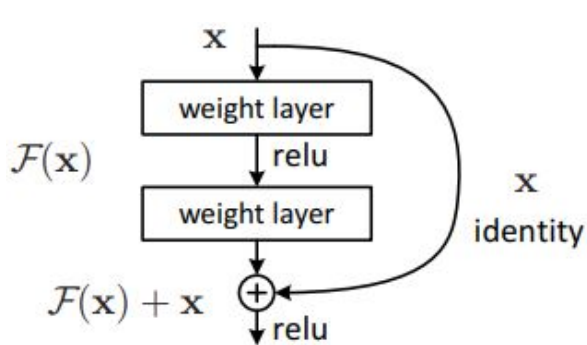
- The **key contribution** was demonstrating that **depth and the use of very small convolutional kernels (3×3)** were crucial for dramatically improving image recognition performance, establishing a simple and scalable architecture that became foundational in deep learning research.
- Achieves 75%-76% accuracy on ImageNet, which is much higher than other networks at that time (AlexNet: 62.5%).

ResNet



- When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated and then degrades rapidly.
- By introducing the residual link, we reduce the complexity of the learning process by ensuring that the performance is at least as good as the shallower DNN.

ResNet



$$y = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}$$

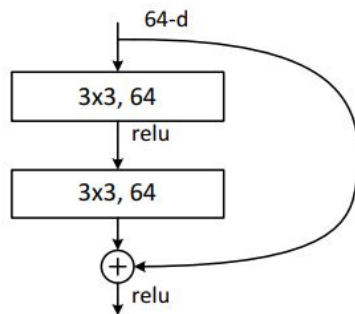
$$y = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s \mathbf{x}$$

- A straightforward strided convolutional layer may also be added to both branches when subsampling the output.

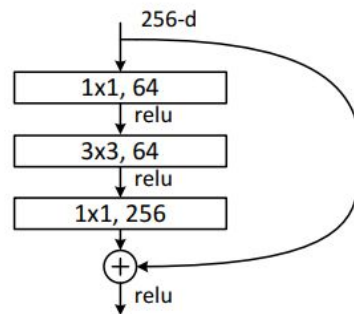
ResNet Bottleneck Design

Total params:

$$3 \times 3 \times 64 \times 64 + \\ 3 \times 3 \times 64 \times 64 = 73728$$



Basic block



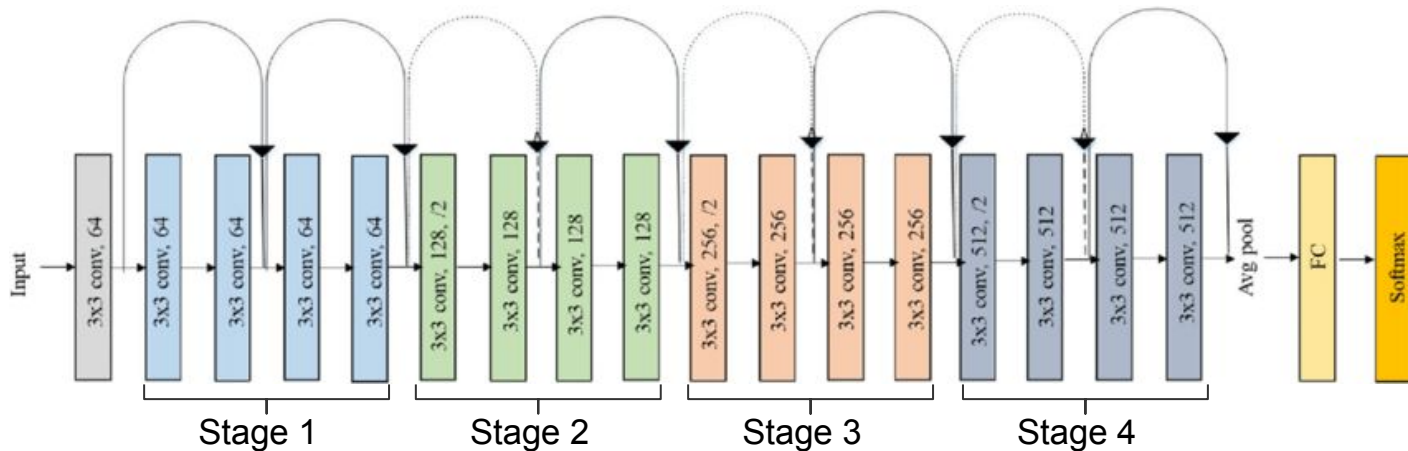
Bottleneck block

Total params:

$$1 \times 1 \times 256 \times 64 + 3 \times 3 \times 64 \times 64 \\ + 1 \times 1 \times 64 \times 256 = 69632$$

- For deeper ResNet, the bottleneck block is used.
- The three layers are 1×1 , 3×3 , and 1×1 convolutions, where the 1×1 layers reduces the output channel dimension.

ResNet 18



- ResNet-18 is partitioned into several stages, across two consecutive stages, the output channels doubles, and the spatial size is 2x2 subsampled.

ResNet

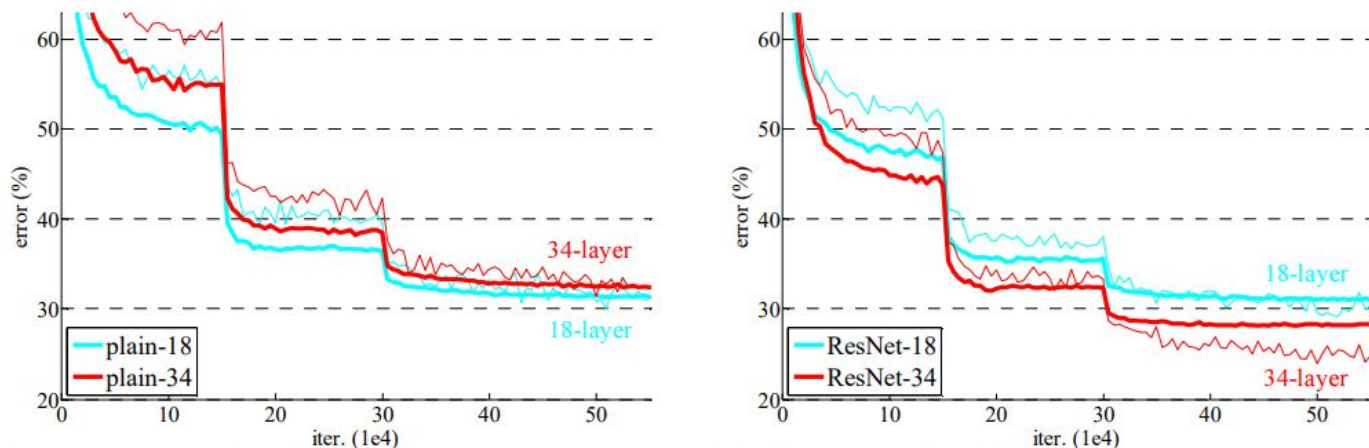


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

ResNet Performance

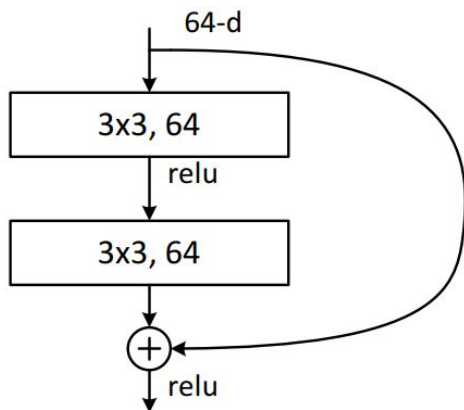
Performance on ImageNet

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	25.03

Performance on CIFAR-10

method			error (%)
Maxout [10]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [35]	19	2.5M	8.39
Highway [42, 43]	19	2.3M	7.54 (7.72±0.16)
Highway [42, 43]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	6.43 (6.61±0.16)
ResNet	1202	19.4M	7.93

ResNet Implementation

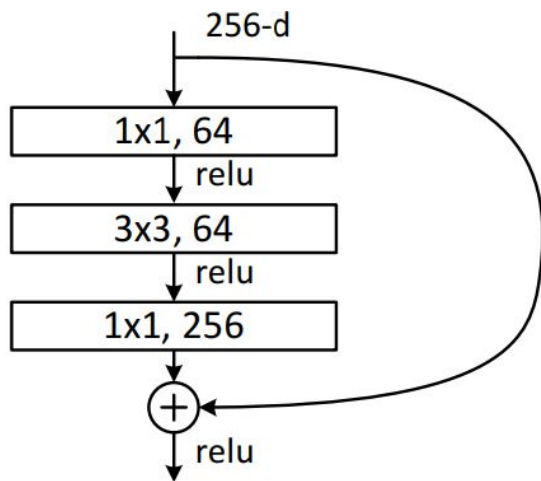


```
class BasicBlock(nn.Module):
    expansion = 1
    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

ResNet Implementation



```
class Bottleneck(nn.Module):
    expansion = 4

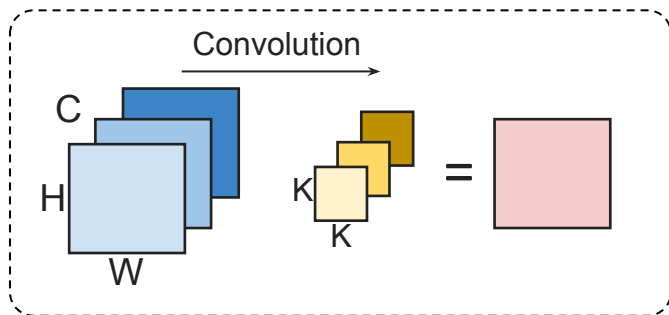
    def __init__(self, in_planes, planes, stride=1):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, self.expansion*planes, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(self.expansion*planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes :
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

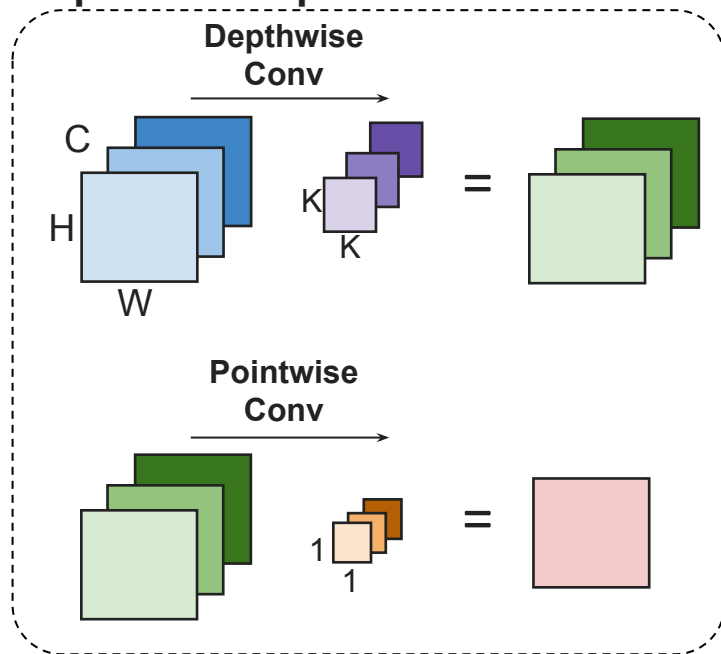
    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```


MobileNet

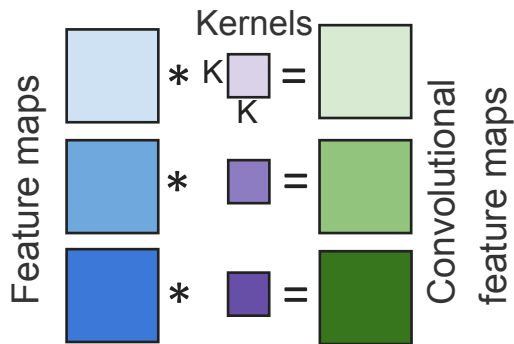
Standard Convolution



Depthwise Separable Convolution

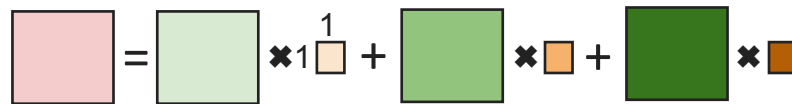


MobileNet



Step 1 Depthwise Convolution

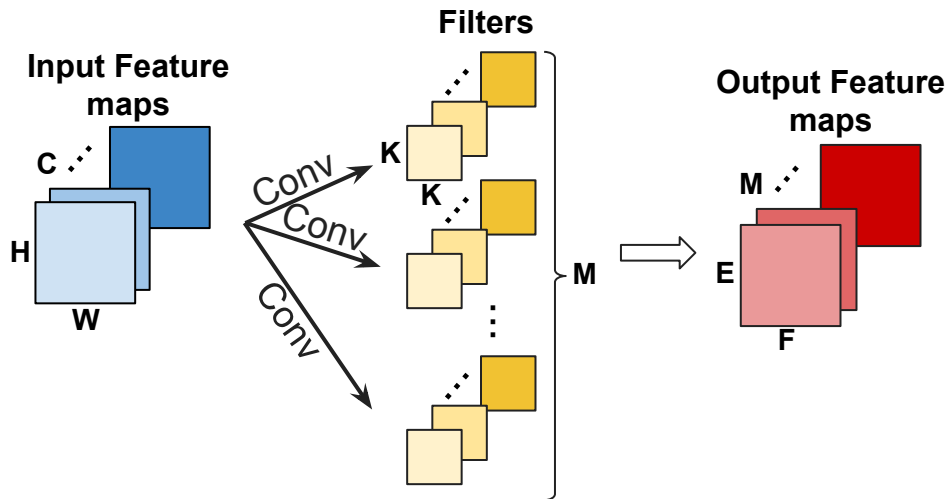
- Each kernel moves across the spatial dimensions of feature maps in the input activations, analyzing the information within those spatial dimensions.



Step 2 Pointwise Convolution

- The information from each feature maps are then aggregated by multiplying with the weight in the pointwise conv kernel and summing the Convolutional feature maps together.
- A bias may be introduced.

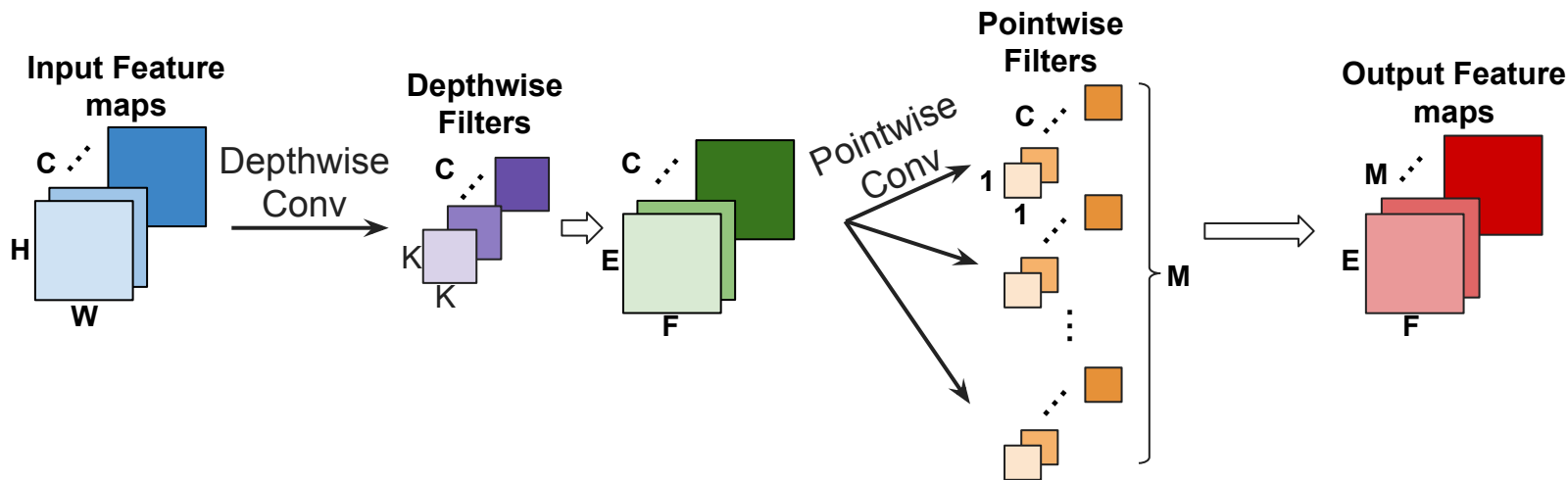
Standard Convolution



- Number of MACs: $M \times K \times K \times C \times E \times F$
- Storage cost:
 $32 \times (M \times C \times K \times K + C \times H \times W + M \times E \times F)$

C : number of input channels
 H, W : size of the input feature maps
 M : number of weight filters
 K : weight kernel size
 E, F : size of the output feature maps

Depthwise Separable Convolution

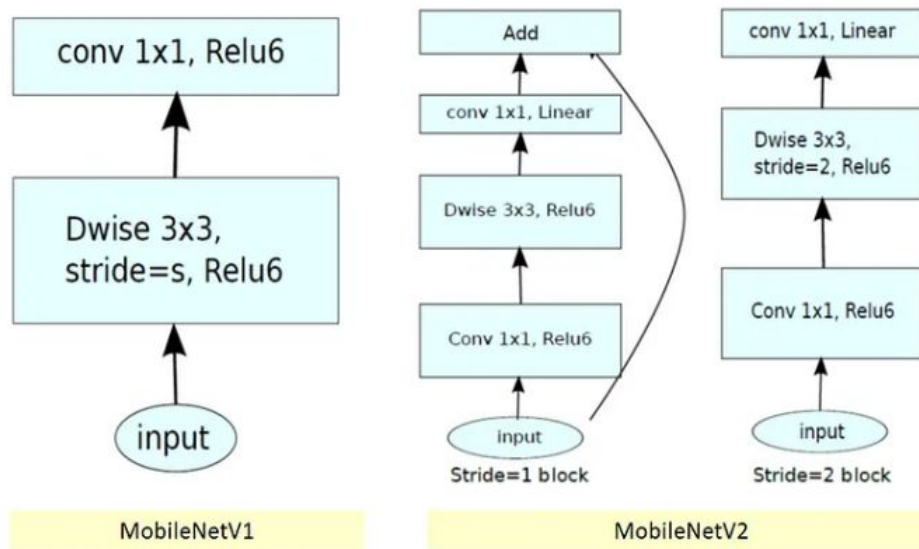


- Number of MACs: $K \times K \times C \times E \times F + M \times C \times E \times F$
- Storage cost: $32 \times (C \times H \times W + C \times K \times K + C \times E \times F + M \times C + M \times E \times F)$

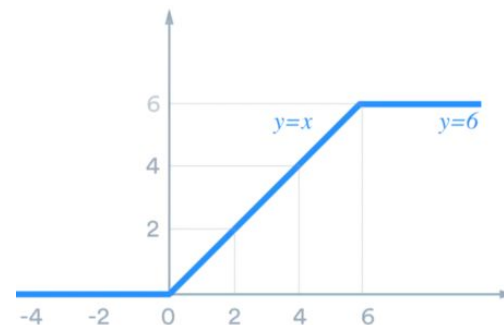
Why Depthwise Conv is Cheaper?

- Number of MACs for depthwise separable Conv: $K \times K \times C \times E \times F + M \times C \times E \times F$
 - Number of MACs for standard Conv: $M \times K \times K \times C \times E \times F$
 - When M is large the computational saving is about $K \times K$ (9) times.
-
- With a batch size of B, number of MACs are:
 - Number of MACs: $B \times K \times K \times C \times E \times F + B \times M \times C \times E \times F$
 - Storage cost: $32 \times (B \times C \times H \times W + C \times K \times K + B \times C \times E \times F + M \times C + B \times M \times E \times F)$

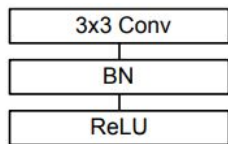
MobileNet-V2



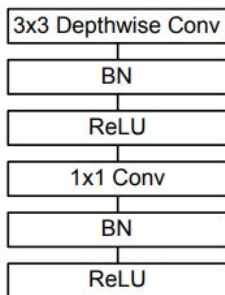
- Add residual link between the blocks.
- Adopt ReLU6 replace ReLU.



MobileNet Implementation



Standard



MobileNet

```
self.depthwise_conv = nn.Conv2d(
    in_channels,
    in_channels,
    (3, 3),
    stride=stride,
    padding=1,
    groups=in_channels,
)
self.bn1 = nn.BatchNorm2d(in_channels)

self.relu1 = nn.ReLU6() if use_relu6 else nn.ReLU()

# Pointwise conv
self.pointwise_conv = nn.Conv2d(in_channels, out_channels, (1, 1))
self.bn2 = nn.BatchNorm2d(out_channels)

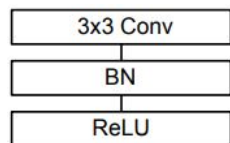
self.relu2 = nn.ReLU6() if use_relu6 else nn.ReLU()
```

```
def forward(self, x):
    """Perform forward pass."""

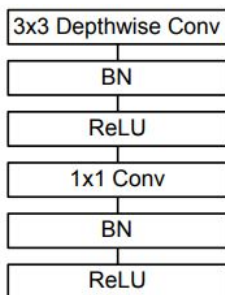
    x = self.depthwise_conv(x)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.pointwise_conv(x)
    x = self.bn2(x)
    x = self.relu2(x)

    return x
```

MobileNet V2 Implementation



Standard

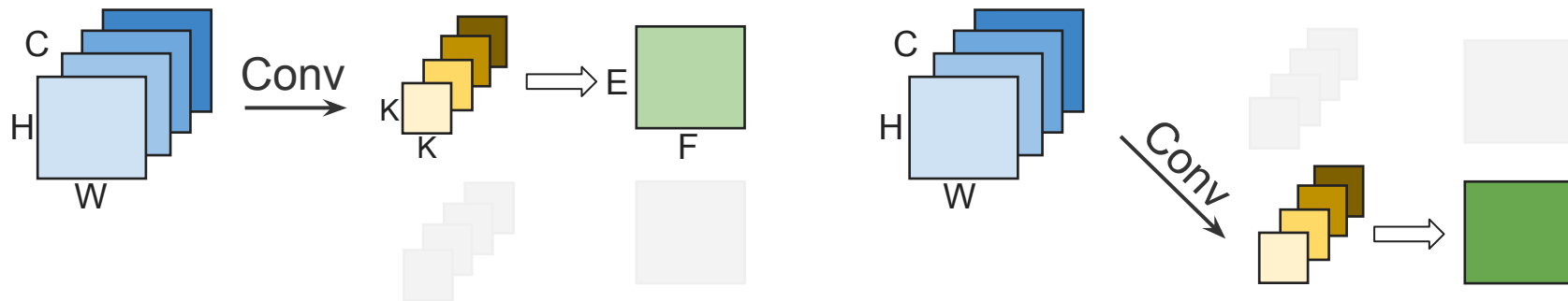


MobileNet

```
self.conv1 = (  
    ConvNormReLUBlock(in_channels, hidden_channels, (1, 1))  
    if in_channels != hidden_channels  
    else nn.Identity()  
)  
self.depthwise_conv = ConvNormReLUBlock(  
    hidden_channels,  
    hidden_channels,  
    (3, 3),  
    stride=stride,  
    padding=1,  
    groups=hidden_channels,  
)  
self.conv2 = ConvNormReLUBlock(  
    hidden_channels, out_channels, (1, 1), activation=nn.Identity  
)
```

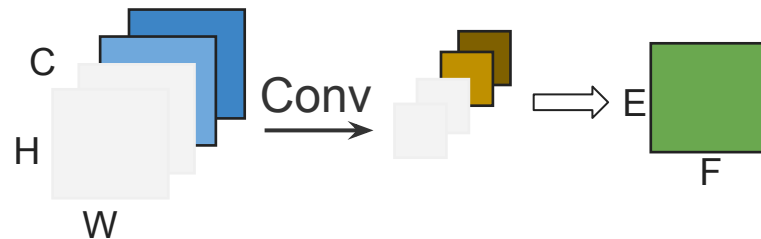
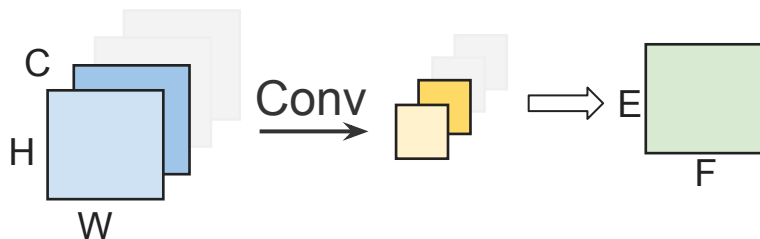
```
def forward(self, x):  
    """Perform forward pass."""  
  
    identity = x  
  
    x = self.conv1(x)  
    x = self.depthwise_conv(x)  
    x = self.conv2(x)  
  
    if self.residual:  
        x = torch.add(x, identity)  
  
    return x
```


Group Convolution



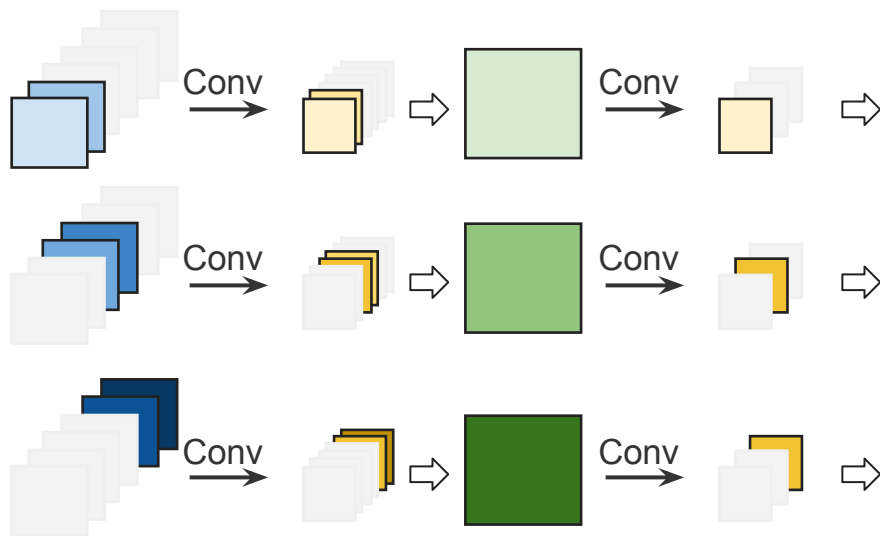
- The original MAC: $E \times F \times K \times K \times C \times M$

Group Convolution



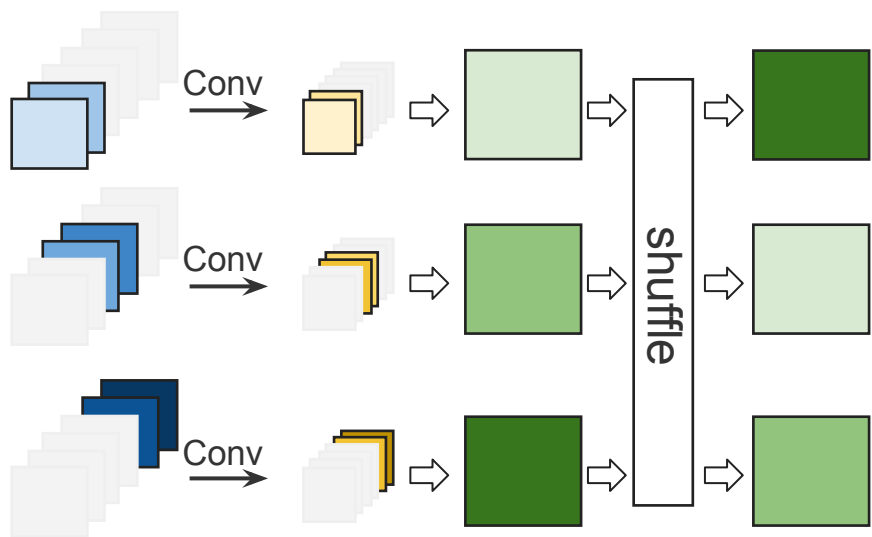
- Group size = 2
- Each group of feature maps within the input only convolved with partial weight kernels.
- This will lead to a large saving on memory consumption and computational cost.
- The number of MAC: $E \times F \times K \times K \times C \times M/G$

ShuffleNet



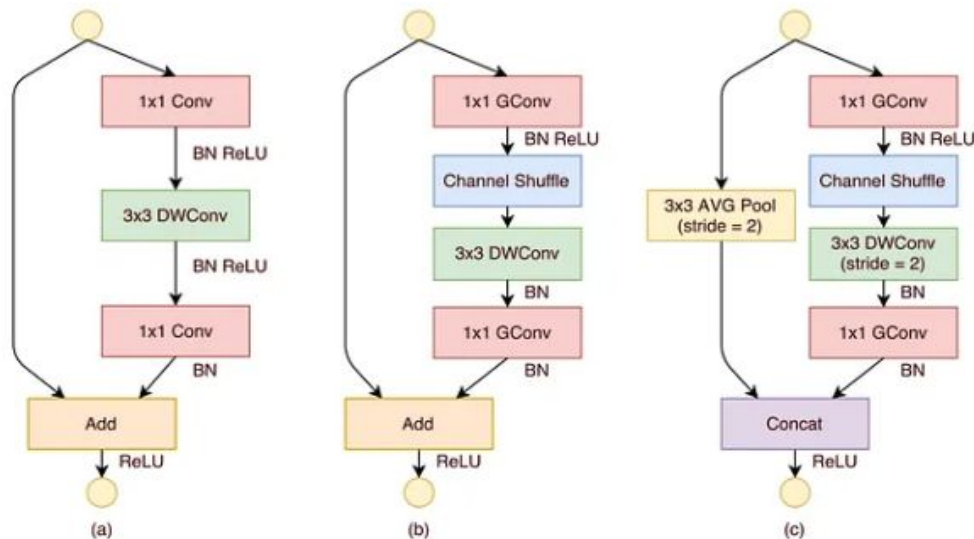
- Group convolution prevents feature maps from different groups from exchanging information.

ShuffleNet



- The shuffle operation is used to exchange the information across the groups.
- The shuffle operation with group convolution can replace the conventional full-channel convolution without noticeable accuracy degradation.
- A predetermined pattern is applied for the shuffling operations.

ShuffleNet



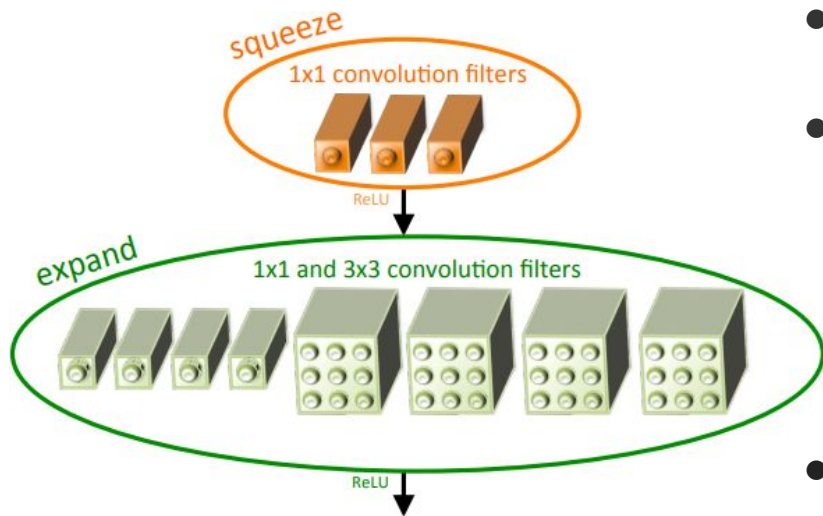
ShuffleNet

Model	Cls err. (% , no shuffle)	Cls err. (% , shuffle)	Δ err. (%)
ShuffleNet 1x ($g = 3$)	34.5	32.6	1.9
ShuffleNet 1x ($g = 8$)	37.6	32.4	5.2
ShuffleNet 0.5x ($g = 3$)	45.7	43.2	2.5
ShuffleNet 0.5x ($g = 8$)	48.1	42.3	5.8
ShuffleNet 0.25x ($g = 3$)	56.3	55.0	1.3
ShuffleNet 0.25x ($g = 8$)	56.5	52.7	3.8

Table 3. ShuffleNet with/without channel shuffle (*smaller number represents better performance*)

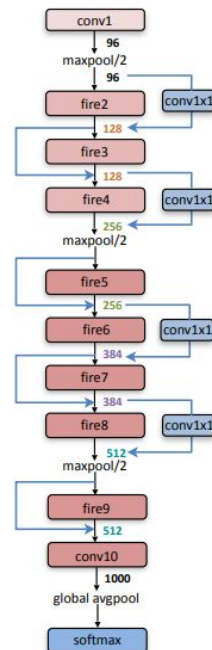
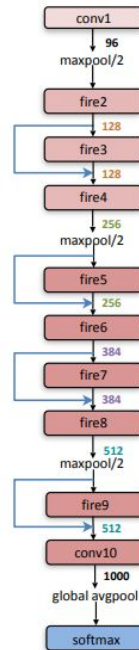
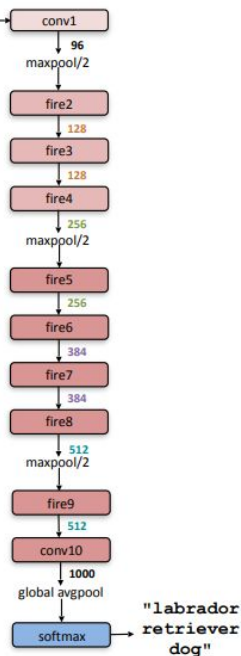
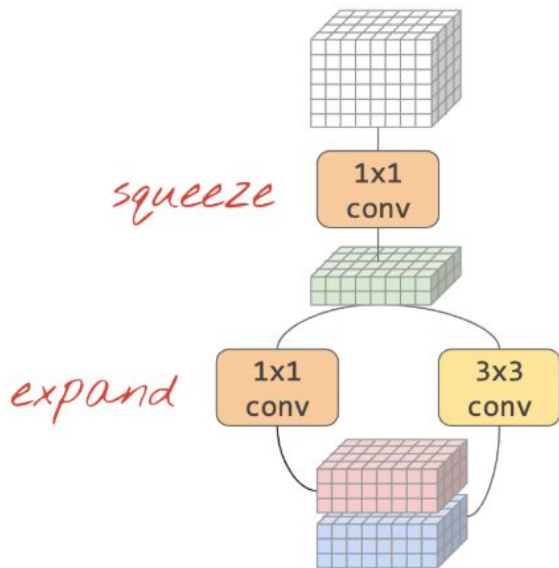
- G is the group size, $a\times$ is the scaling factor on number of channels.
- Shuffling operation can greatly improve the accuracy.

SqueezeNet

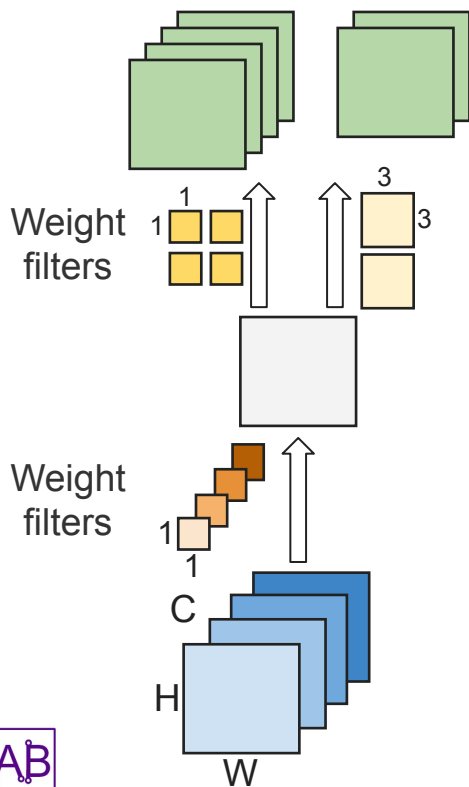


- Achieves great accuracy with 50x smaller parameters than other baselines (4.8MB).
- Some strategies:
 - Replace 3x3 filters with 1x1 filters.
 - Decrease the number of input channels to 3x3 filters.
 - Downsample late in the network so that convolution layers have large activation maps.
- Aims to reduce the CNN parameter size, not computational cost.

SqueezeNet



SqueezeNet



```
class fire(nn.Module):
```

```
    def __init__(self, inplanes, squeeze_planes, expand_planes):
        super(fire, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, squeeze_planes, kernel_size=1, stride=1)
        self.bn1 = nn.BatchNorm2d(squeeze_planes)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(squeeze_planes, expand_planes, kernel_size=1, stride=1)
        self.bn2 = nn.BatchNorm2d(expand_planes)
        self.conv3 = nn.Conv2d(squeeze_planes, expand_planes, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(expand_planes)
        self.relu2 = nn.ReLU(inplace=True)
```

```
    # using MSR initialization
```

```
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            n = m.kernel_size[0] * m.kernel_size[1] * m.in_channels
            m.weight.data.normal_(0, math.sqrt(2./n))
```

```
    def forward(self, x):
```

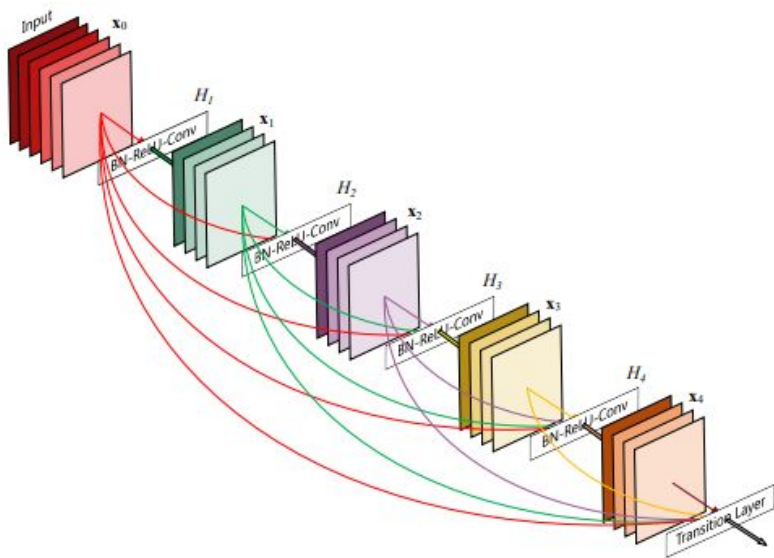
```
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        out1 = self.conv2(x)
        out1 = self.bn2(out1)
        out2 = self.conv3(x)
        out2 = self.bn3(out2)
        out = torch.cat([out1, out2], 1)
        out = self.relu2(out)
        return out
```

SqueezeNet

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	50x	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	363x	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	510x	57.5%	80.3%

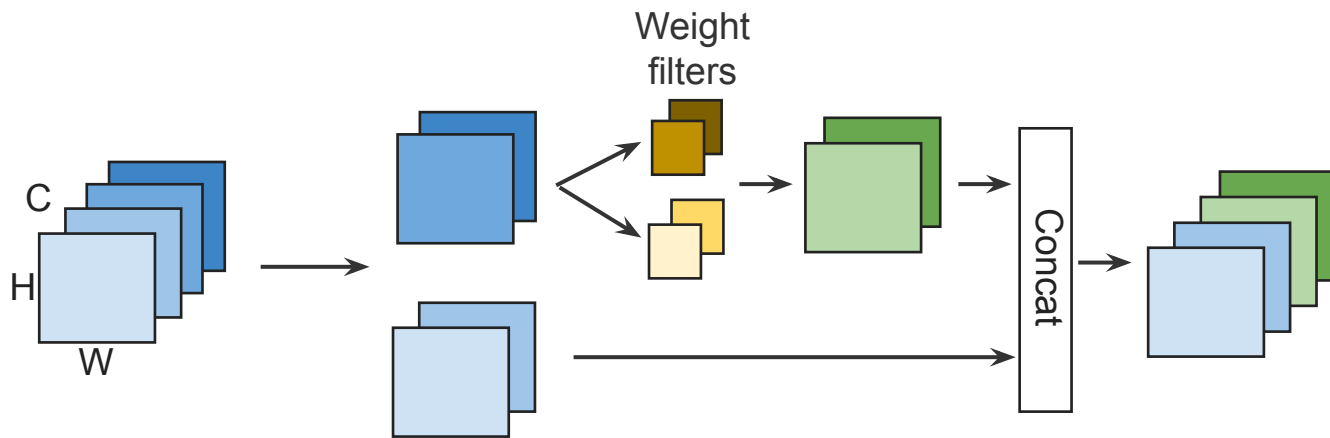
- Achieve a comparable performance as AlexNet, but still suboptimal compare against other architectures.
- ResNet 50: 100MB, Vision Transformer base> 300MB.

DenseNet

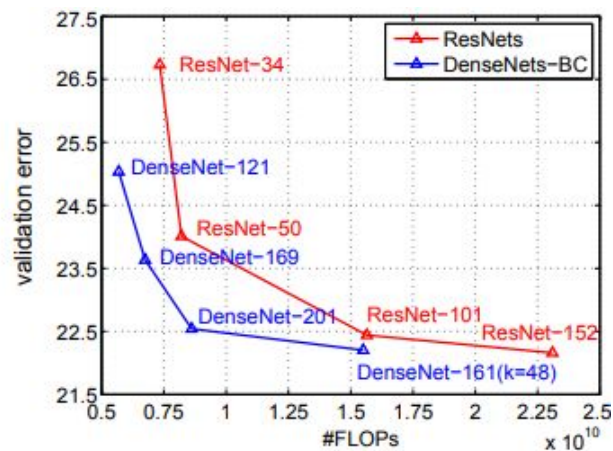
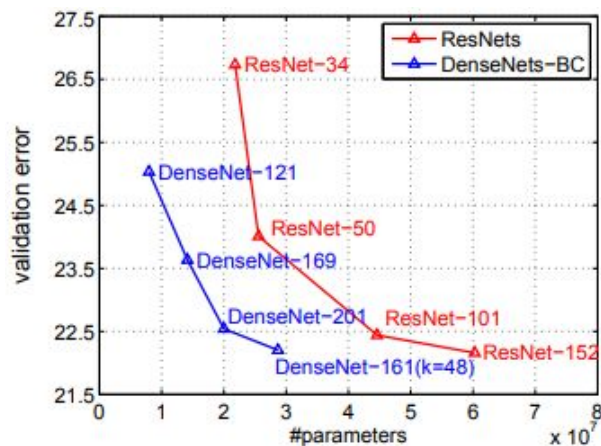


- ResNet:
$$\mathbf{x}_\ell = H_\ell(\mathbf{x}_{\ell-1}) + \mathbf{x}_{\ell-1}$$
- DenseNet:
$$\mathbf{x}_\ell = H_\ell([\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{\ell-1}])$$
- $H(\cdot)$ is the function of batch normalization, followed by ReLU and 3x3 Convolution.

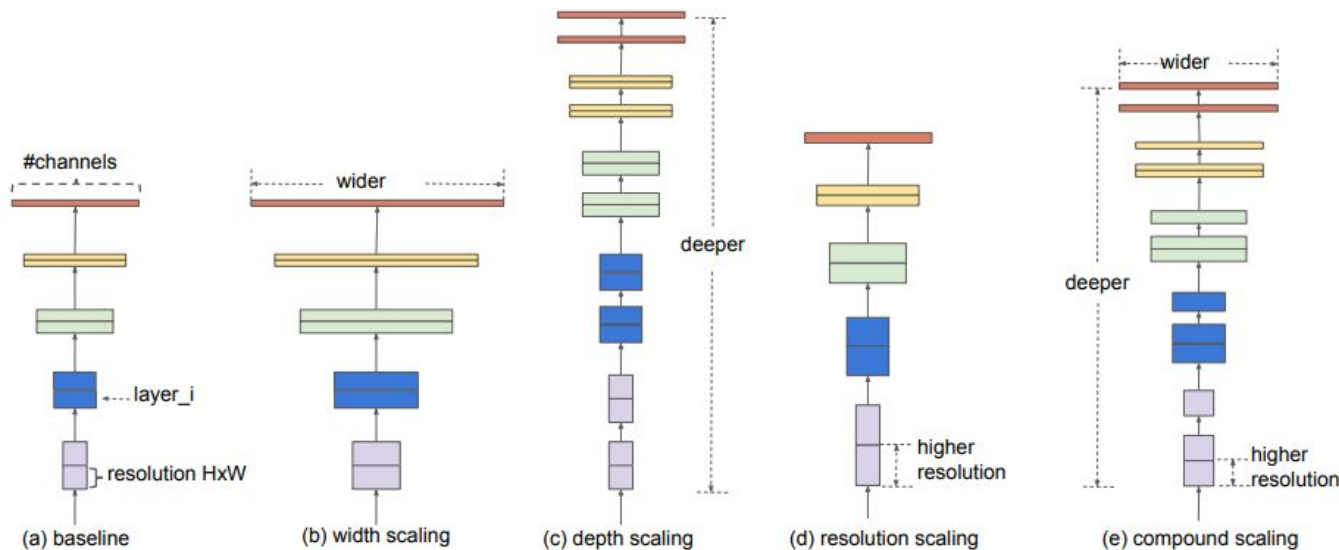
DenseNet



DenseNet

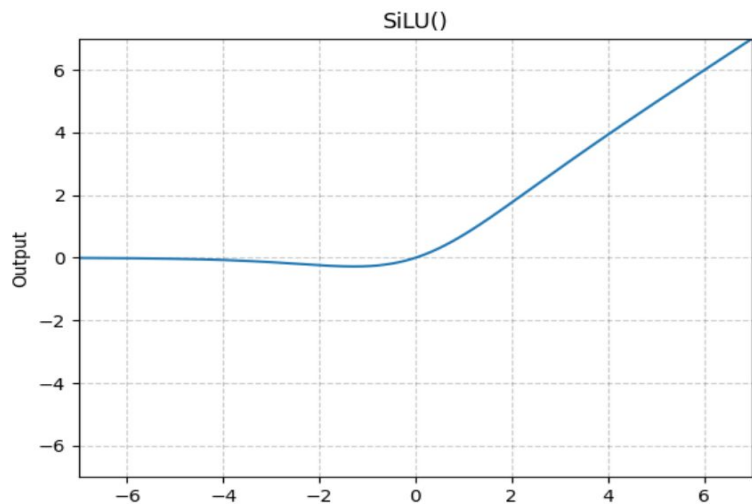


EfficientNet

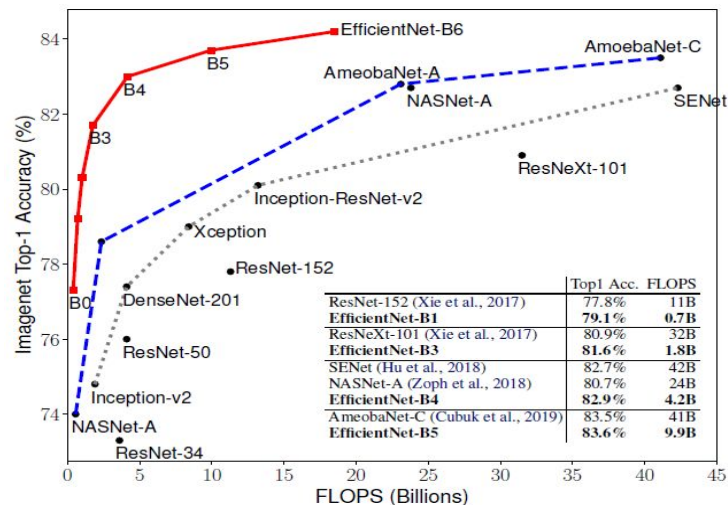


- It is critical to balance all dimensions of network width/depth/resolution, and surprisingly such balance can be achieved by simply scaling each of them with constant ratio.

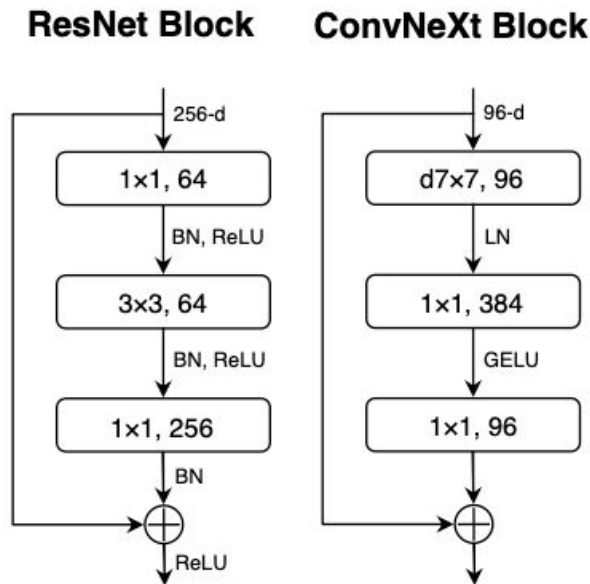
EfficientNet



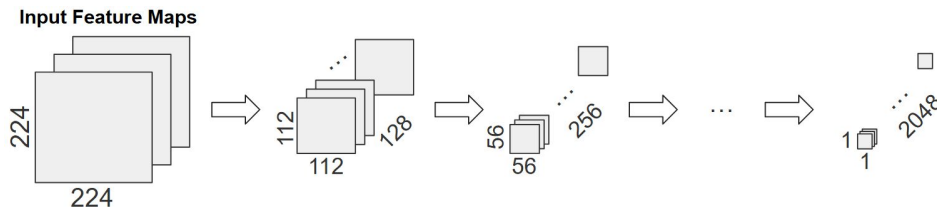
- SiLU is used in the EfficientNet architecture.
- $\text{SiLU}(x) = x * \sigma(x)$



ConvNext

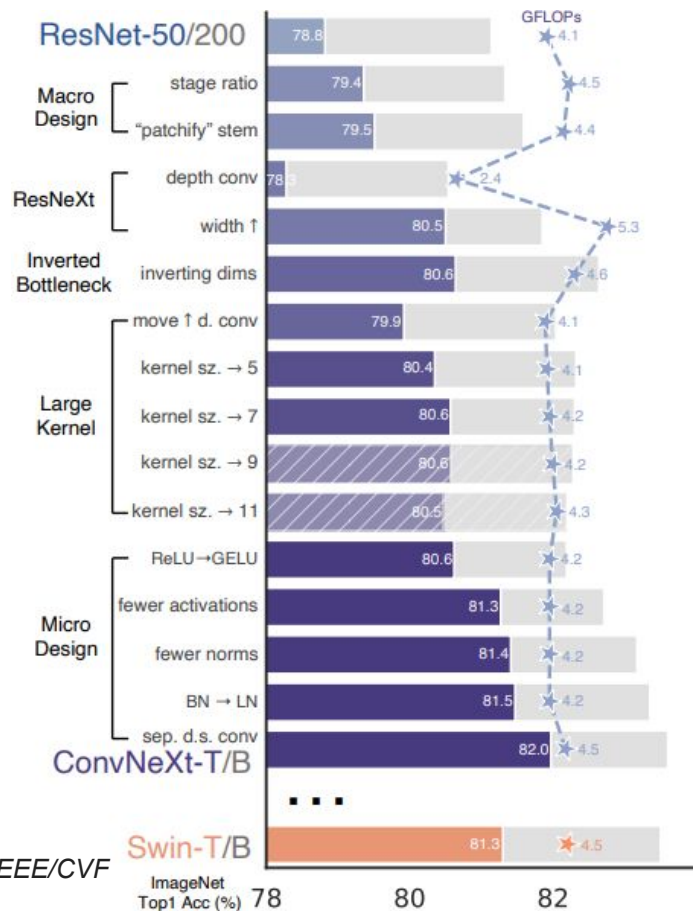


- Leverage the insight of vision transformer (Swin-T) to enhance the performance of CNN.
- Some major changes to change ResNet 50 to ConvNext 50:
 - Change number of blocks in each stage from (3, 4, 6, 3) in ResNet-50 to (3, 3, 9, 3).
 - Use depthwise separable convolution
 - Large convolutional kernel.
 - Replacing ReLU with GELU
 - Substituting BN with LN.



ConvNext

- For larger Swin Transformers, the ratio is 1:1:9:1. Following the design → we adjust the number of blocks in each stage from (3, 4, 6, 3) in ResNet-50 to (3, 3, 9, 3).
- One of the most distinguishing aspects of vision Transformers is their non-local self-attention, which enables each layer to have a global receptive field, so we increase the window size to 7×7 .
- Replacing ReLU with GELU: One discrepancy between NLP and vision architectures is the specifics of which activation functions to use.



ConvNext

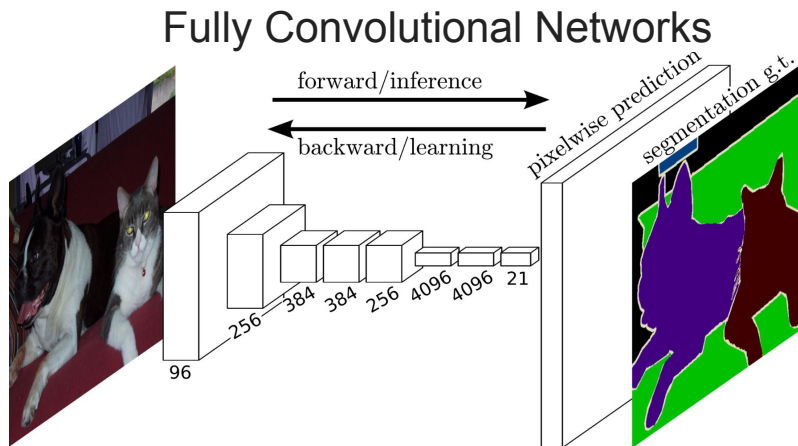
model	image size	#param.	FLOPs	throughput (image / s)	IN-1K top-1 acc.
ImageNet-1K trained models					
• RegNetY-16G [54]	224 ²	84M	16.0G	334.7	82.9
• EffNet-B7 [71]	600 ²	66M	37.0G	55.1	84.3
• EffNetV2-L [72]	480 ²	120M	53.0G	83.7	85.7
○ DeiT-S [73]	224 ²	22M	4.6G	978.5	79.8
○ DeiT-B [73]	224 ²	87M	17.6G	302.1	81.8
○ Swin-T	224 ²	28M	4.5G	757.9	81.3
• ConvNeXt-T	224 ²	29M	4.5G	774.7	82.1
○ Swin-S	224 ²	50M	8.7G	436.7	83.0
• ConvNeXt-S	224 ²	50M	8.7G	447.1	83.1
○ Swin-B	224 ²	88M	15.4G	286.6	83.5
• ConvNeXt-B	224 ²	89M	15.4G	292.1	83.8
○ Swin-B	384 ²	88M	47.1G	85.1	84.5
• ConvNeXt-B	384 ²	89M	45.0G	95.7	85.1
• ConvNeXt-L	224 ²	198M	34.4G	146.8	84.3
• ConvNeXt-L	384 ²	198M	101.0G	50.4	85.5

- ConvNext achieves a much better accuracy under the same amount of parameters and computation budgets.

Topics

- Convolutional Neural Network
 - Basic building blocks
 - Popular CNN architectures
 - VGG
 - ResNet
 - MobileNet
 - ShuffleNet
 - SqueezeNet
 - DenseNet
 - EfficientNet
 - ConvNext
 - CNN architectures for other vision tasks
 - Image Segmentation, Object Detection

CNNs for Other Tasks: Image Segmentation



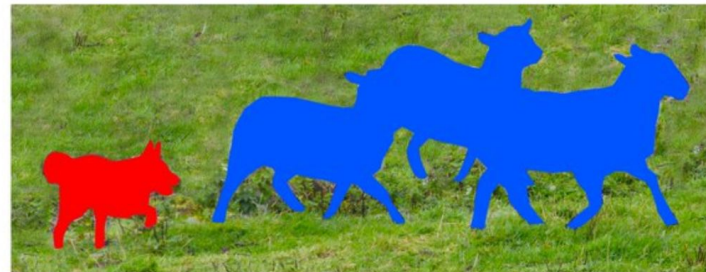
- A fully convolutional based DNN for image segmentation.
- Input: $H \times W \times 3 \rightarrow$ Output: $H \times W \times C$

- Image segmentation is a computer vision technique used to divide an image into multiple segments or regions, each representing a different object, part of an object, or background.
- The goal of image segmentation is to simplify or change the representation of an image into something more meaningful and easier to analyze.

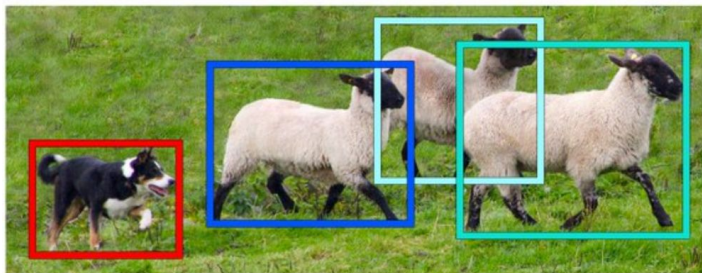
Segmentation



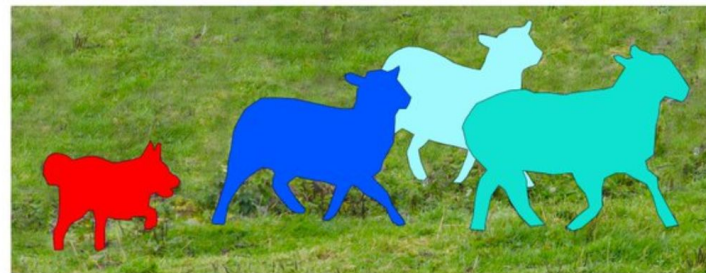
Image Recognition



Semantic Segmentation

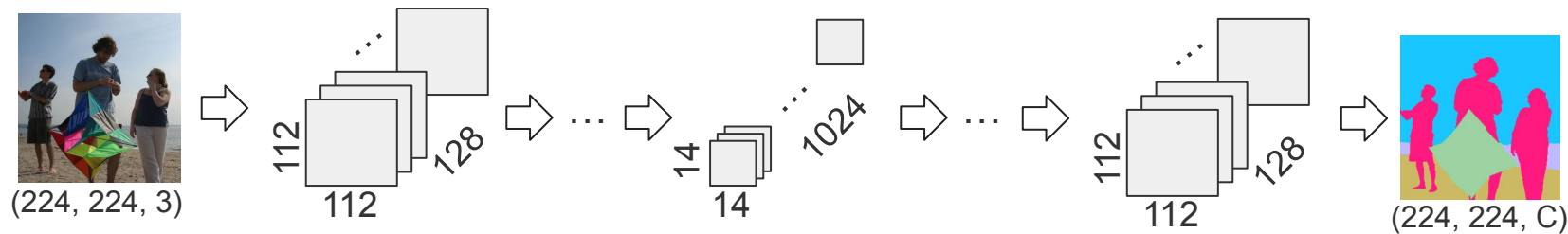


Object Detection



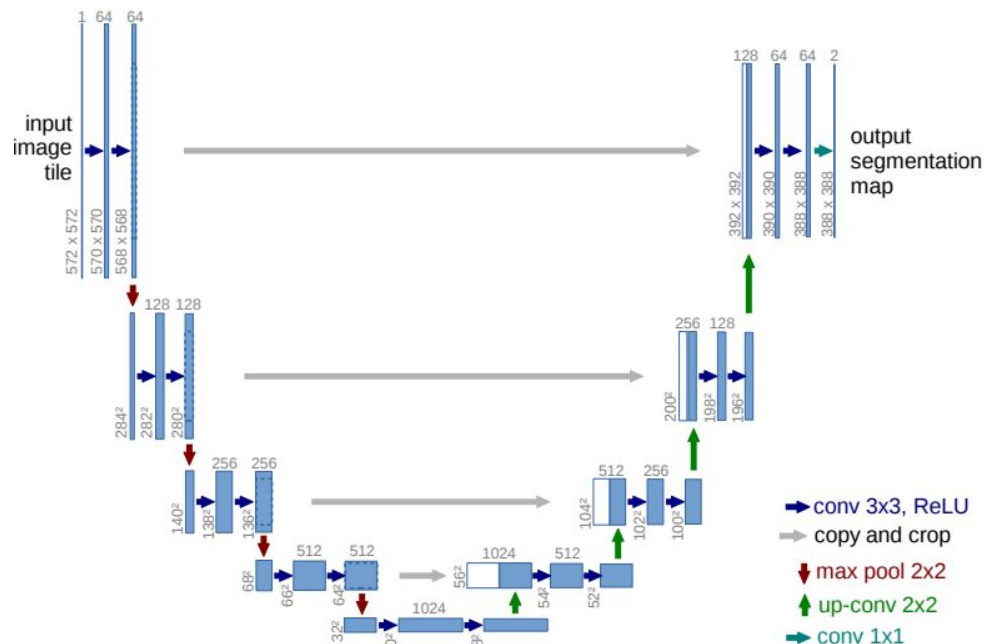
Instance Segmentation

Segmentation



- Segmentation is a pixel-level task in which each pixel is assigned an output label.
- The loss function (cross-entropy loss) is applied on each pixel.

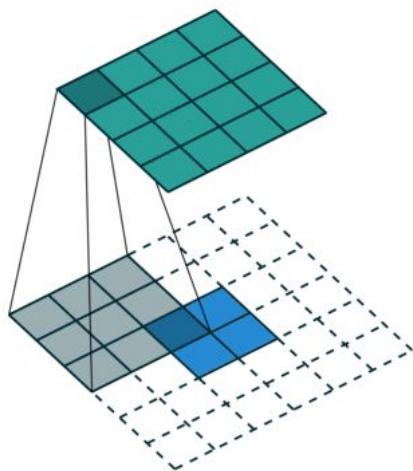
U-Net



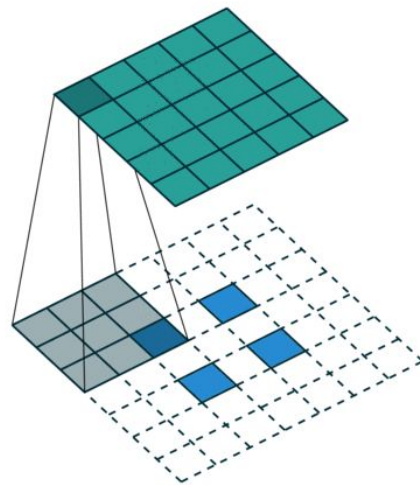
- The direct path sends feature maps from the encoder directly to the corresponding decoder layers, allowing the decoder to recover spatial precision.
- This stabilizes training and improves convergence.

Transposed Convolution

- To upsample the input, we can apply transposed convolution.

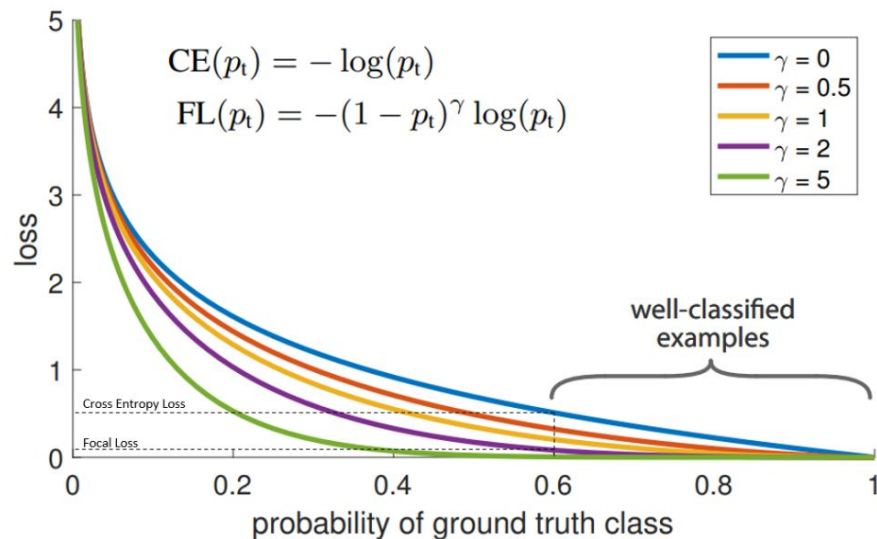


Stride = 1



Stride = 2

Focal Loss

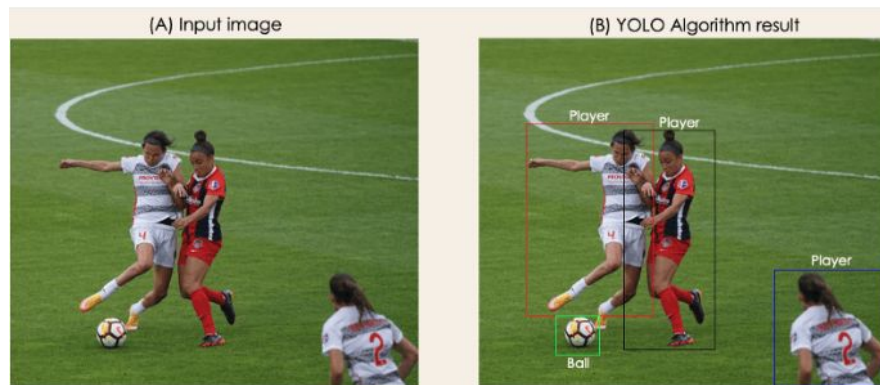
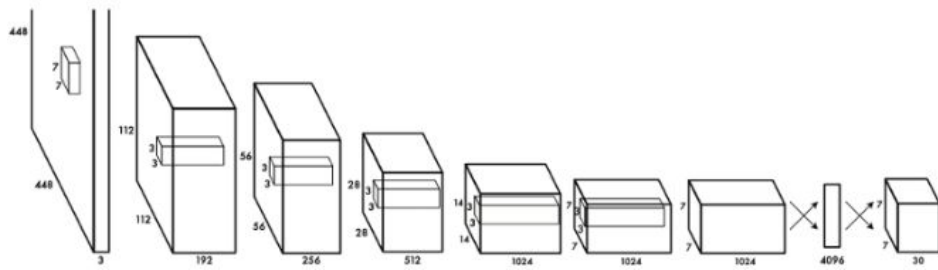


- A modified cross-entropy designed to perform better with class imbalance.
- Often used in the problem of object detection and image segmentation.
 - Down-weight easy examples and thus focus training on hard negatives

$$\text{FL}(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t)$$

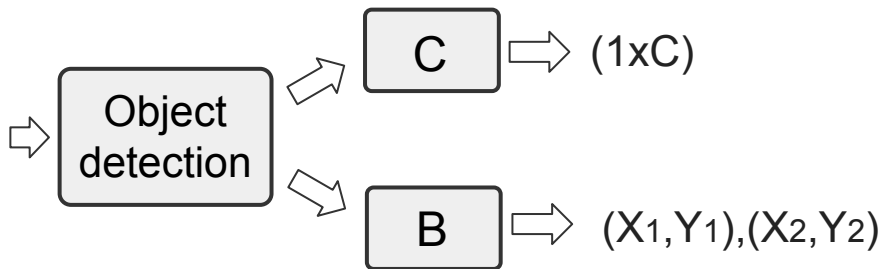
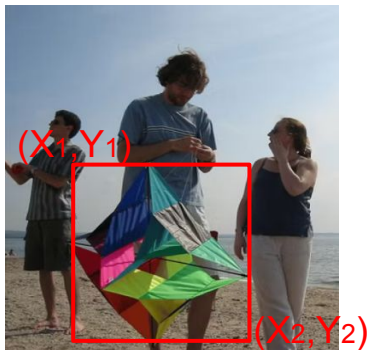
- γ controls the shape of the curve
- α controls the class imbalance and introduce weights to each class.

CNNs for Other Tasks: Object Detection



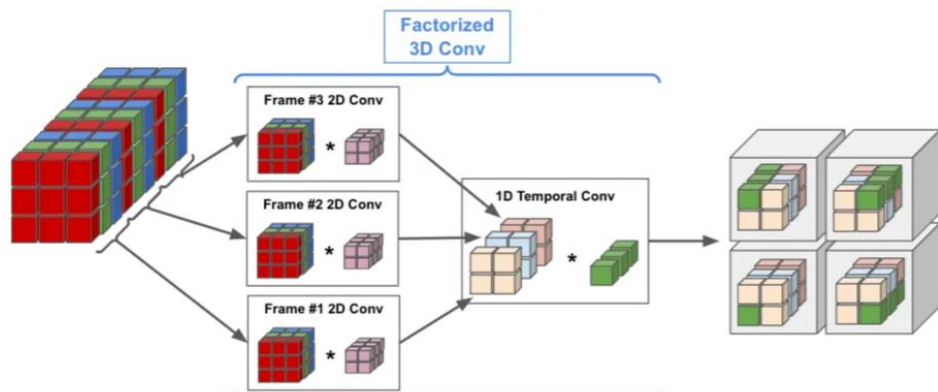
- NN will generate the likelihood of each anchor point and the coordinates of its bounding box.
- Another branch will produce the category of each bounding box

CNNs for Other Tasks: Object Detection



- The bounding box is defined by its top-left and bottom-right coordinates, and the object detection network also outputs a $1 \times C$ classification vector.

CNNs for Other Tasks: Video Processing



- To process video, we can concatenate the consecutive frames together and use 2D convolution to process it.

Presentations

- You Only Look Once: Unified, Real-Time Object Detection
 - Aman, Vanshika
- TSM: Temporal Shift Module for Efficient Video Understanding
 - Zuheng, Zewen