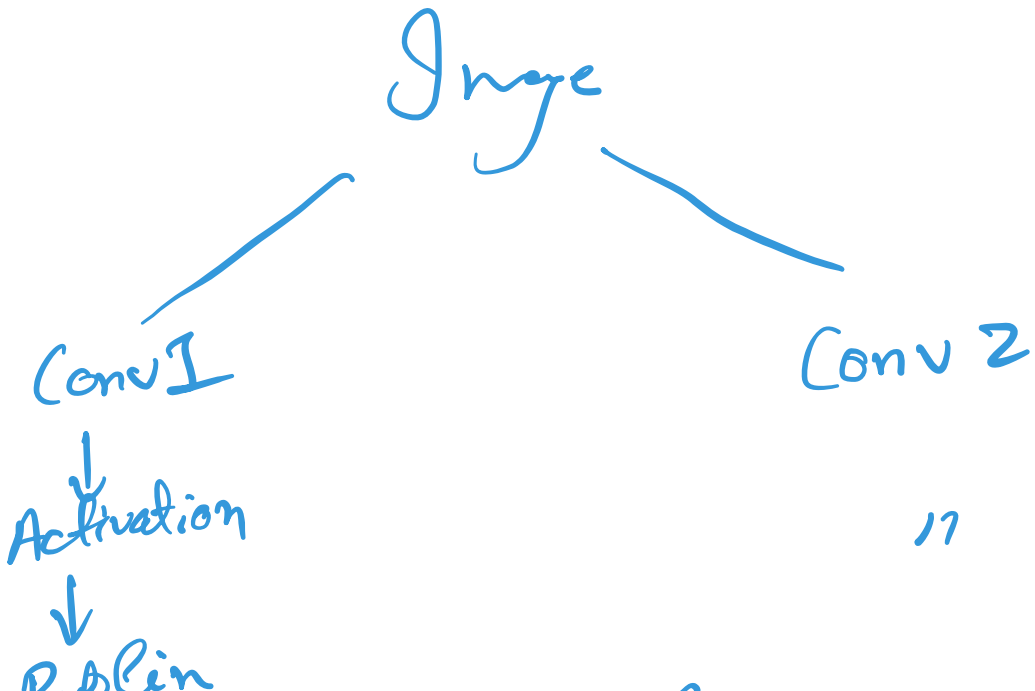
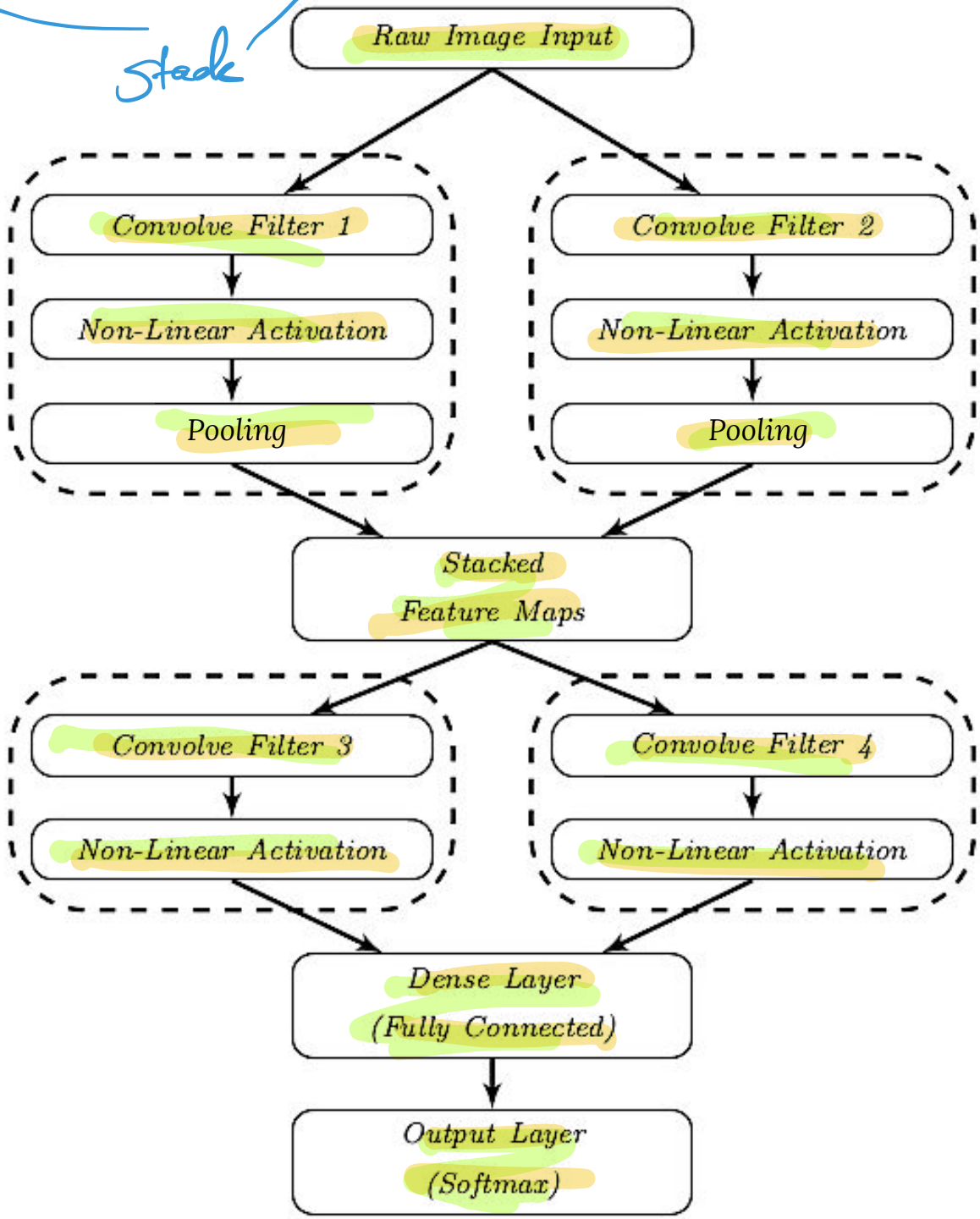
Deep learning for Computer Vision



Convolution Neural Networks



Schematic of typical sequences of layers in CNNs

Special Convolution (Depth-wise separable convolutions)

Ek single channel alag alag layer ka conv. lekar fir 1x1 se overall filters banaha do. i.e Depthwise

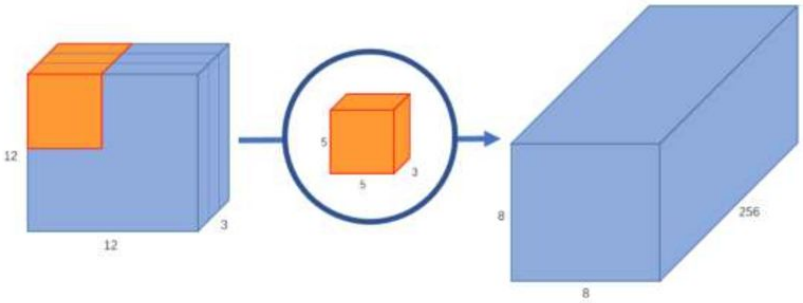
Separable convolutions.

Depth-wise separable convolutions

```
from keras.layers import Input, DepthwiseConv2D, Conv2D, BatchNormalization,
Activation

# Input layer for an image of size 12x12 with 3 channels
inputs = Input(shape=(12, 12, 3))

# Depthwise convolutional layer
depthwise_conv = DepthwiseConv2D(kernel_size=(5, 5), padding='valid')(inputs)
depthwise_conv = Activation('relu')(depthwise_conv)

# Pointwise convolutional layer
pointwise_conv = Conv2D(filters=256, kernel_size=(1, 1))(depthwise_conv)
pointwise_conv = Activation('relu')(pointwise_conv)
```

But why?

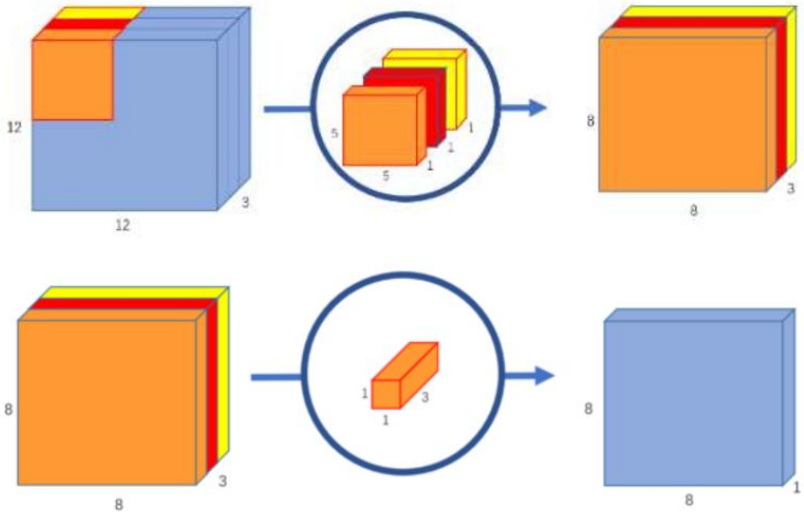
→ reductions in params as well as multiplications



- Each filter has a kernel size of 5x5x3 (there are 3 input channels). Plus one bias term for each filter.
- Total number of parameters in the Convolutional layer is 256x(5x5x3+1) = 19456.

Simple convolution

But why?



Depthwise Convolutional layer:

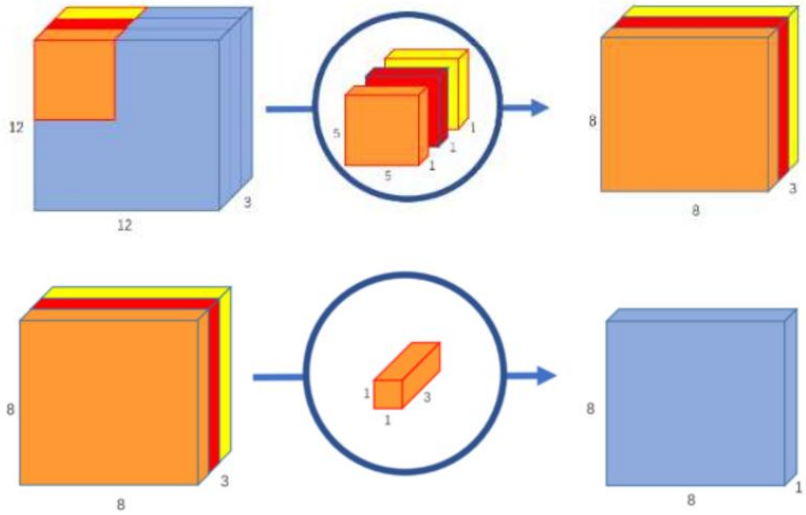
- Each channel of the input tensor is convolved separately using a 5x5 kernel.
- There are 3 input channels and a depth multiplier of 1, so there are 3 depthwise filters in total.

3 (5x5+1)

Pointwise Convolutional layer:

- There are 256 output filters, and the input has 3 channels.
- Each filter has a kernel size of 1x1x3 (as 3 input channels). Plus one bias term for each filter.

But why?



Depthwise Convolutional layer:

- Each channel of the input tensor is convolved separately using a 5x5 kernel.
- There are 3 input channels and a depth multiplier of 1, so there are 3 depthwise filters in total.
- Each depthwise filter has 5x5=25 parameters. Plus one bias term for each filter.
- Total number of parameters in the Depthwise Convolutional layer is 3x(25+1) = 78.

Pointwise Convolutional layer:

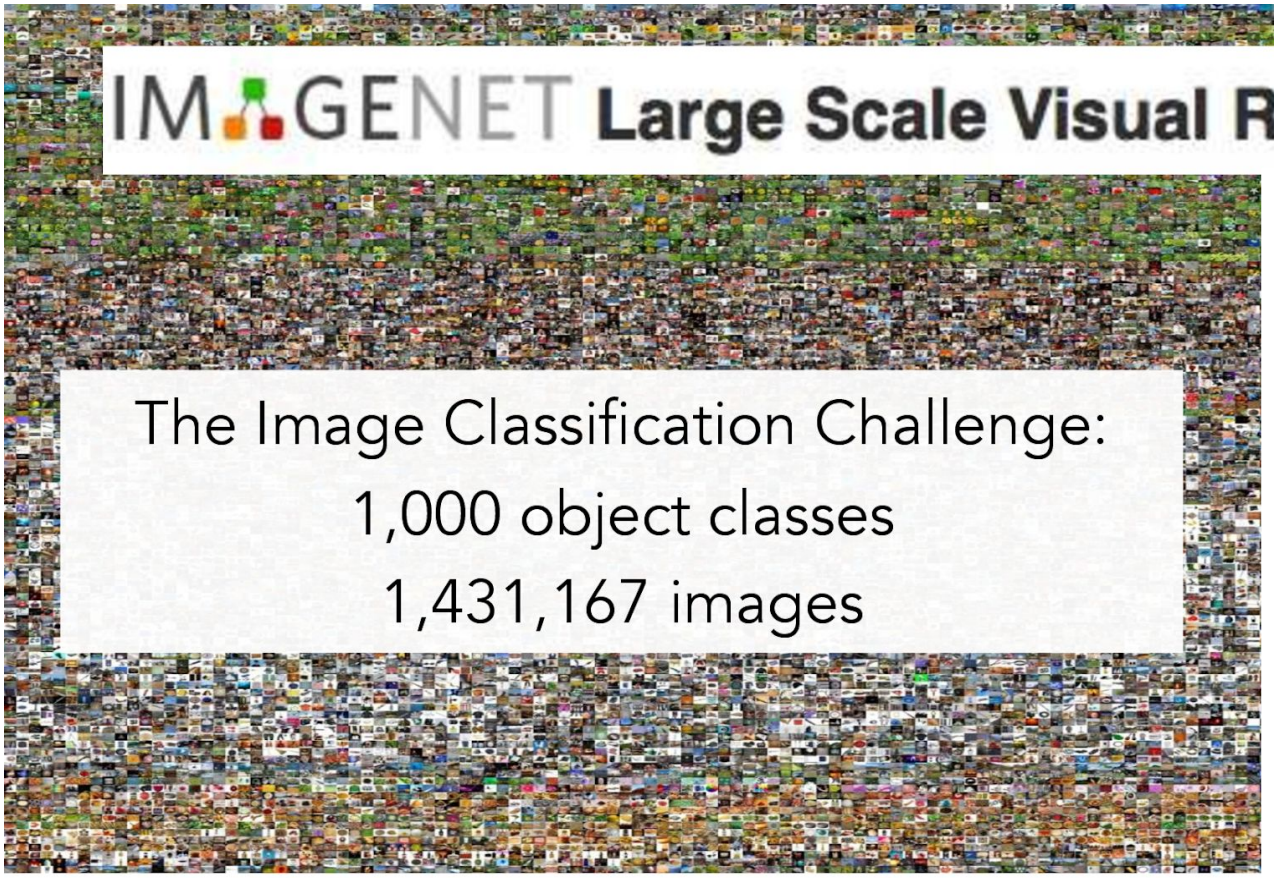
- There are 256 output filters, and the input has 3 channels.
- Each filter has a kernel size of 1x1x3 (as 3 input channels). Plus one bias term for each filter.
- Total number of parameters in the Pointwise Convolutional layer is 256x(1x1x3+1) = 1024.

256 (1x1x3 +1)

Depthwise Separable Convolution

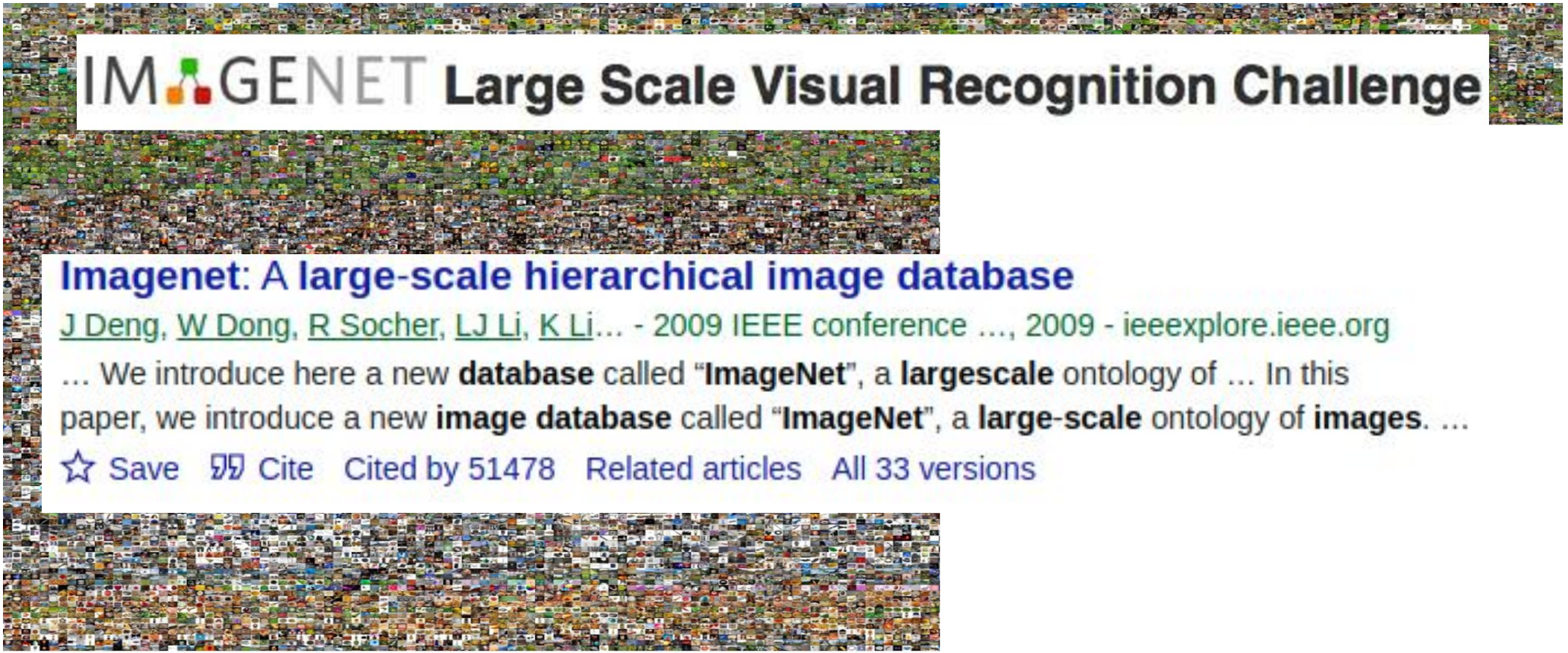
- **Fewer parameters**: Compared to a standard convolution, depthwise separable convolution requires fewer parameters to learn.
- **Computationally efficient**: The depthwise convolution requires fewer computations than a standard convolution
- **Improved performance**: Depthwise separable convolution can achieve similar or better performance than a standard convolution.

ImageNet



- ILSVRC 2017
- ILSVRC 2016
- ILSVRC 2015
- ILSVRC 2014
- ILSVRC 2013
- ILSVRC 2012
- ILSVRC 2011
- ILSVRC 2010

ImageNet

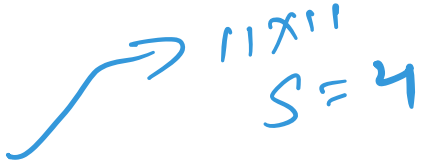

IMAGENET Large Scale Visual Recognition Challenge
Imagenet: A large-scale hierarchical image database
J Deng, W Dong, R Socher, LJ Li, K Li... - 2009 IEEE conference ..., 2009 - ieeexplore.ieee.org
... We introduce here a new database called "ImageNet", a largescale ontology of ... In this paper, we introduce a new image database called "ImageNet", a large-scale ontology of images. ...
Save Cite Cited by 51478 Related articles All 33 versions

ImageNet

IMAGENET Large Scale Visual Recognition Challenge

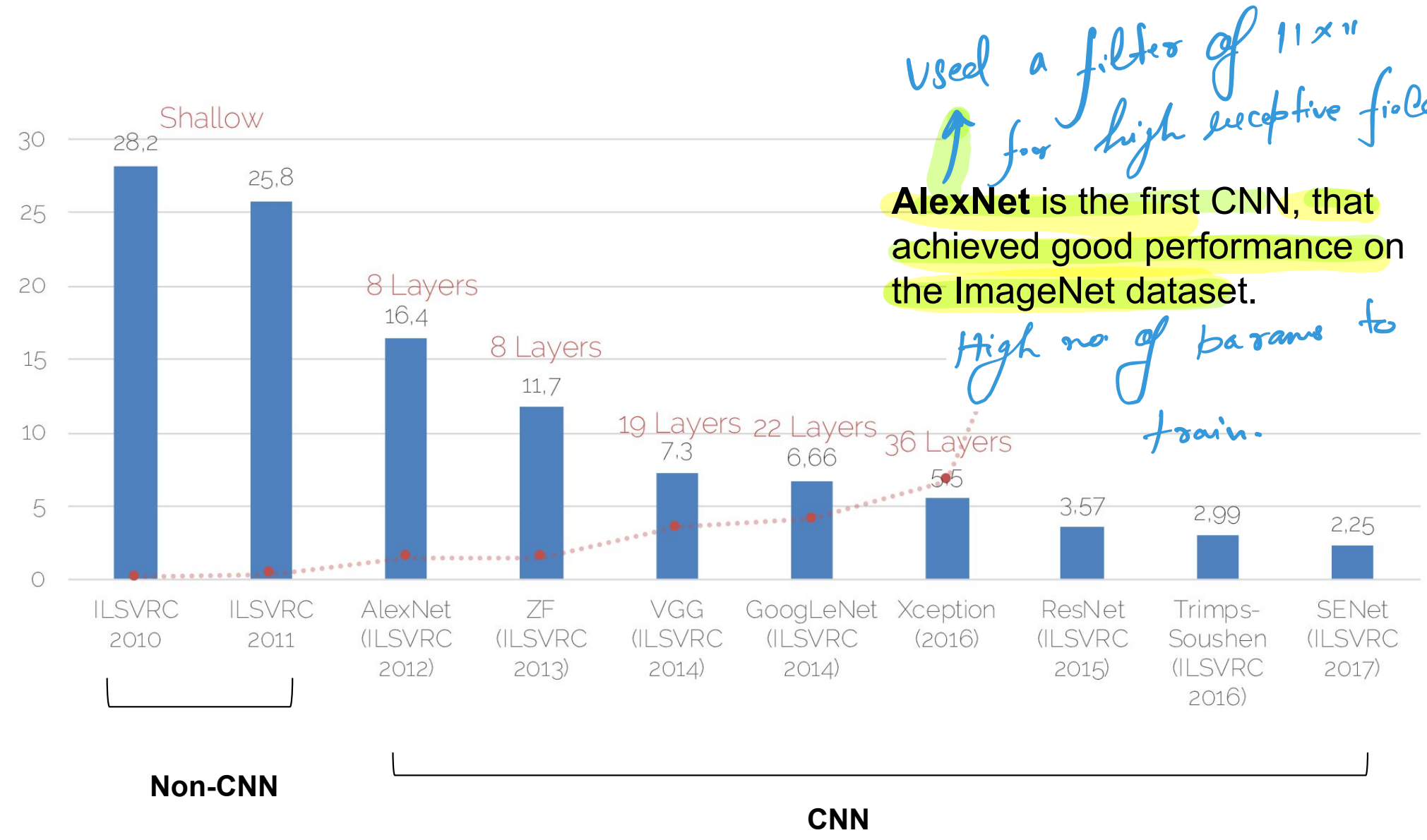
AI researcher **Fei-Fei Li** began working on the idea for ImageNet in 2006.

Classical Architecture



- Alexnet
- VGG Network
- ResNet
- Inception Net

Revolution of depth (ImageNet Benchmark)

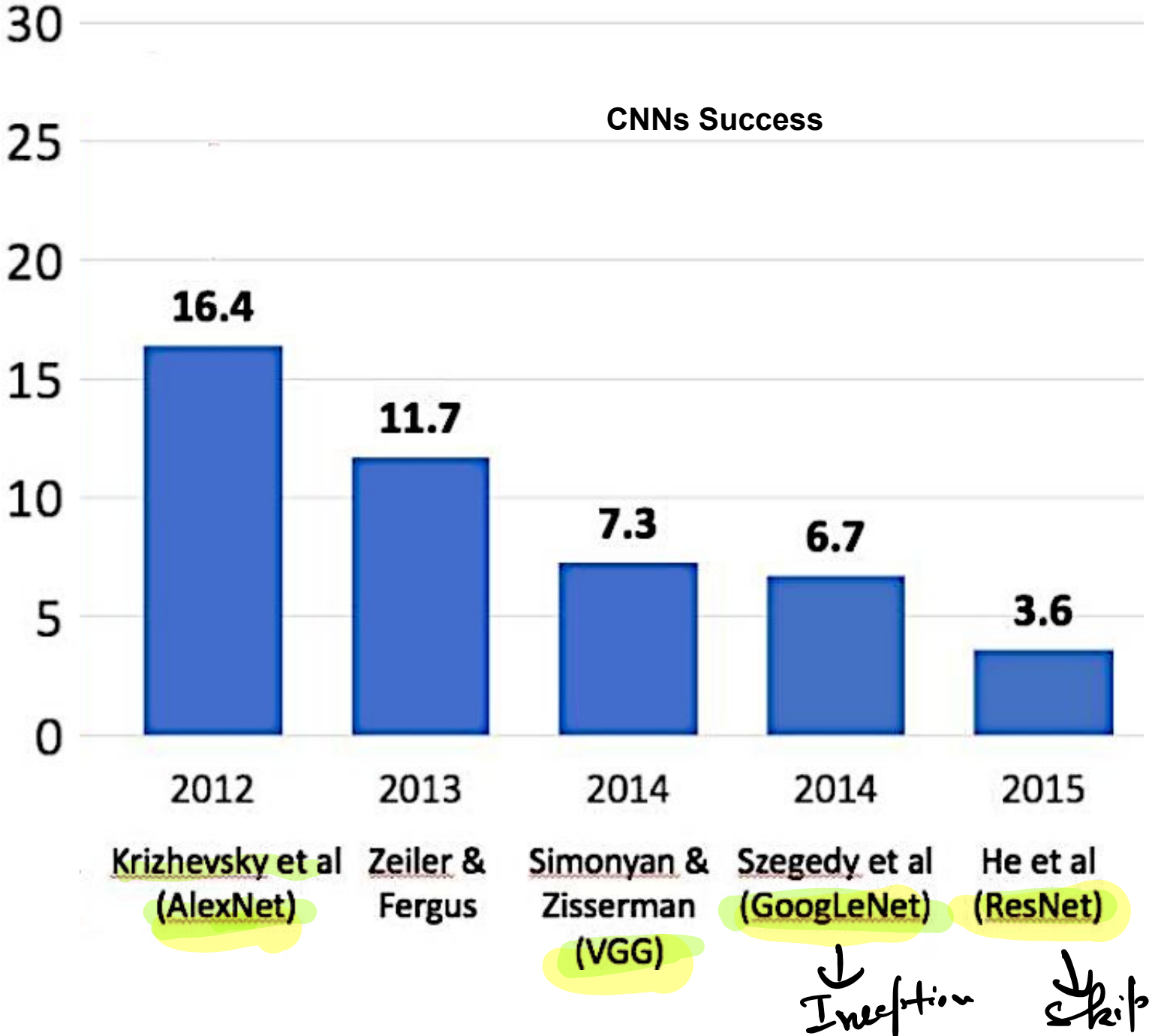


Used a filter of 11×11 for high receptive field.

AlexNet is the first CNN, that achieved good performance on the ImageNet dataset.

High no. of params to train.

Alexnet


CNNs Success
30
25
20
15
10
5
0
16.4
11.7
7.3
6.7
3.6
2012
2013
2014
2014
2015
Krizhevsky et al (AlexNet)
Zeiler & Fergus
Simonyan & Zisserman (VGG)
Szegedy et al (GoogLeNet)
He et al (ResNet)
Inception
Skip connections

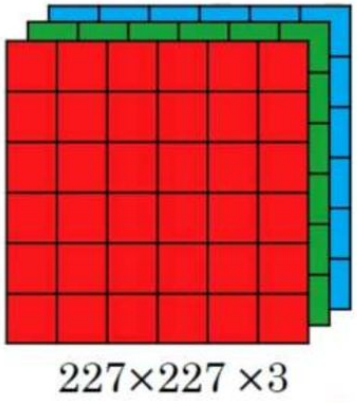
Alexnet

- The architecture of AlexNet was inspired by LeNet.
- AlexNet is a convolutional neural network (CNN) architecture designed by Alex Krizhevsky in 2012.

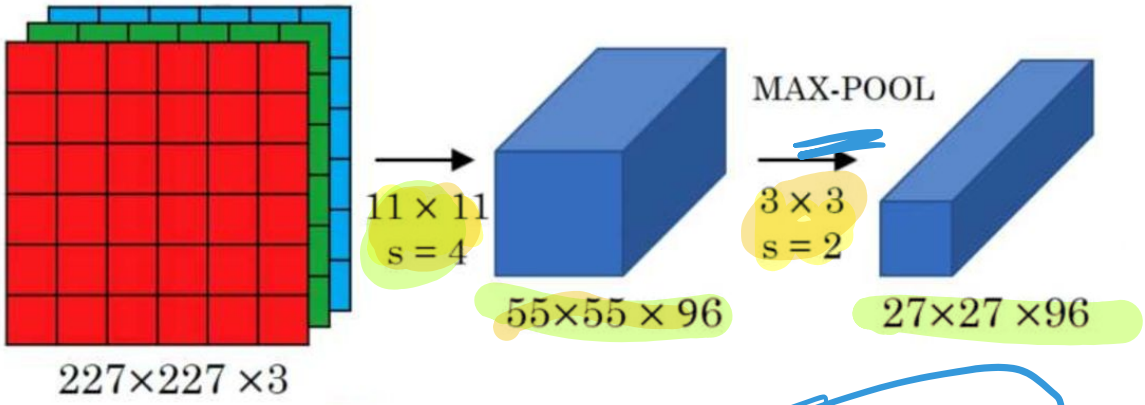
Alexnet

- The architecture of AlexNet was inspired by LeNet.
- AlexNet is a convolutional neural network (CNN) architecture designed by Alex Krizhevsky in 2012.
- AlexNet was one of the first CNN to achieve a good improvement in image classification performance on the ImageNet dataset.
- AlexNet helped to popularize deep learning and convolutional neural networks.

AlexNet



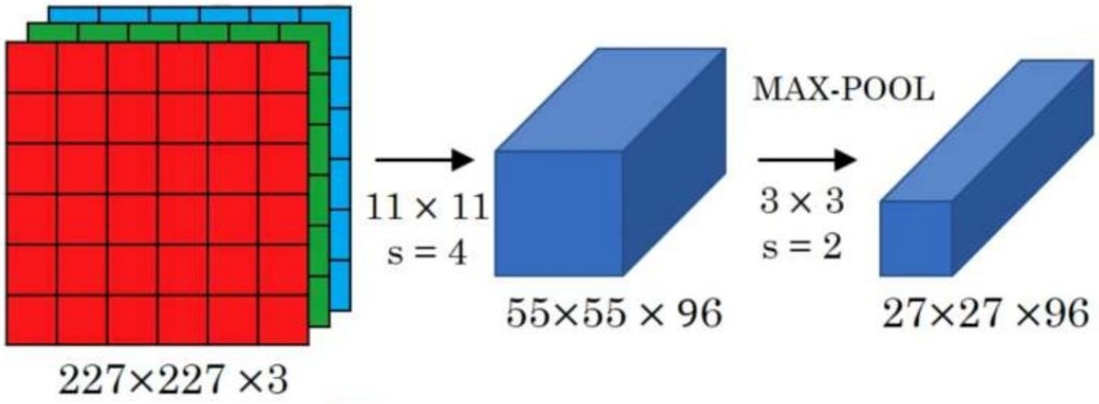
AlexNet



3×3
s=2

The first convolutional layer has 96 filters

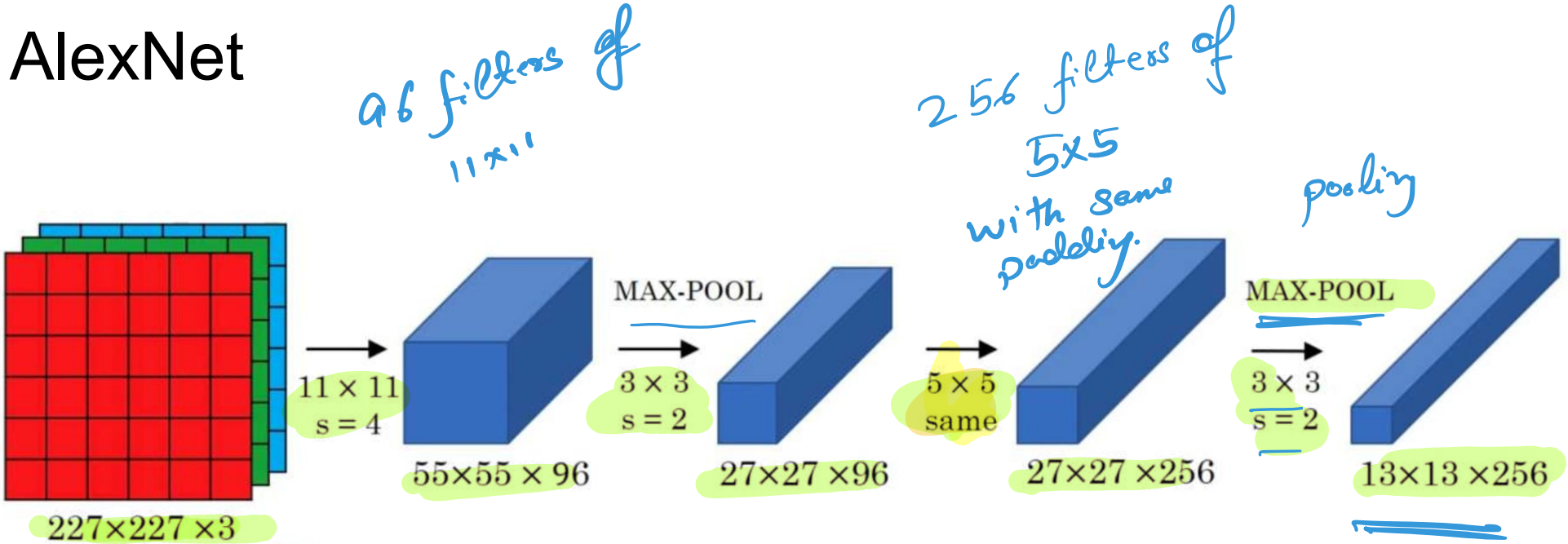
AlexNet



The first convolutional layer has 96 filters

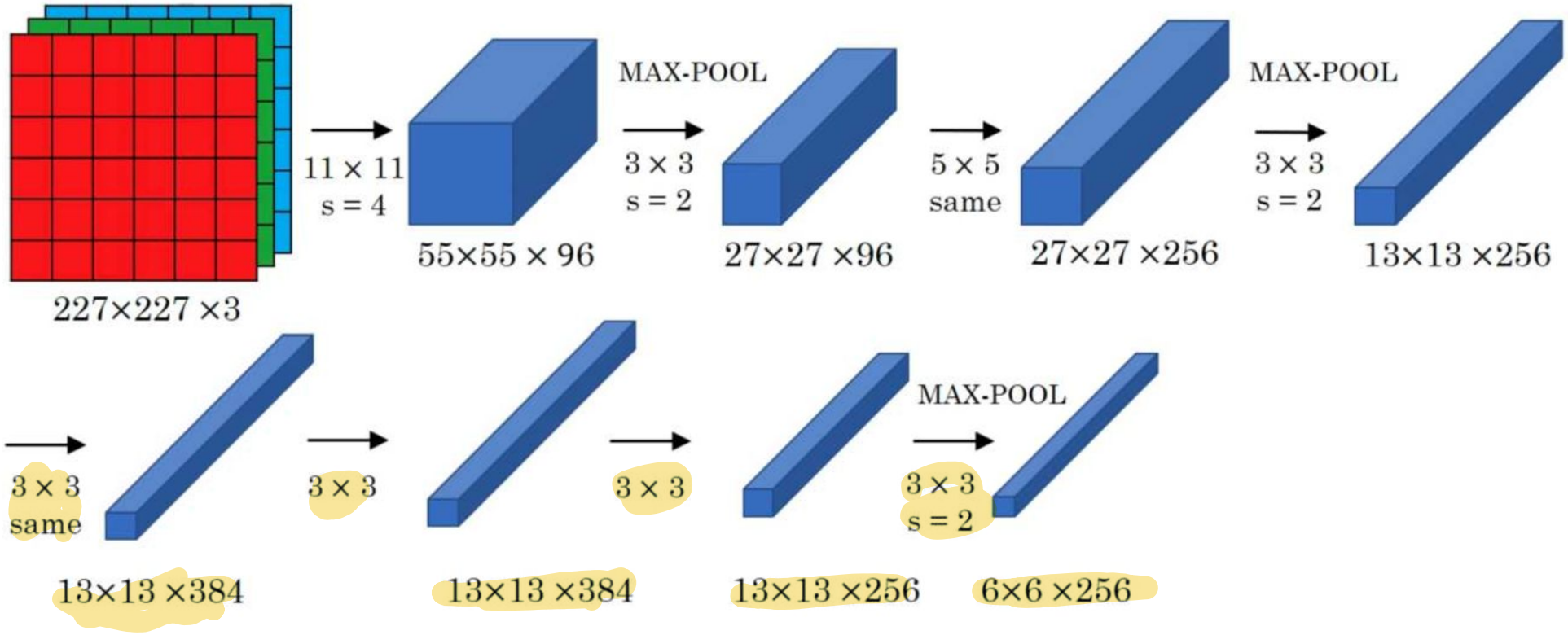
Convolution output is passed through a ReLU activation function and then max pooled with a window size of 3x3 and a stride of 2 pixels.

AlexNet



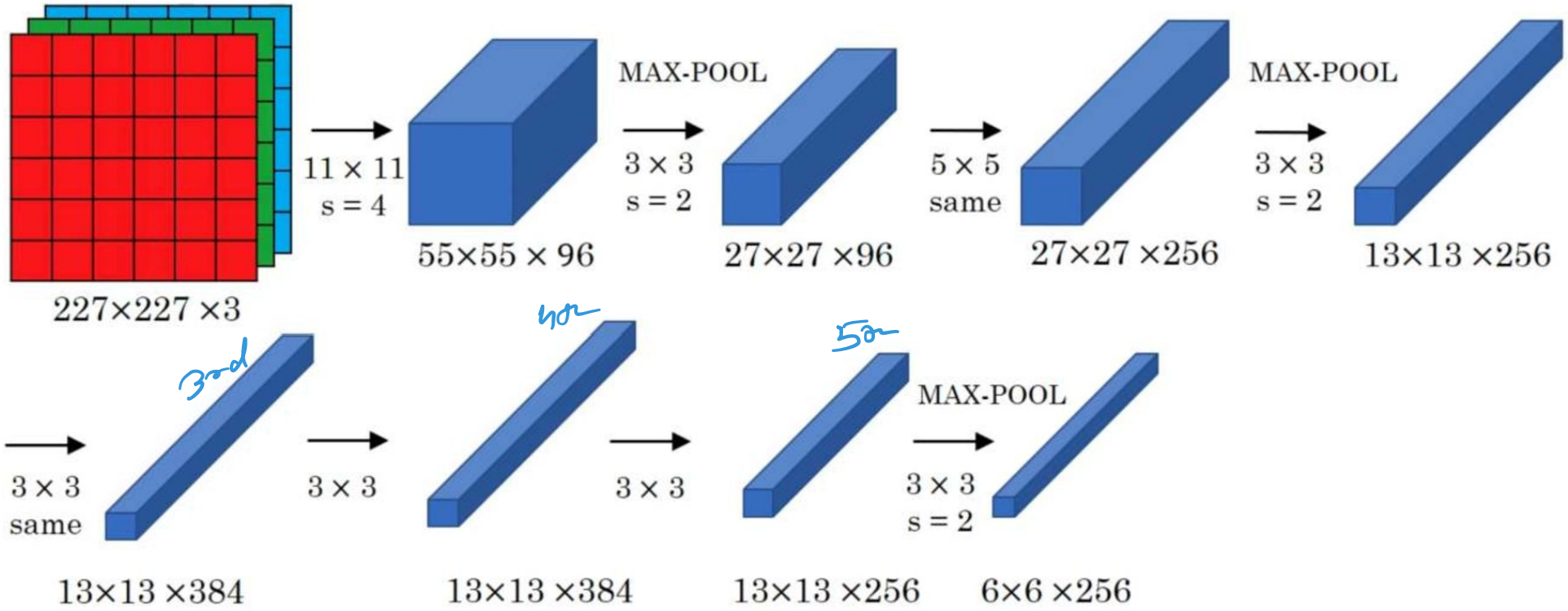
the second convolutional layer has 256 filters.

AlexNet



Next convolution layers applies 384 filters and 256 filters

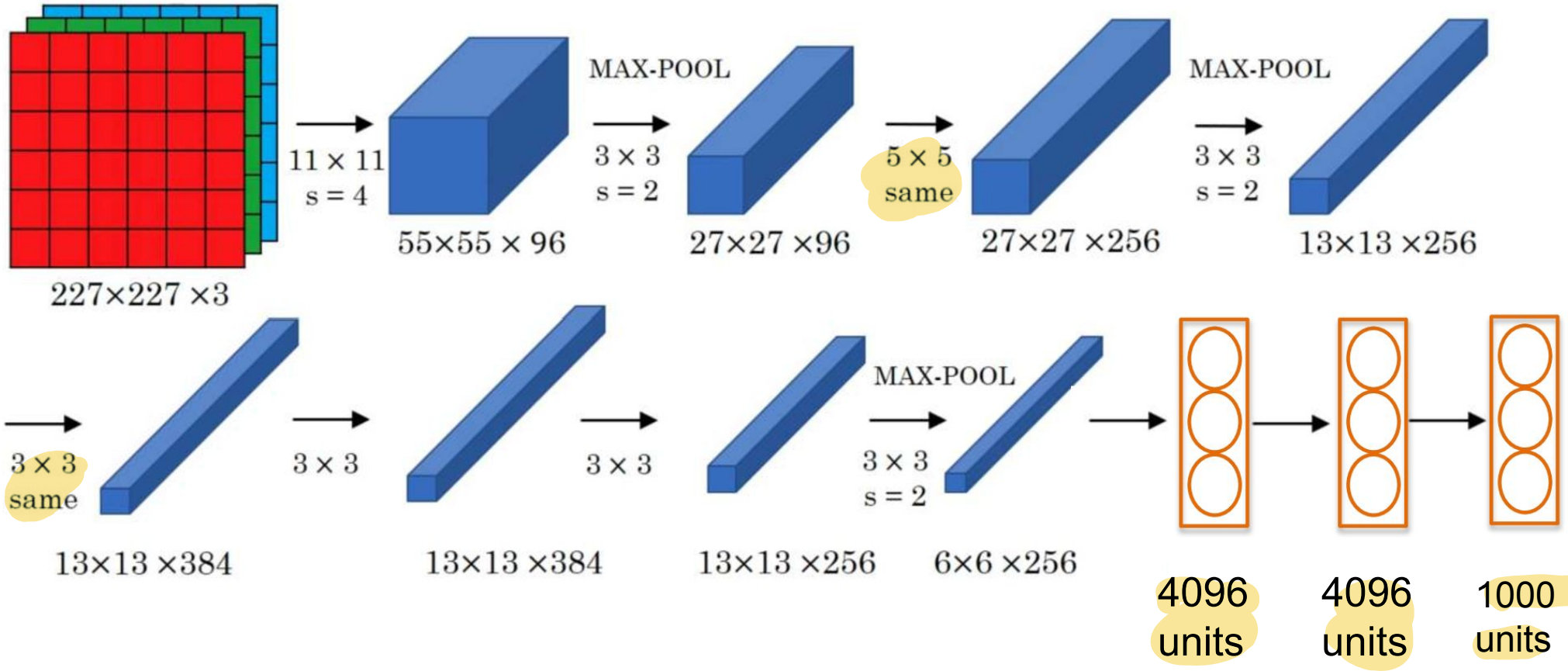
AlexNet



AlexNet uses max pooling after the first, second, and fifth convolutional layers to reduce the spatial dimensions of the feature maps.

1 2

AlexNet



At last we have Fully Connected layer and softmax layer

AlexNet

Let **C**onvolutional **L**ayer denoted by ***CL***. AlexNet has following layers.

- CL 1: applies 96 filters of size 11x11 to input image with a stride of 4 pixels.
- CL 2: applies 256 filters of size 5x5 to the output of the first max pooling layer.
- CL 3: applies 384 filters of size 3x3 to the output of the second max pooling layer.
- CL 4: applies 384 filters of size 3x3 to the output of the third CL.
- CL 5: applies 256 filters of size 3x3 to the output of the fourth CL.

Do you observe a pattern in number of filters

256 3x3
max pooling
Dense Layer.

AlexNet

Let Convolutional layer denoted by CL. AlexNet has following layers.

- CL 1: applies 96 filters of size 11x11 to input image with a stride of 4 pixels.
- CL 2: applies **256** filters of size 5x5 to the output of the first max pooling layer.
- CL 3: applies **384** filters of size 3x3 to the output of the second max pooling layer.
- CL 4: applies **384** filters of size 3x3 to the output of the third CL.
- CL 5: applies **256** filters of size 3x3 to the output of the fourth CL.

Do you observe a pattern in number of filters

AlexNet

Fully connected layers

- FC 1: This layer has 4096 units. The output is passed through a ReLU activation function and then subject to dropout regularization.
- FC 2: This layer is similar to the first fully connected layer, with 4096 units, a ReLU activation function, and dropout regularization.

Output layer:

- This layer has 1000 units (i.e., number of classes in the ImageNet dataset).
- The output is passed through a softmax activation function to produce the final class probabilities.

AlexNet

```
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.models import Model

input_shape = (227, 227, 3) # Input shape of the image

# Define the input layer
inputs = Input(shape=input_shape)
```

AlexNet

```
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.models import Model

input_shape = (227, 227, 3) # Input shape of the image

# Define the input layer
inputs = Input(shape=input_shape)

# First convolutional layer, 96 filters, kernel size of 11x11 and stride of 4x4, followed by ReLU
conv1 = Conv2D(filters=96, kernel_size=(11, 11), strides=(4, 4), activation='relu')(inputs)

# Max pooling layer with pool size of 3x3 and stride of 2x2
pool1 = MaxPooling2D(pool_size=(3, 3), strides=(2, 2))(conv1)

# Batch normalization layer
bn1 = BatchNormalization()(pool1)
```

AlexNet

```
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.models import Model

input_shape = (227, 227, 3) # Input shape of the image

# Define the input layer
inputs = Input(shape=input_shape)

# First convolutional layer, 96 filters, kernel size of 11x11 and stride of 4x4, followed by ReLU
conv1 = Conv2D(filters=96, kernel_size=(11, 11), strides=(4, 4), activation='relu')(inputs)

# Max pooling layer with pool size of 3x3 and stride of 2x2
pool1 = MaxPooling2D(pool_size=(3, 3), strides=(2, 2))(conv1)

# Batch normalization layer
bn1 = BatchNormalization()(pool1)

# Second convolutional layer with 256 filters, kernel size of 5x5 and padding of same, followed by
ReLU activation
conv2 = Conv2D(filters=256, kernel_size=(5, 5), padding='same', activation='relu')(bn1)

# Max pooling layer with pool size of 3x3 and stride of 2x2
pool2 = MaxPooling2D(pool_size=(3, 3), strides=(2, 2))(conv2)

# Batch normalization layer
bn2 = BatchNormalization()(pool2)
```

AlexNet

```
# Third convolutional layer with 384 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv3 = Conv2D(filters=384, kernel_size=(3, 3), padding='same', activation='relu')(bn2)
# Fourth convolutional layer with 384 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv4 = Conv2D(filters=384, kernel_size=(3, 3), padding='same', activation='relu')(conv3)
```

AlexNet

```
# Third convolutional layer with 384 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv3 = Conv2D(filters=384, kernel_size=(3, 3), padding='same', activation='relu')(bn2)
# Fourth convolutional layer with 384 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv4 = Conv2D(filters=384, kernel_size=(3, 3), padding='same', activation='relu')(conv3)

# Fifth convolutional layer, 256 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv5 = Conv2D(filters=256, kernel_size=(3, 3), padding='same', activation='relu')(conv4)
# Max pooling layer with pool size of 3x3 and stride of 2x2
pool5 = MaxPooling2D(pool_size=(3, 3), strides=(2, 2))(conv5)
# Batch normalization layer
bn3 = BatchNormalization()(pool5)
```

AlexNet

```
# Third convolutional layer with 384 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv3 = Conv2D(filters=384, kernel_size=(3, 3), padding='same', activation='relu')(bn2)
# Fourth convolutional layer with 384 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv4 = Conv2D(filters=384, kernel_size=(3, 3), padding='same', activation='relu')(conv3)

# Fifth convolutional layer, 256 filters, kernel size of 3x3 and padding of same, followed by ReLU
conv5 = Conv2D(filters=256, kernel_size=(3, 3), padding='same', activation='relu')(conv4)
# Max pooling layer with pool size of 3x3 and stride of 2x2
pool5 = MaxPooling2D(pool_size=(3, 3), strides=(2, 2))(conv5)
# Batch normalization layer
bn3 = BatchNormalization()(pool5)

# Flatten layer
flatten = Flatten()(bn3)

# First fully connected layer with 4096 units, followed by ReLU activation and dropout
fc1 = Dense(units=4096, activation='relu')(flatten)
dropout1 = Dropout(0.5)(fc1)
# Second fully connected layer with 4096 units, followed by ReLU activation and dropout
fc2 = Dense(units=4096, activation='relu')(dropout1)
dropout2 = Dropout(0.5)(fc2)

# Output layer with 1000 units and softmax activation
outputs = Dense(units=1000, activation='softmax')(dropout2)
```

LeNet and AlexNet

- AlexNet is deeper and more complex than LeNet, with more layers and more parameters.

LeNet and AlexNet

- AlexNet is deeper and more complex than LeNet, with more layers and more parameters.
- AlexNet has larger filters in initial layers (e.g., 11x11 in the first layer), while LeNet uses smaller filters (e.g., 5x5).

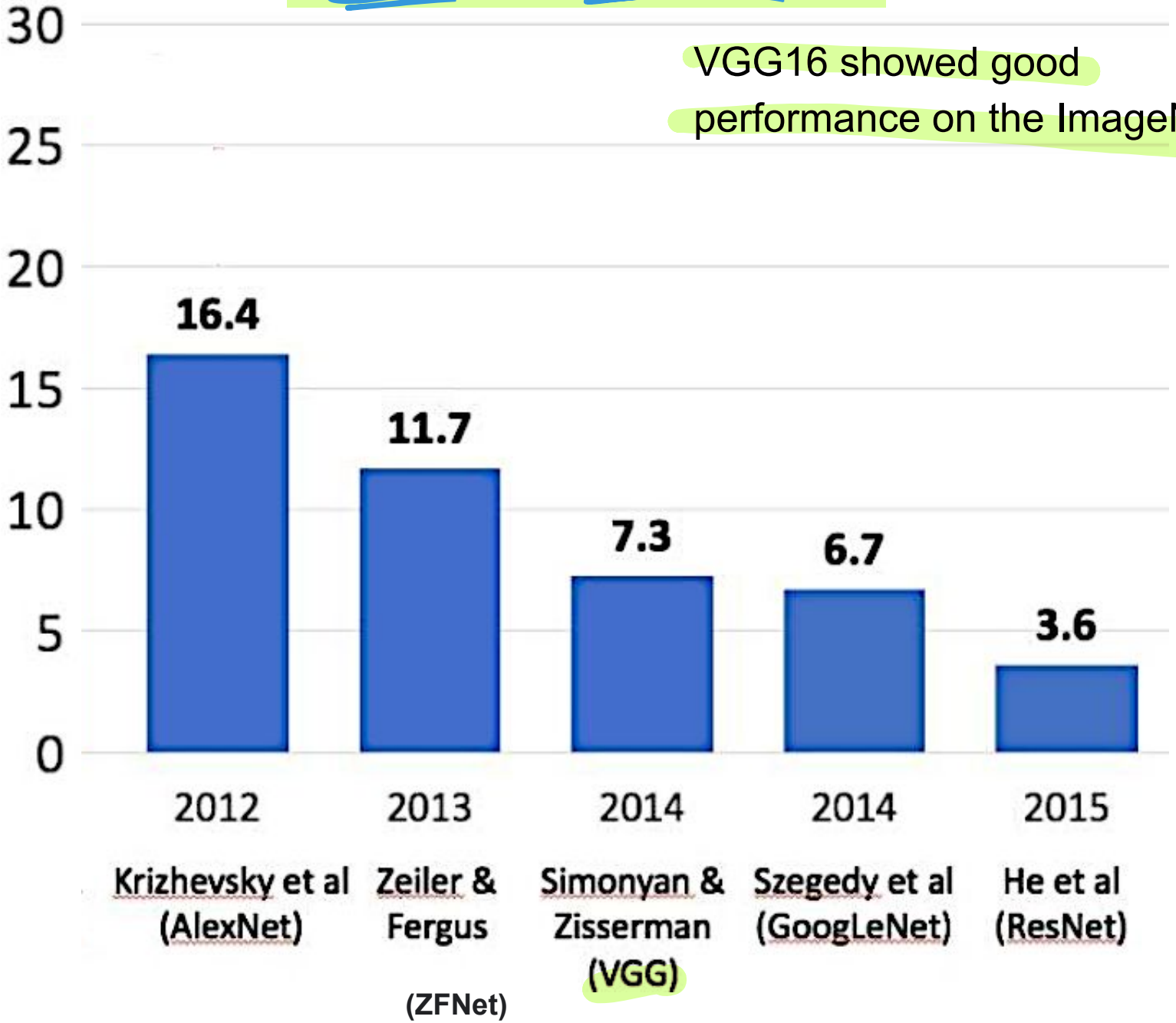
LeNet and AlexNet

- AlexNet is deeper and more complex than LeNet, with more layers and more parameters.
- AlexNet has larger filters in initial layers (e.g., 11x11 in the first layer), while LeNet uses smaller filters (e.g., 5x5).
- AlexNet achieved state-of-the-art performance on the ImageNet dataset, while LeNet was designed and tested on a smaller handwritten digit recognition task.

VGG Network

VGG Network

VGG16 showed good performance on the ImageNet.



VGG Networks

- VGG stands for Visual Geometry Group (University of Oxford) that developed a family of CNN architectures for image classification.

VGG Networks

- VGG stands for Visual Geometry Group (University of Oxford) that developed a family of CNN architectures for image classification.
- The original VGG model, VGG16, has 16 layers including 13 convolutional layers and 3 fully connected layers.

3 fc layers are there in every VGGs.

VGG Networks

- VGG stands for Visual Geometry Group (University of Oxford) that developed a family of CNN architectures for image classification.
- The original VGG model, VGG16, has 16 layers including 13 convolutional layers and 3 fully connected layers.
- VGG19 is with 19 layers including 16 convolutional layers and 3 fully connected layers.

VGG Networks

- VGG stands for Visual Geometry Group (University of Oxford) that developed a family of CNN architectures for image classification.
- The original VGG model, VGG16, has 16 layers including 13 convolutional layers and 3 fully connected layers.
- VGG19 is with 19 layers including 16 convolutional layers and 3 fully connected layers.
- VGG family also includes VGG11 and VGG13, with fewer convolutional layers than VGG16 and VGG19.

VGG Networks

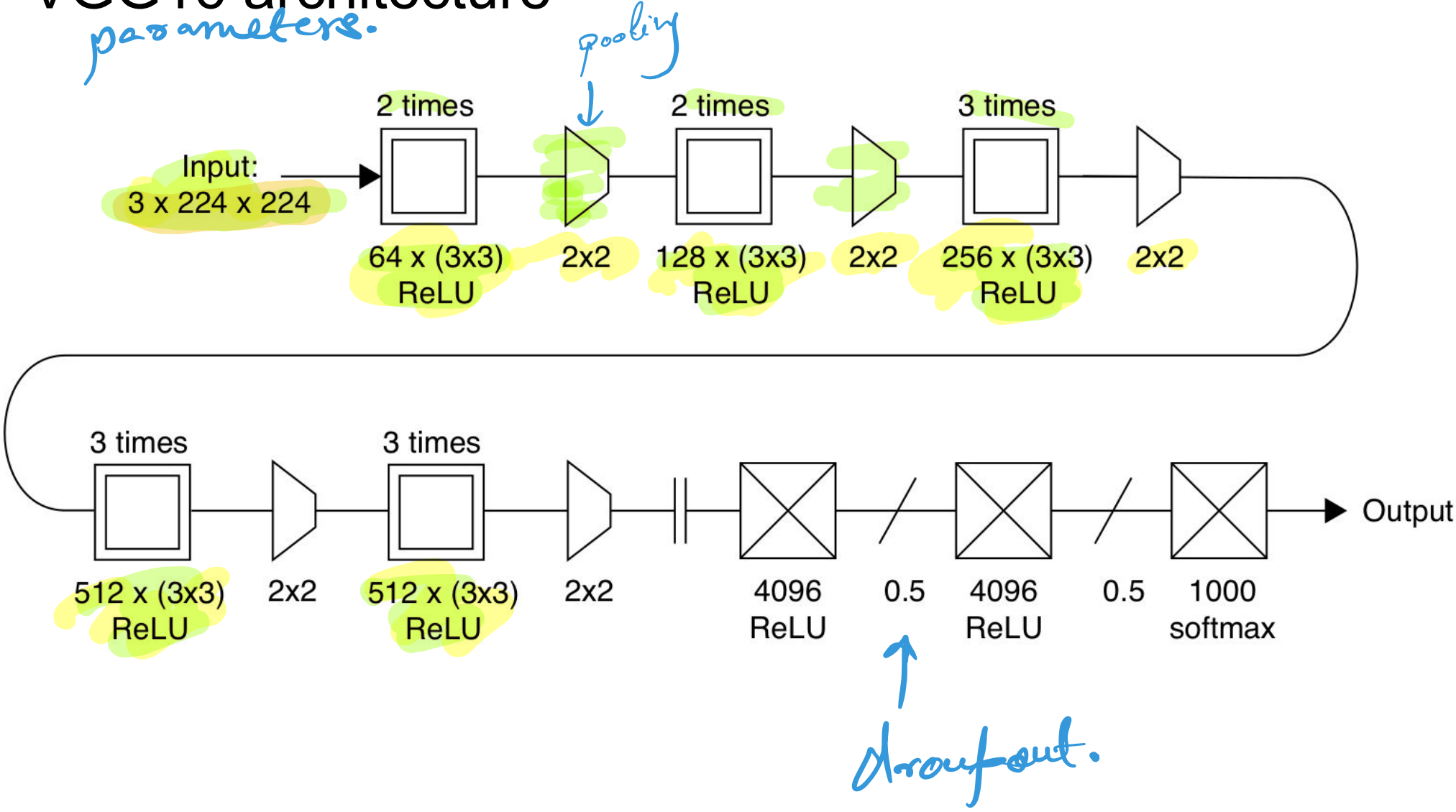
- VGG stands for Visual Geometry Group (University of Oxford) that developed a family of CNN architectures for image classification.
- The original VGG model, VGG16, has 16 layers including 13 convolutional layers and 3 fully connected layers.
- VGG19 is with 19 layers including 16 convolutional layers and 3 fully connected layers.
- VGG family also includes VGG11 and VGG13, with fewer convolutional layers than VGG16 and VGG19.
- VGG family generalize well to many tasks such as classification, object detection, segmentation, style transfer, and transfer learning.

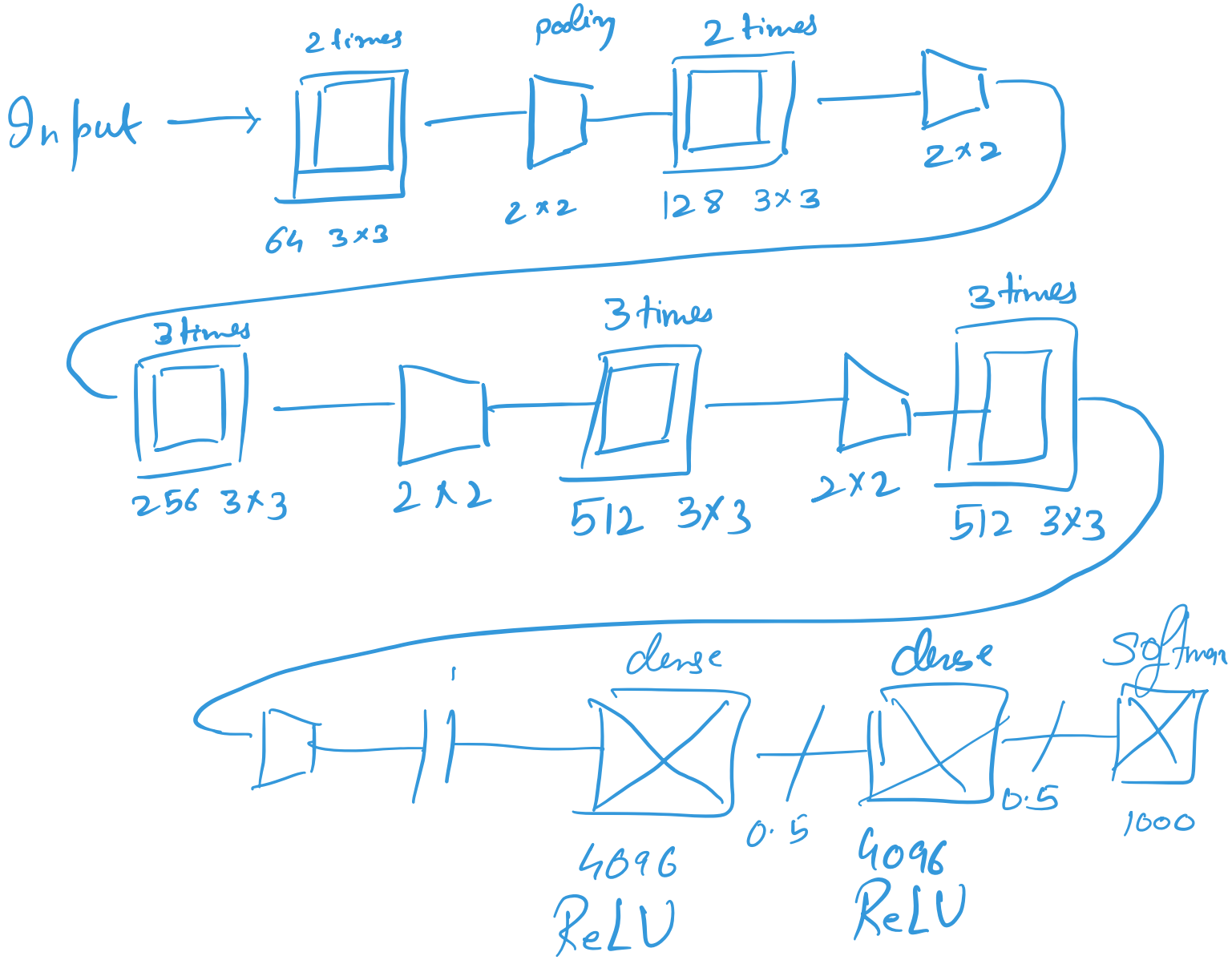
what VGGNet does better than Alexnet

→ Use of 3×3 filters multiple times gave

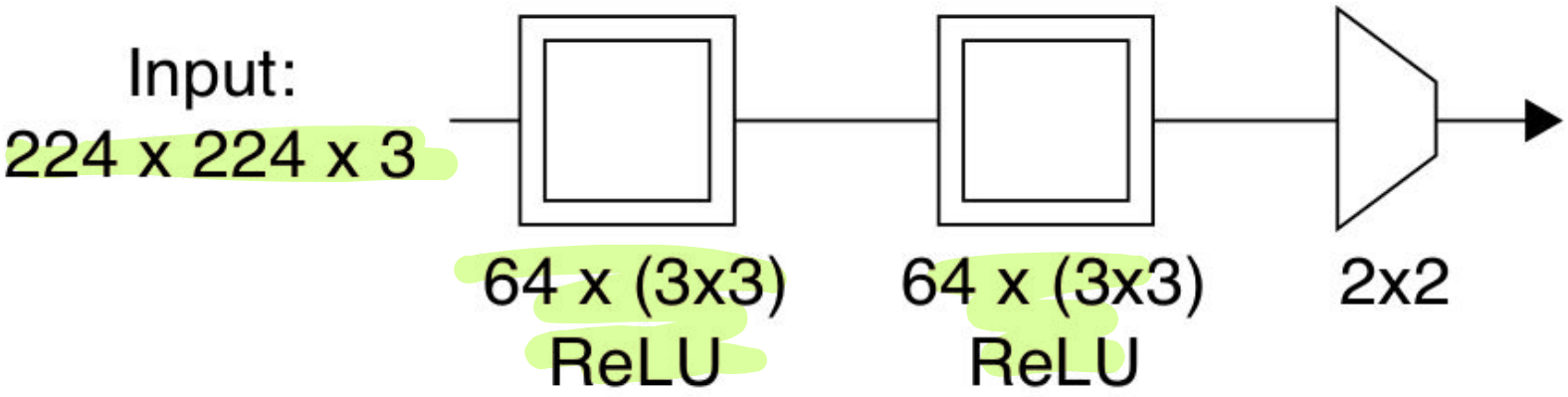
higher receptive field with the use of Less parameters.

VGG16 architecture



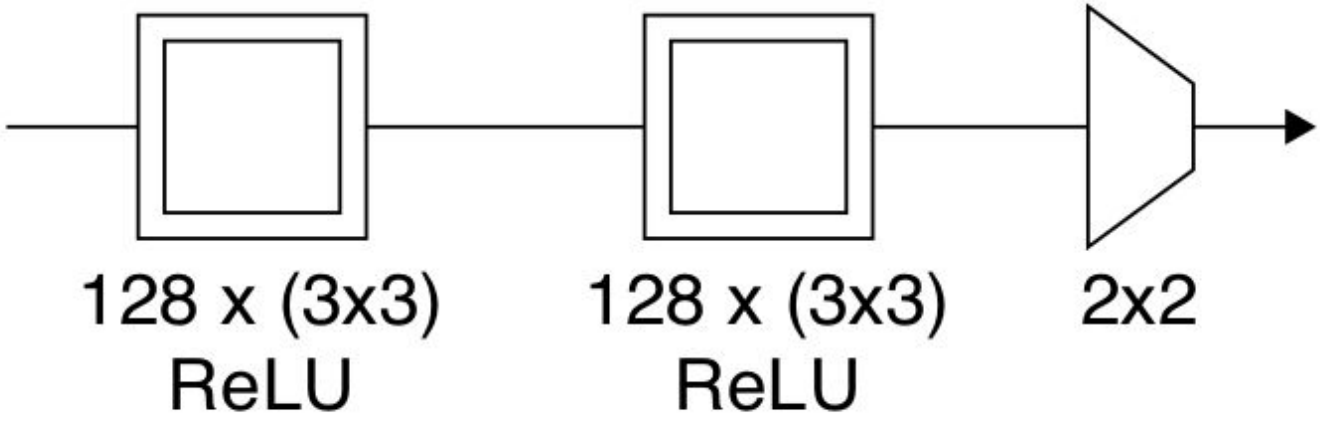

Input
2 times
64 3×3
pooling
2×2
2 times
128 3×3
2×2
3 times
256 3×3
2×2
3 times
512 3×3
2×2
3 times
512 3×3
dense
4096
ReLU
0.5
dense
4096
ReLU
0.5
Softmax
1000

VGG16 architecture



Group 1 of VGG16. We convolve the input tensor with 64 filters each of size 3 by 3. Then we convolve again with 64 new filters. Finally, we use max pooling to reduce the output tensor's height and width by half.

VGG16 architecture

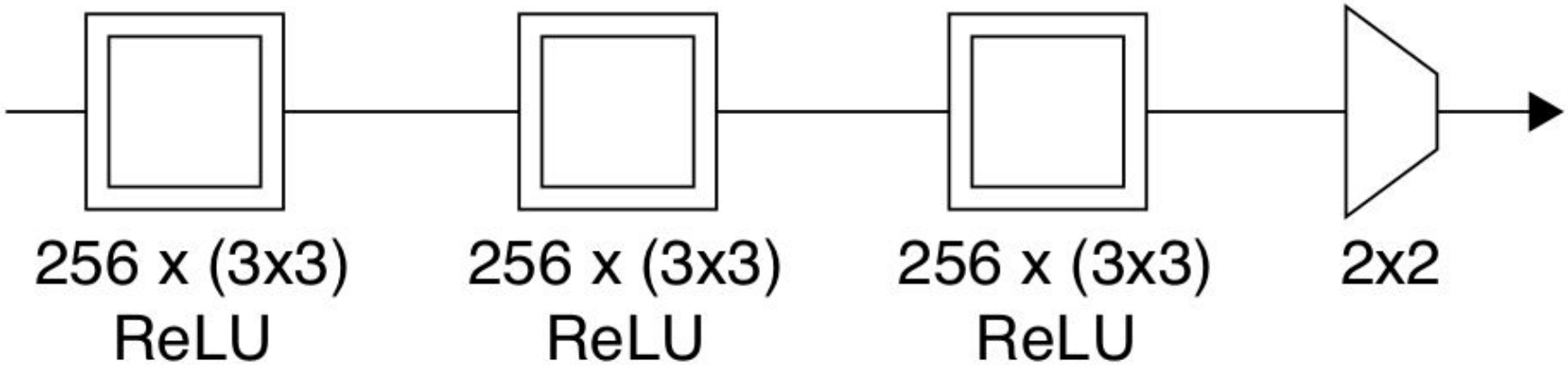


Group 2 of VGG16 is just like the first block except that we use 128 filters in each convolution layer rather than 64.

2 2 3

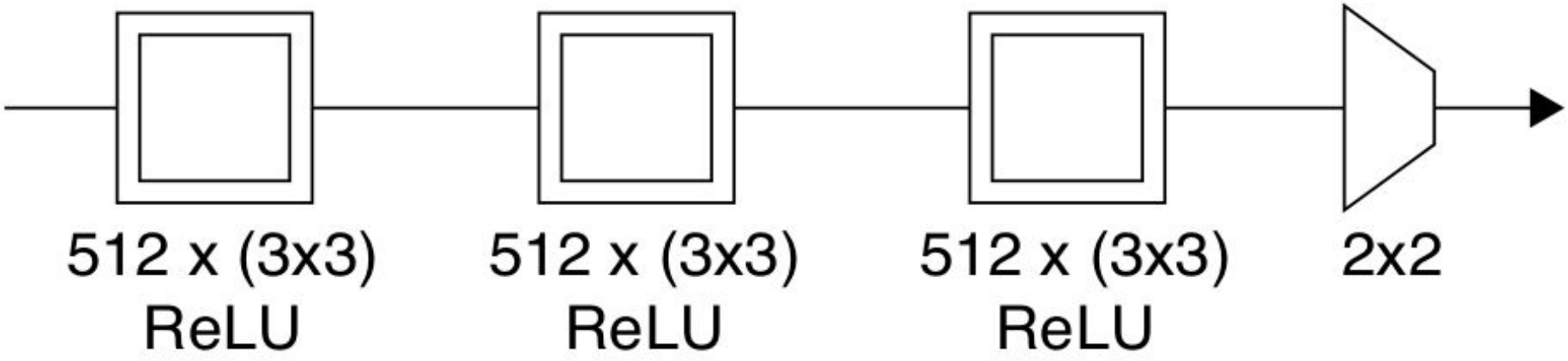
64 128 256

VGG16 architecture



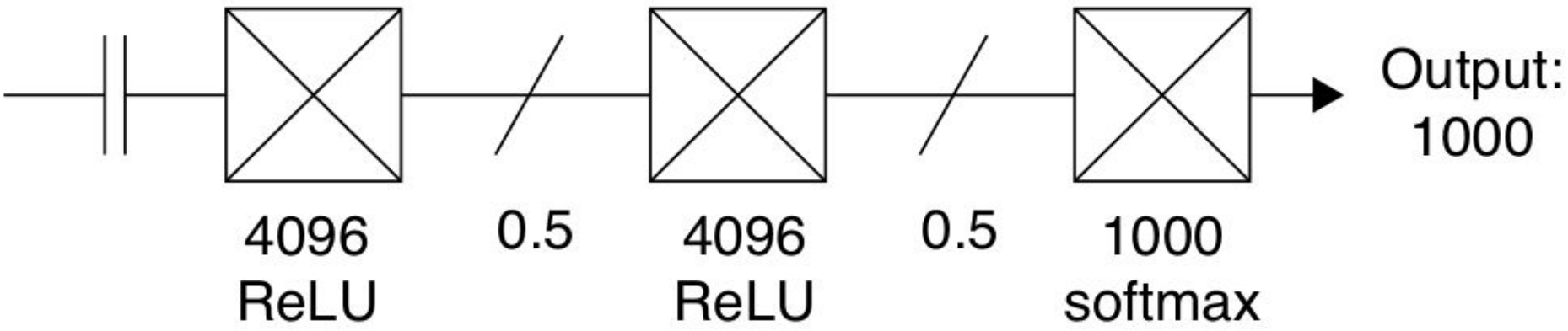
Group 3 of VGG16 doubles the number of filters again to 256 and repeats the convolution step three times rather than two as before.

VGG16 architecture



Groups 4 and 5 of VGG16 are the same. They each have three convolution layers, followed by a two by two max pooling layer.

VGG16 architecture



The final steps of processing in VGG16. We flatten the image, then run it through two dense layers each using ReLU, followed by dropout, then through a dense layer with softmax.

VGG16 (summary)

- VGG16 has 13 convolutional layers and 3 fully connected layers (16 total).

VGG16 (summary)

- VGG16 has 13 convolutional layers and 3 fully connected layers (16 total).
- *Convolutional layers* use 3x3 filters, stride 1, same padding, followed by a ReLU (deeper networks & less parameters)

VGG16 (summary)

- VGG16 has 13 convolutional layers and 3 fully connected layers (16 total).
- *Convolutional layers* use 3x3 filters, stride 1, same padding, followed by a ReLU (deeper networks & less parameters)
- Pooling layers use 2x2 filters with a stride of 2 pixels.

VGG16 (summary)

- VGG16 has 13 convolutional layers and 3 fully connected layers (16 total).
- *Convolutional layers* use 3x3 filters, stride 1, same padding, followed by a ReLU (deeper networks & less parameters)
- Pooling layers use 2x2 filters with a stride of 2 pixels.
- The first two convolutional layers have 64 filters each, while the remaining layers have 128, 256, 512, and 512 filters, respectively.

VGG16 (summary)

- VGG16 has 13 convolutional layers and 3 fully connected layers (16 total).
- *Convolutional layers* use 3x3 filters, stride 1, same padding, followed by a ReLU (deeper networks & less parameters)
- Pooling layers use 2x2 filters with a stride of 2 pixels.
- The first two convolutional layers have 64 filters each, while the remaining layers have 128, 256, 512, and 512 filters, respectively.
- Fully connected layers have 4096 units that use ReLU, and output layer has 1000 units corresponding to number of classes in the ImageNet.

VGG16

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions
```

VGG16

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions

# Load the VGG16 model
model = VGG16(weights='imagenet')
```

VGG16

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions

# Load the VGG16 model
model = VGG16(weights='imagenet')

# Load the image you want to classify
img_path = 'tiger_shark.jpeg'
img = image.load_img(img_path, target_size=(224, 224))
```

VGG16

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions

# Load the VGG16 model
model = VGG16(weights='imagenet')

# Load the image you want to classify
img_path = 'tiger_shark.jpeg'
img = image.load_img(img_path, target_size=(224, 224))

# Convert the image to an array
x = image.img_to_array(img)
x = tf.expand_dims(x, axis=0)
x = preprocess_input(x)
```

VGG16

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions

# Load the VGG16 model
model = VGG16(weights='imagenet')

# Load the image you want to classify
img_path = 'tiger_shark.jpeg'
img = image.load_img(img_path, target_size=(224, 224))

# Convert the image to an array
x = image.img_to_array(img)
x = tf.expand_dims(x, axis=0)
x = preprocess_input(x)

# Use the model to predict the class of the image
preds = model.predict(x)

# Print the top 5 predictions
print('Predicted:', decode_predictions(preds, top=5)[0])
```

VGG16 Implementation

```
# Import necessary libraries
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense

def vgg16(input_shape=(224, 224, 3), num_classes=1000):
    input_tensor = Input(shape=input_shape)
```

VGG16 Implementation

```
# Import necessary libraries
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense

def vgg16(input_shape=(224, 224, 3), num_classes=1000):
    input_tensor = Input(shape=input_shape)

    # Block 1
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(input_tensor)
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)
```

VGG16 Implementation

```
# Import necessary libraries
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense

def vgg16(input_shape=(224, 224, 3), num_classes=1000):
    input_tensor = Input(shape=input_shape)

    # Block 1
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(input_tensor)
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

    # Block 2
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

    # Block 3
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)
```

VGG16 Implementation

```
    # Block 4
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

    # Block 5
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)
```

VGG16 Implementation

```
    # Block 4
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

    # Block 5
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

    # Flatten and dense layers
    x = Flatten(name='flatten')(x)
    x = Dense(4096, activation='relu', name='fc1')(x)
    x = Dense(4096, activation='relu', name='fc2')(x)
    output_tensor = Dense(num_classes, activation='softmax', name='predictions')(x)
```

VGG16 Implementation

```
    # Block 4
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

    # Block 5
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

    # Flatten and dense layers
    x = Flatten(name='flatten')(x)
    x = Dense(4096, activation='relu', name='fc1')(x)
    x = Dense(4096, activation='relu', name='fc2')(x)
    output_tensor = Dense(num_classes, activation='softmax', name='predictions')(x)

    # Create model
    model = Model(inputs=input_tensor, outputs=output_tensor, name='vgg16')

    return model
```