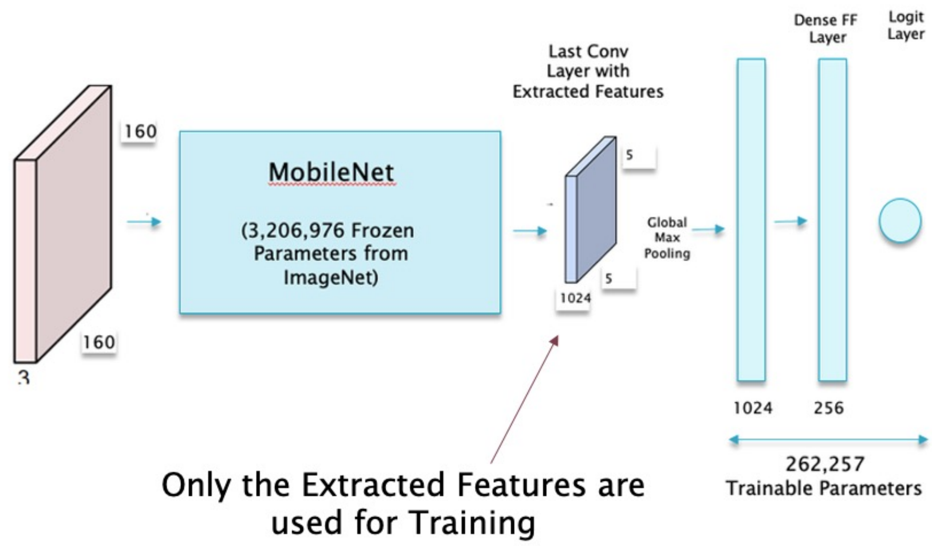
Convolutional Neural Networks: Part 3

Lecture 12
Subir Varma

Transfer Learning

- Enables the use of Pre-Trained Models
  - Pre-Training is usually done with very large datasets, using Large Models
- You can take the Pre-Trained Model and Fine Tune it for your own Dataset



Trends in ConvNet Design

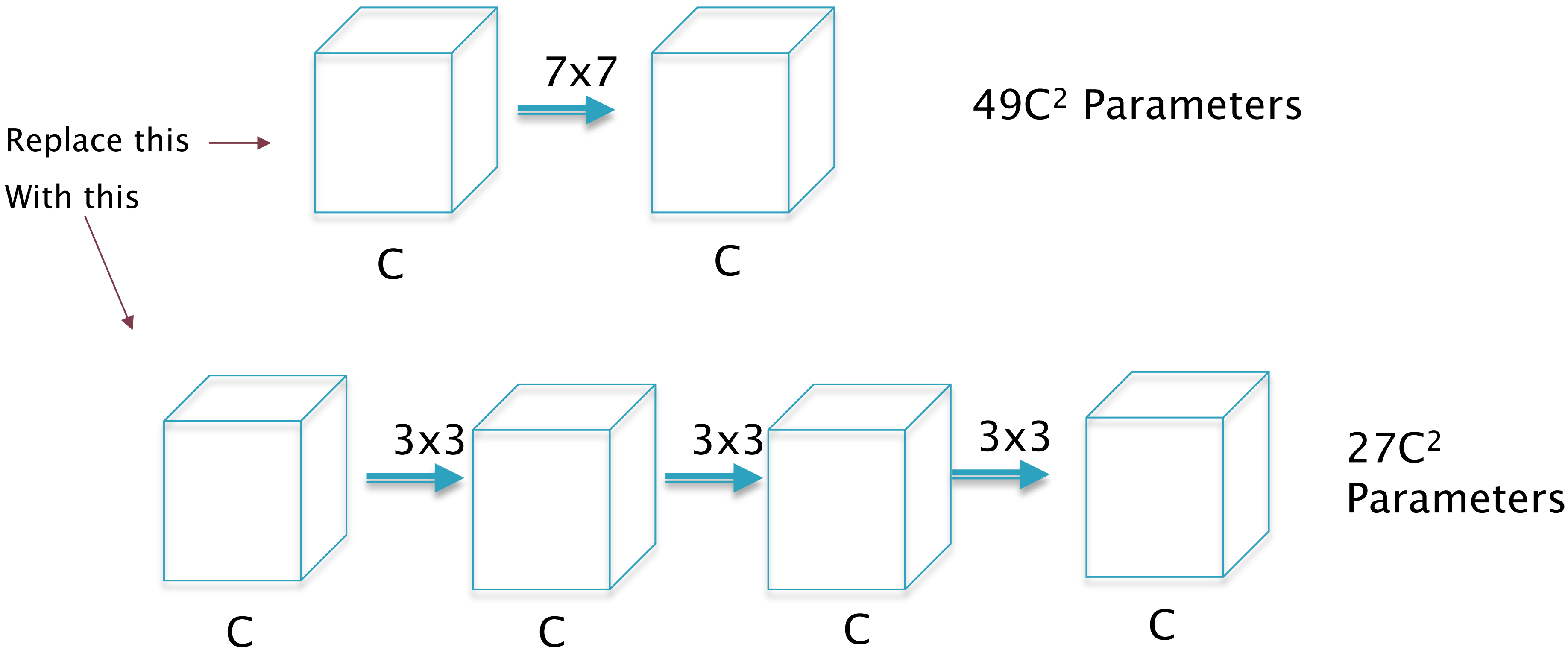
Trends in ConvNet Design

- Small Filters
- Bottlenecking
- Split-Transform-Merge (Grouped Convolutions)
- Depthwise Separable Convolutions
- Residual Connections

- Average Pooling
- Dispensing with Pooling Layer
- Dispensing with Fully Connected Layers

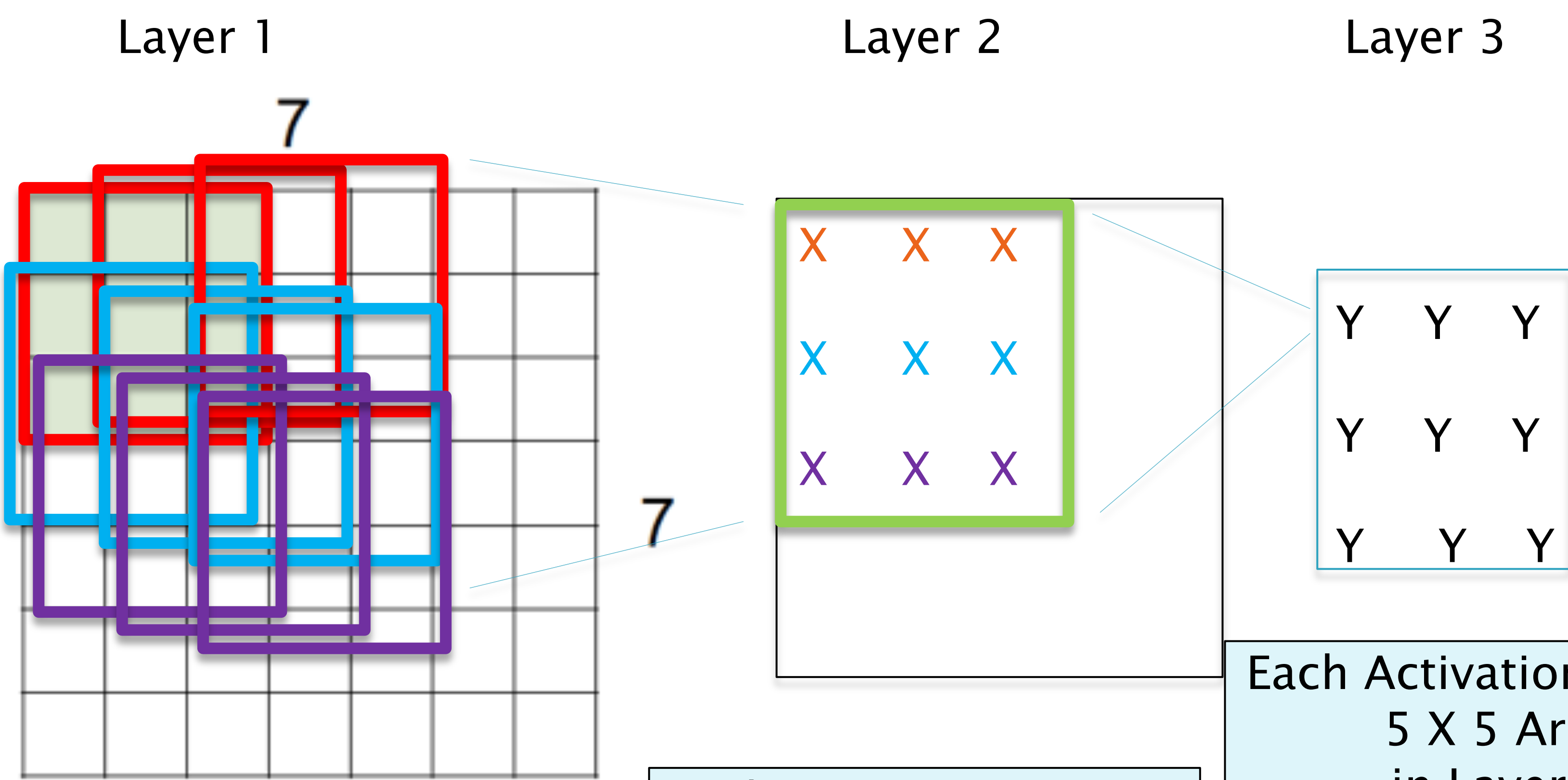
Small Filters

Better to use 3 Layers of 3x3 Filters rather than a single 7x7 Filter



Benefits:
Less number of Parameters
More Non-Linearity

Small Filters: Telescoping Effect with Multiple Layers

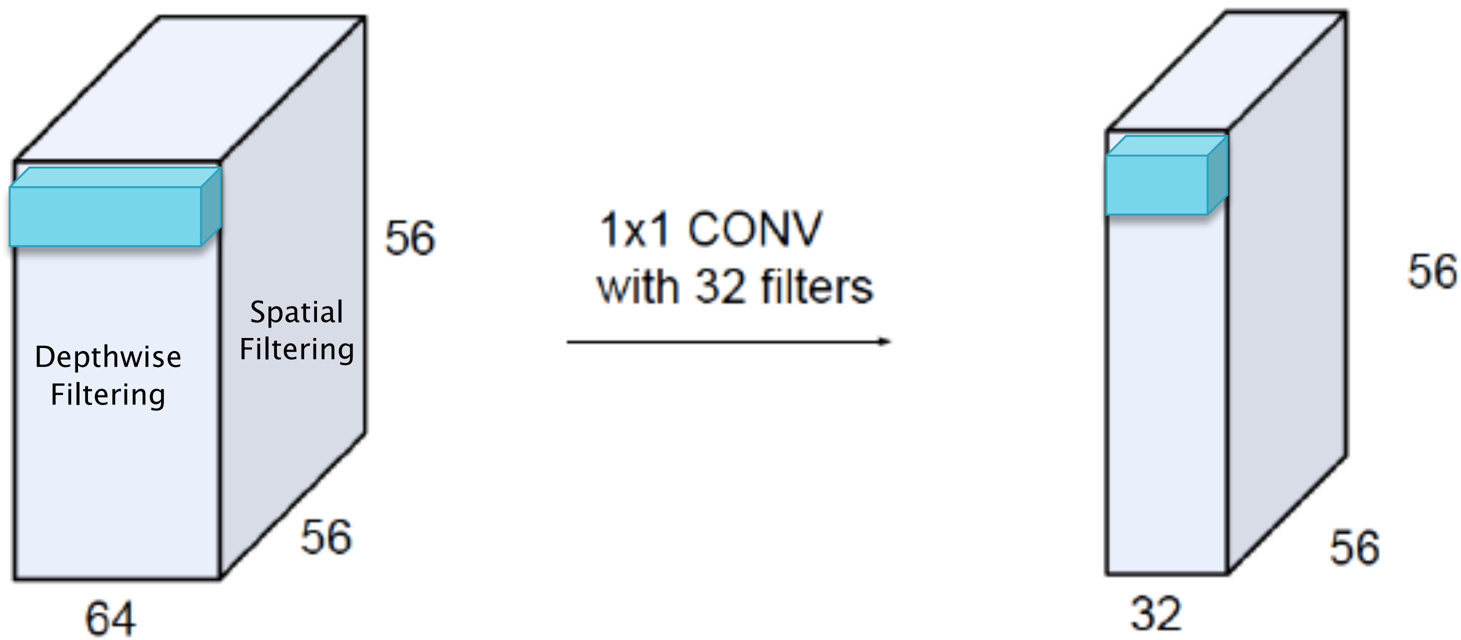


Each Activation Covers
3 X 3 Area
in Layer 1

Each Activation Covers
5 X 5 Area
in Layer 1

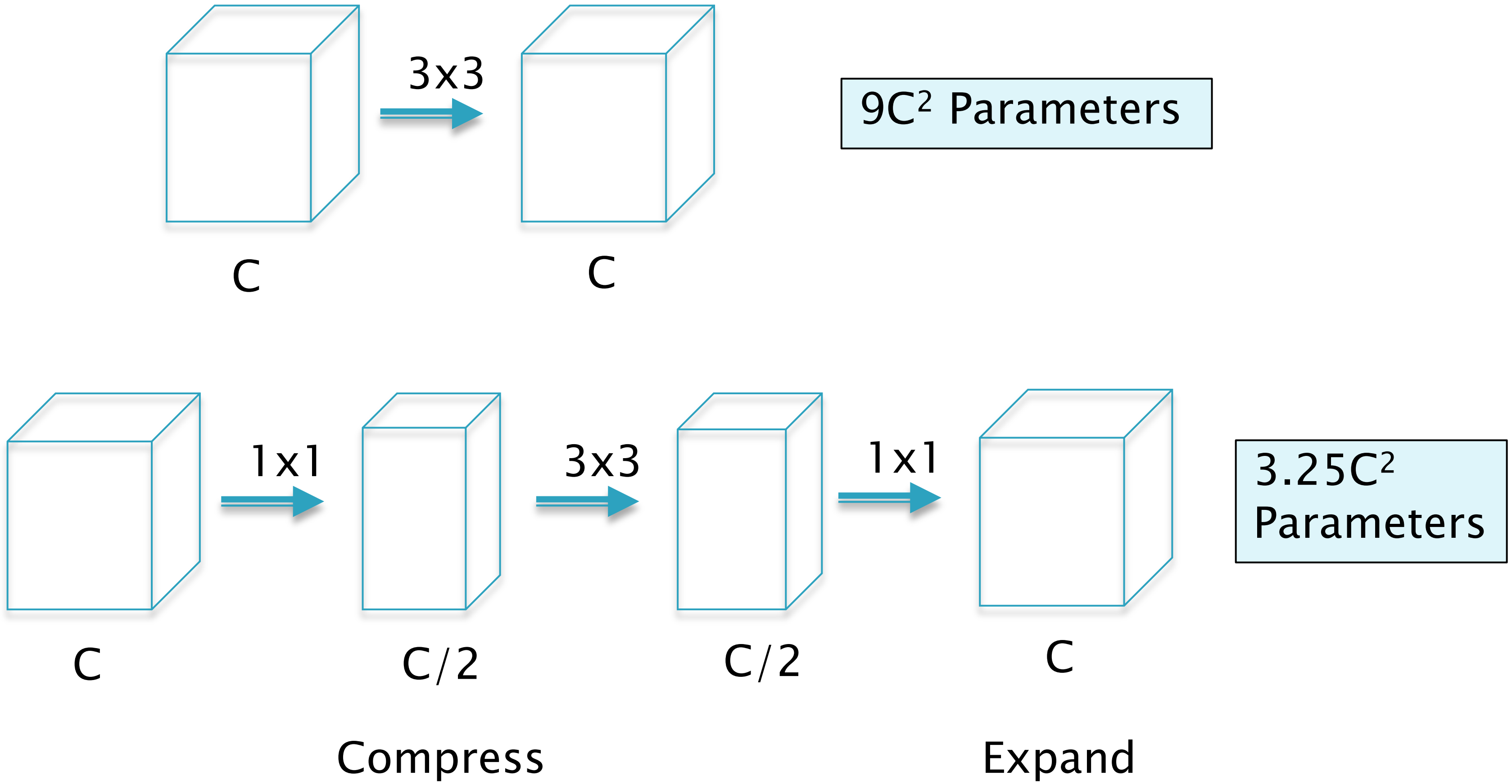
1X1 Filters: Bottlenecking

- Filtering across multiple Activation Maps
- No Spatial Filtering (in a single Map)

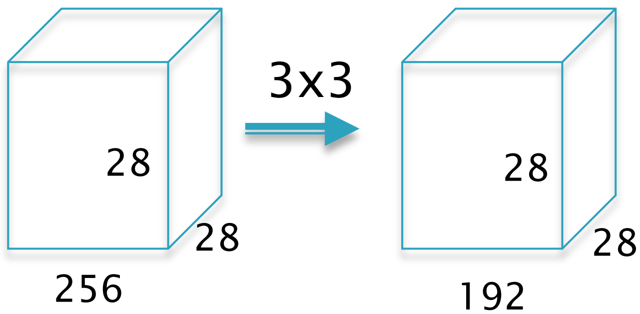


Uses:
- Compress the number of Activation Maps
- Reduce number of parameters/computations

Using 1x1 Filters: Parameters Reduction

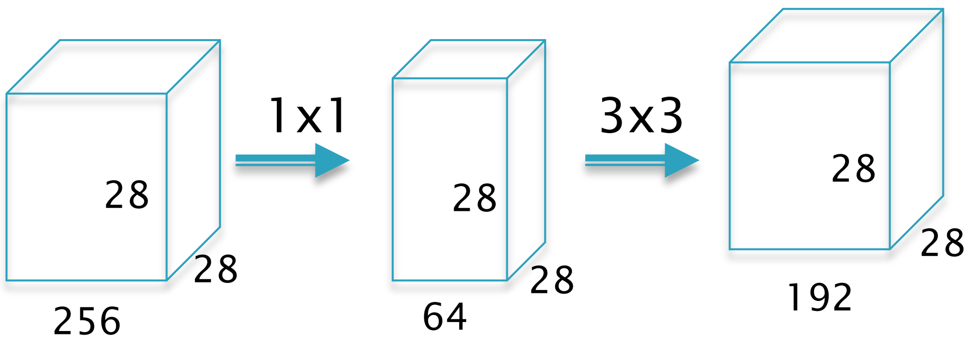


Using 1x1 Filters: Computations Reduction



(3x3)x256x28x28x192 computations

346,816,512

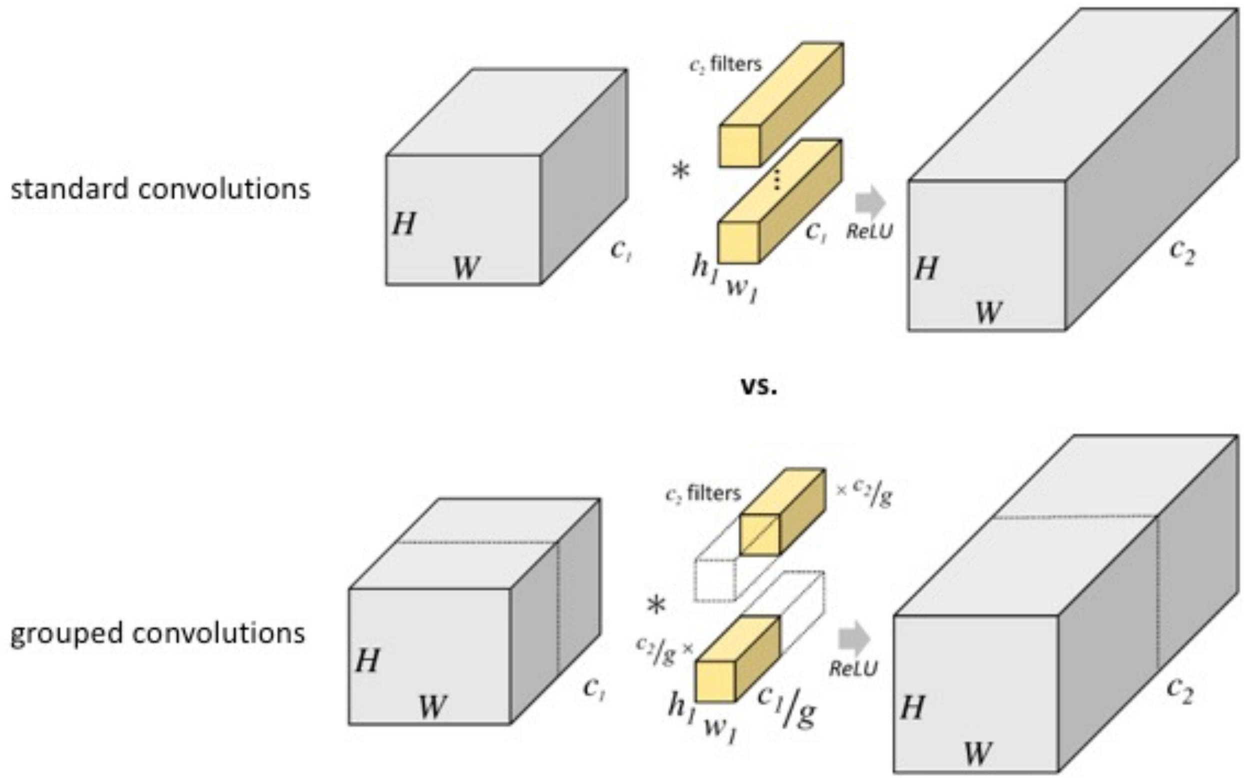


Compress

(1x1)x256x28x28x64 + (3x3)x64x28x28x192 computations

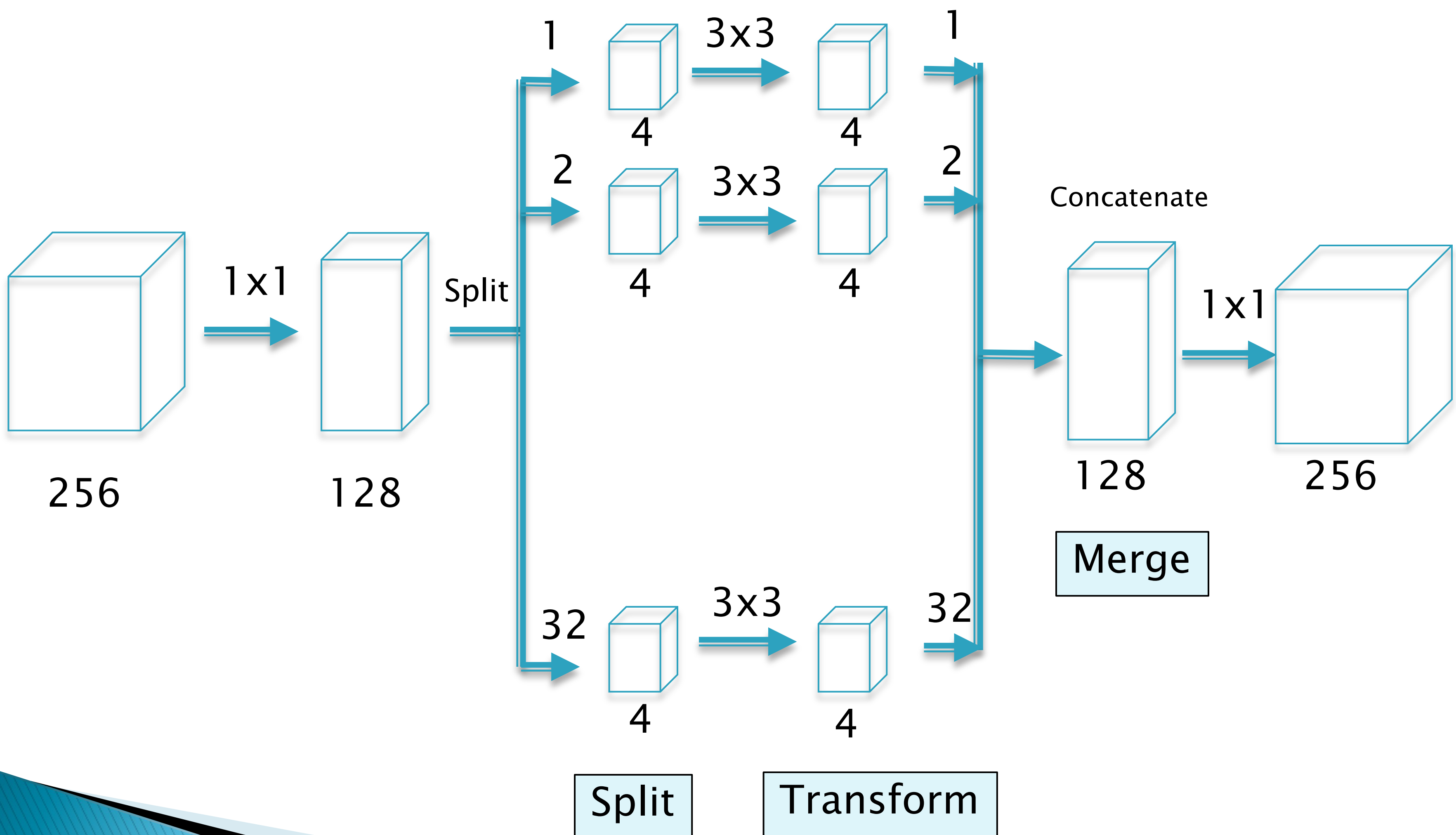
99,549,184

Grouped Convolutions

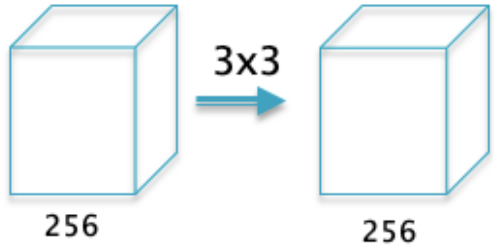


Split up the input channels into 2 groups
and in parallel process each group with smaller filters

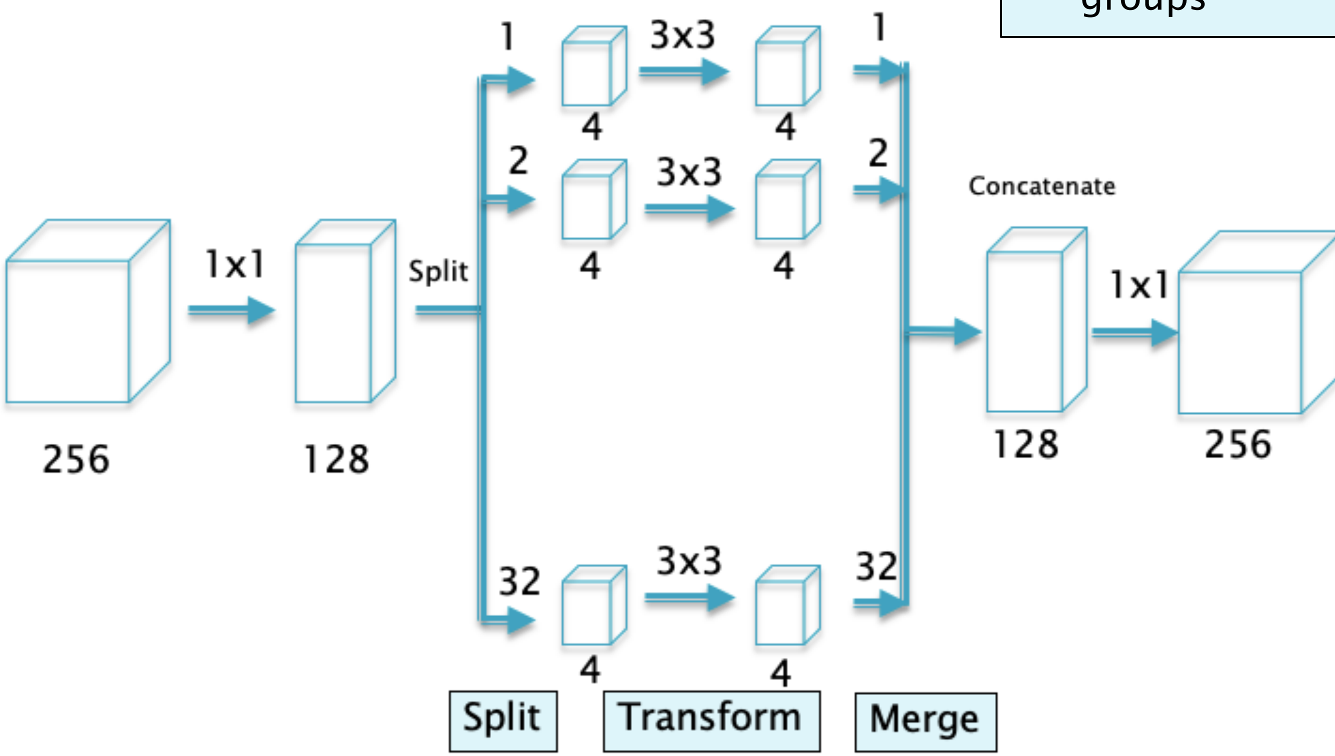
Application of Grouped Convolutions: Split, Transform, Merge



Replace

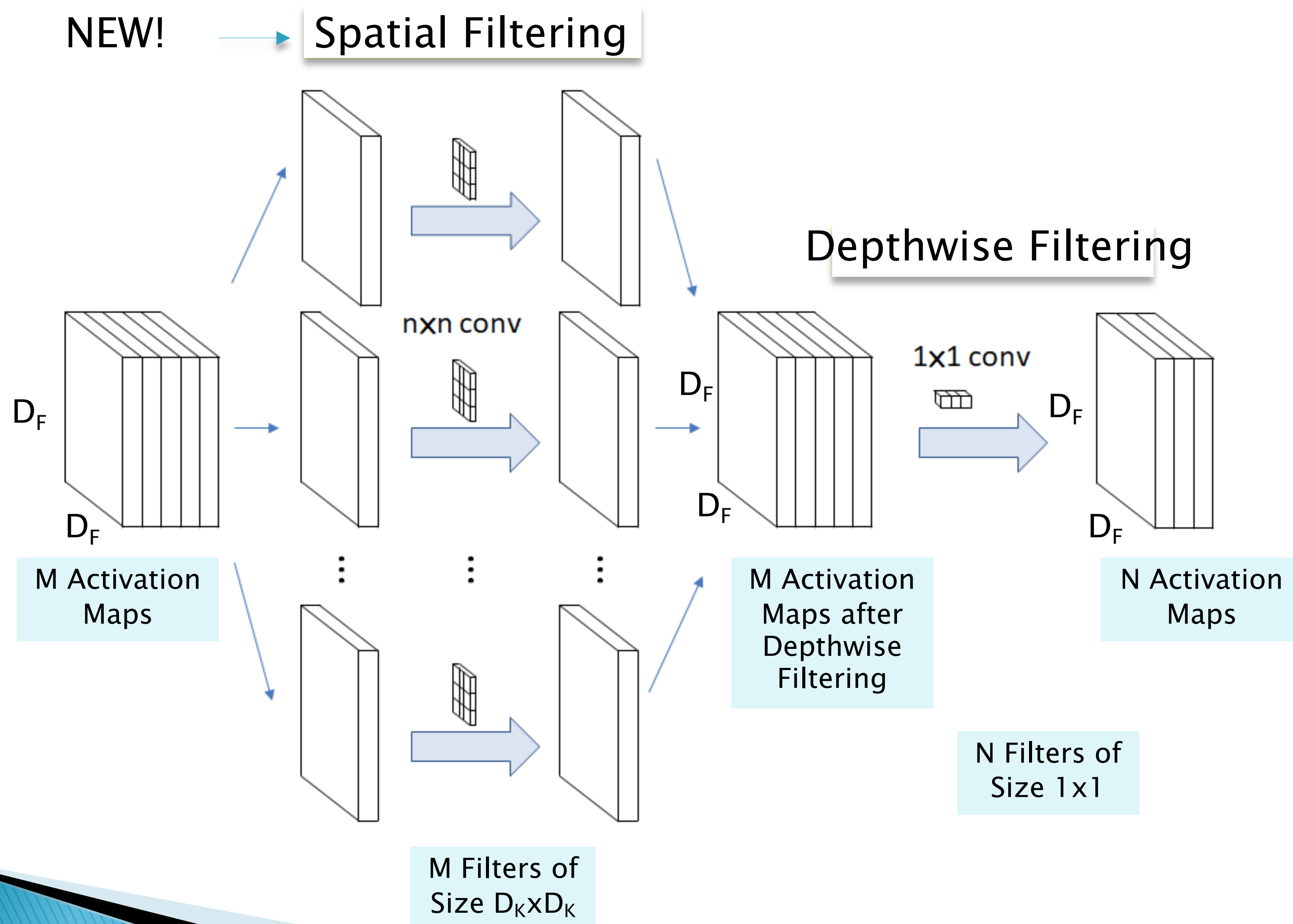


By

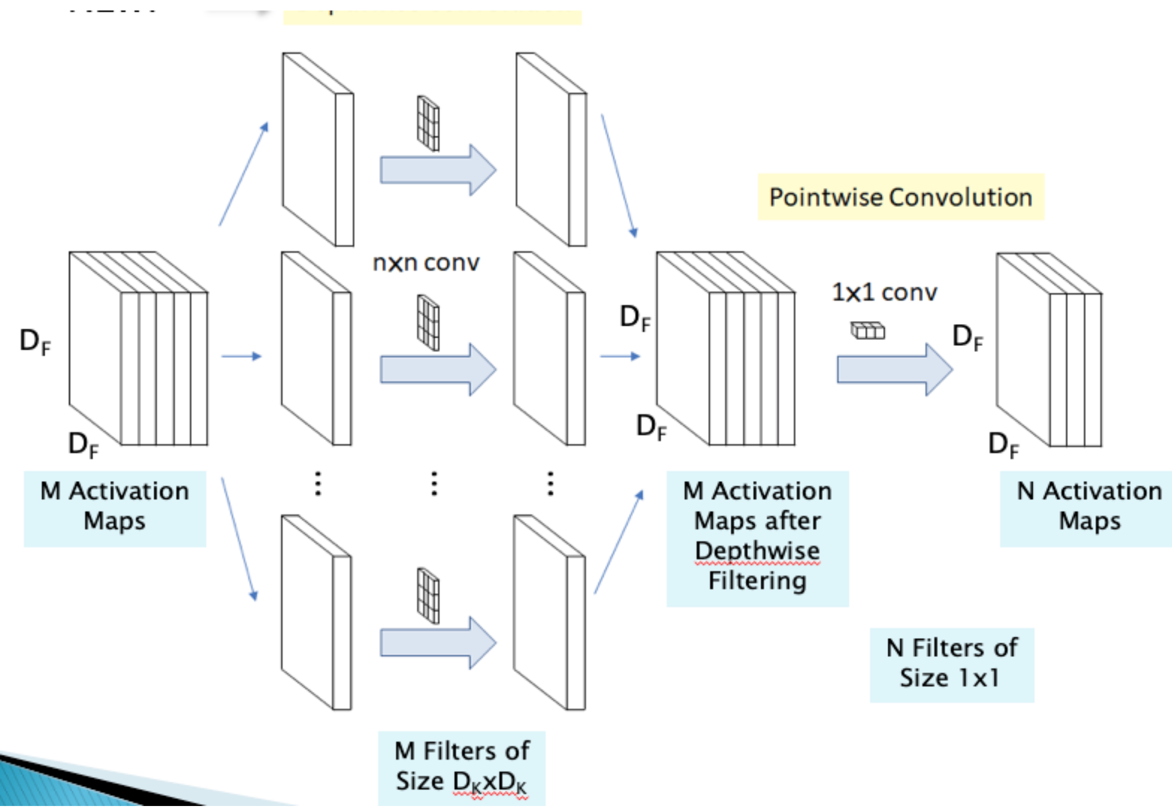


Benefits:
- Better performance: Activation Maps in a Group Tend to learn similar features
- Decrease in number of computations in proportion to number of groups

Depthwise Separable Convolutions



Depthwise Separable Convolutions



Standard Convolutions have a computational cost of:
$D_K \,.\, D_K \,.\, M \,.\, N \,.\, D_F \,.\, D_F$

Depthwise Separable Convs have a computational cost of:
$D_K \,.\, D_K \,.\, M \,.\, D_F \,.\, D_F + M \,.\, N \,.\, D_F \,.\, D_F$

Savings of: $\frac{1}{N} + \frac{1}{D_K^2}$

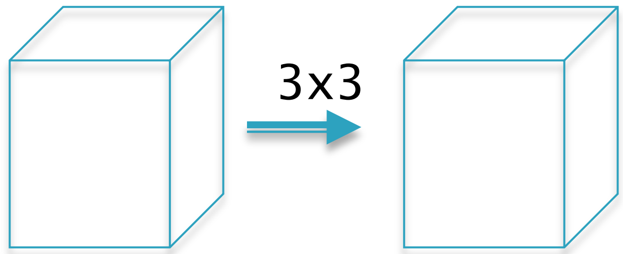
D_K: Size of Filter
M: Number of Input Activation Maps
N: Number of Output Activation Maps
D_F: Size of Output Activation Map

Reference: https://arxiv.org/pdf/1704.04861.pdf

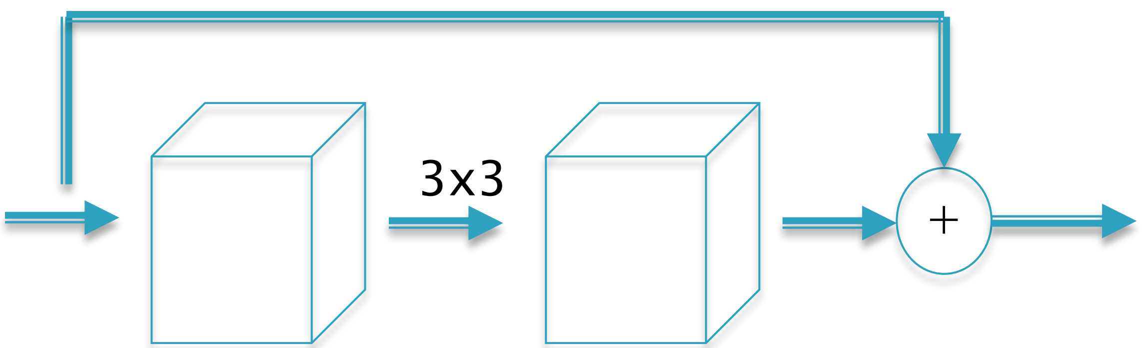
Depthwise Separable Convolutions in Keras

```
# Create the model
model = Sequential()
model.add(SeparableConv2D(32, kernel_size=(3, 3), activation='relu', input_s
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(SeparableConv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))
```

Residual Connections



(a)



(b)

ConvNet Architectures

Pre-Trained Models in Keras

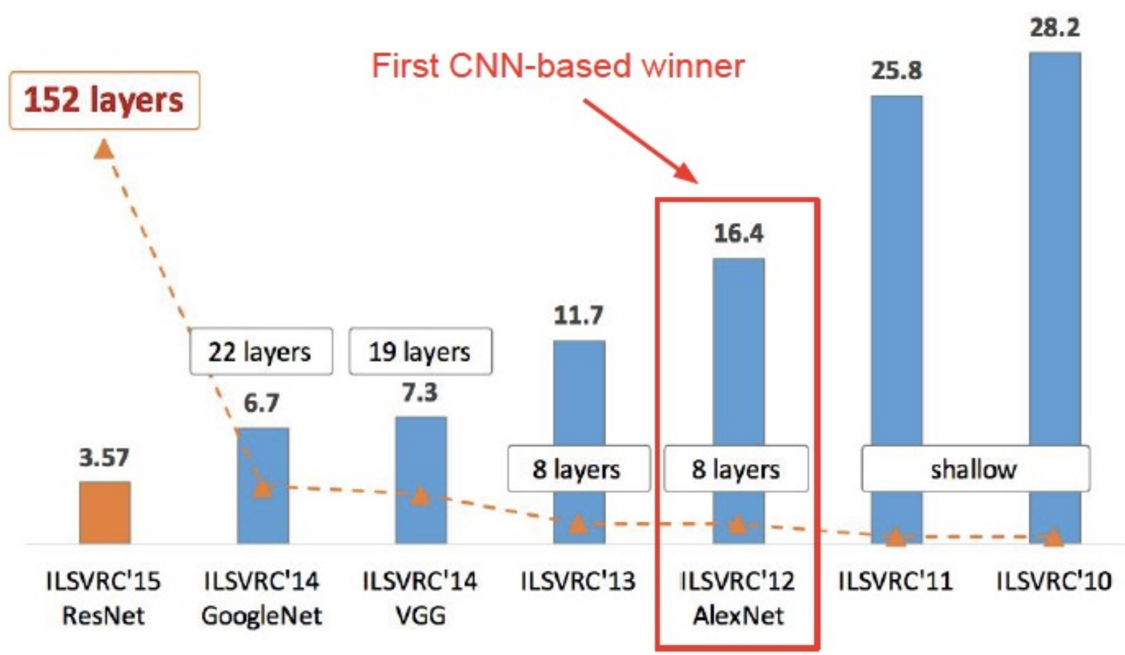
Pre-Trained on ImageNet

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

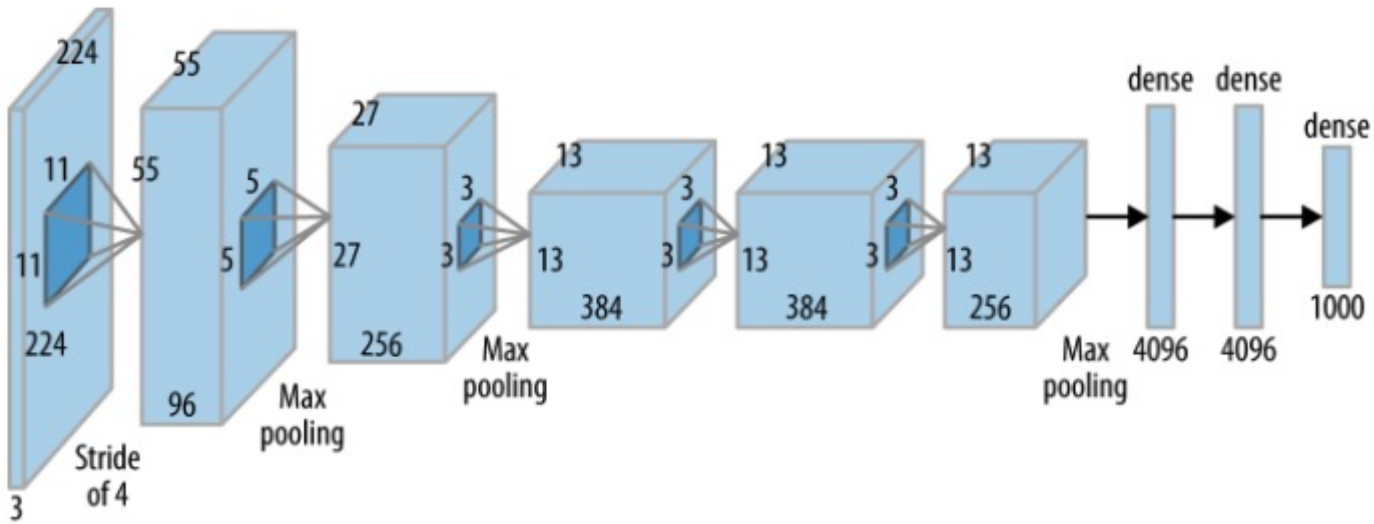
Using Transfer Learning it is possible to
Use these models for non-ImageNet problems

ILSVRC Challenge

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

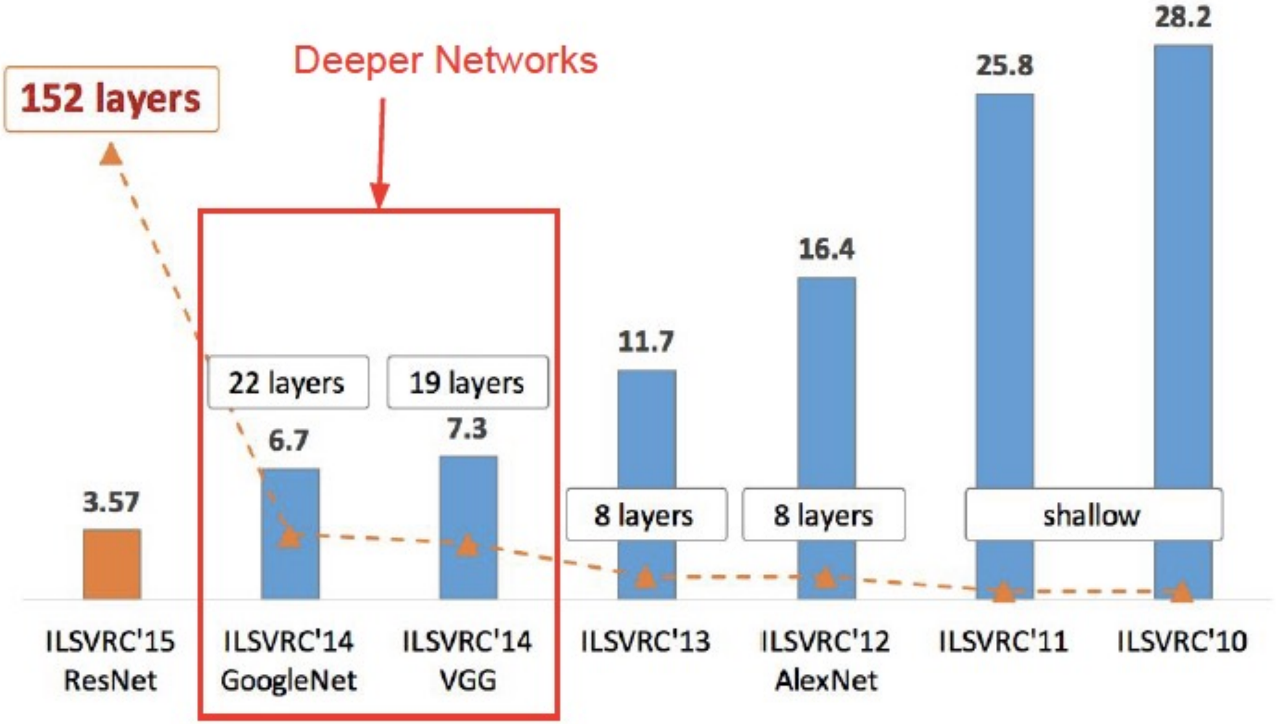


AlexNet (2012)



- AlexNet replaced the *tanh()* activation function used in LeNet5, with the ReLU function and also the MSE loss function with the Cross Entropy loss.
- AlexNet used a much bigger training set. Whereas LeNet5 was trained on the MNIST dataset with 50,000 images and 10 categories, AlexNet used a subset of the ImageNet dataset with a training set containing 1+ million images, from 1000 categories.
- AlexNet used Dropout regularization (= 0.5) to combat overfitting (but only in the Fully Connected Layers).

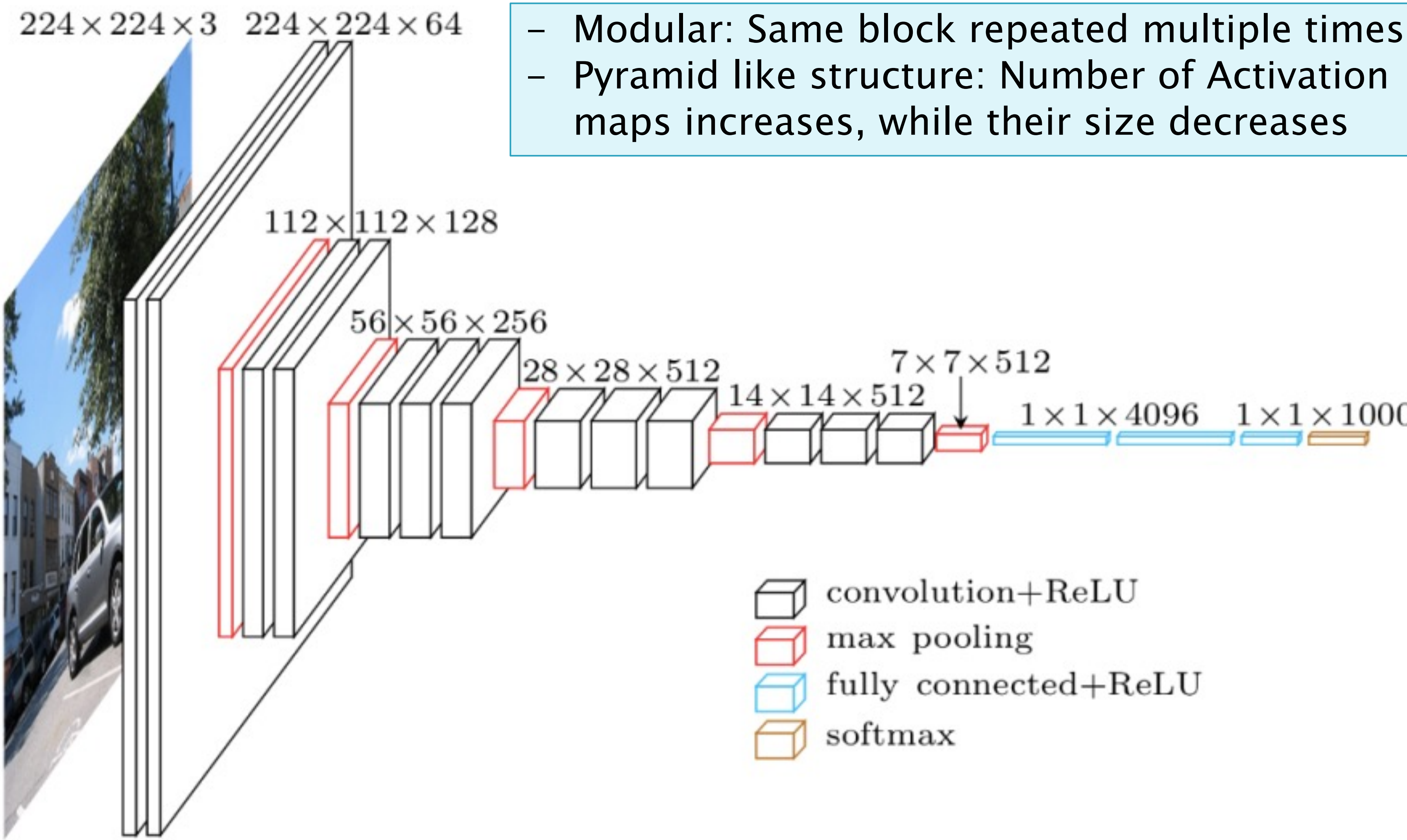
VGGNet (2014)



152 layers
Deeper Networks
28.2
25.8
16.4
11.7
22 layers
19 layers
6.7
7.3
3.57
8 layers
8 layers
shallow
ILSVRC'15
ResNet
ILSVRC'14
GoogleNet
ILSVRC'14
VGG
ILSVRC'13
ILSVRC'12
AlexNet
ILSVRC'11
ILSVRC'10

VGGNet (2014)

- Modular: Same block repeated multiple times
- Pyramid like structure: Number of Activation maps increases, while their size decreases

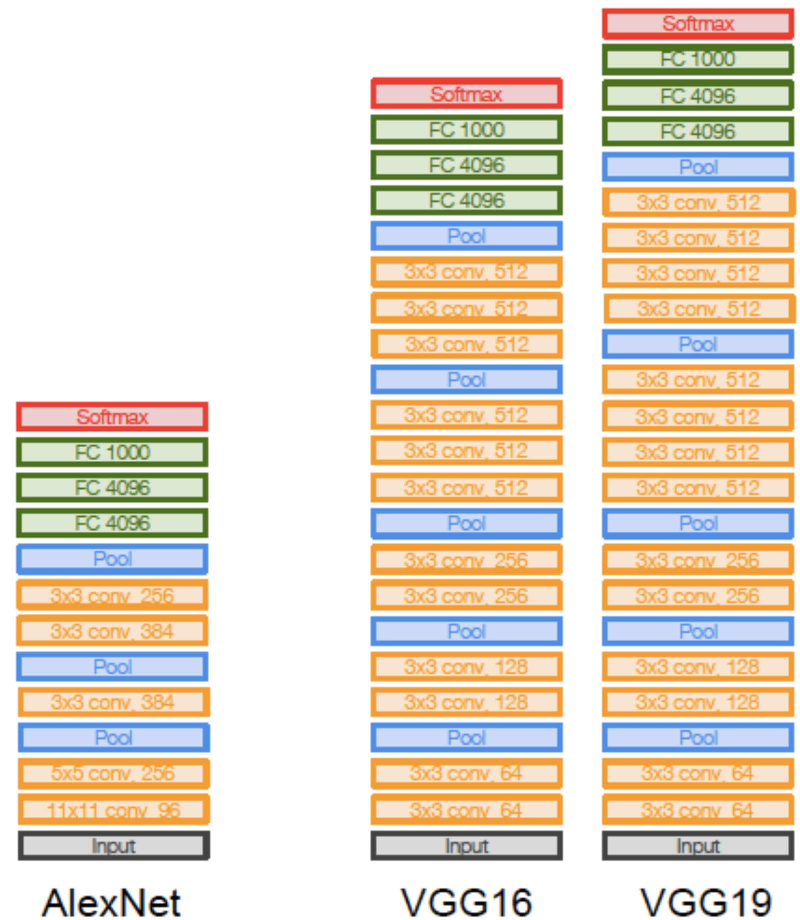


VGGNet

Case Study: VGGNet

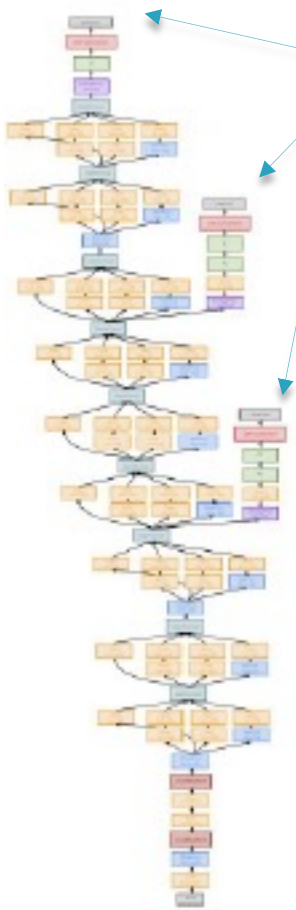
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

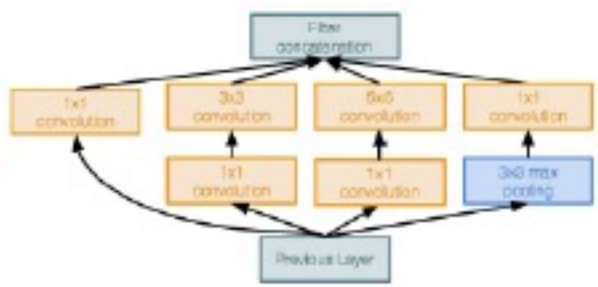


Google InceptionNet
(2014)

Google LeNet



Multiple Outputs



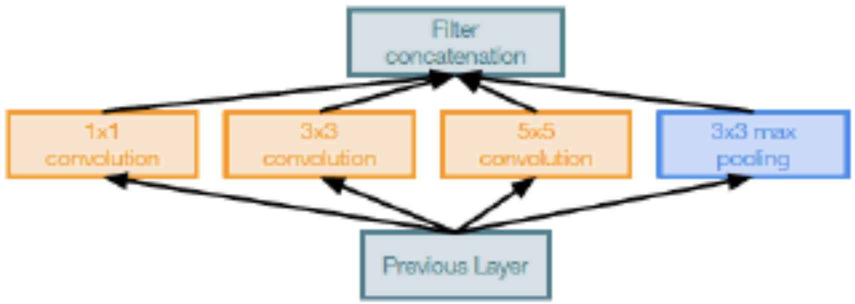
Inception module

Deeper networks, with computational efficiency

- 22 layers
- Efficient "Inception" module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC'14 classification winner (6.7% top 5 error)

"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other

Google LeNet



Naive Inception module

Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

Q: What is the problem with this?

Conv Ops:

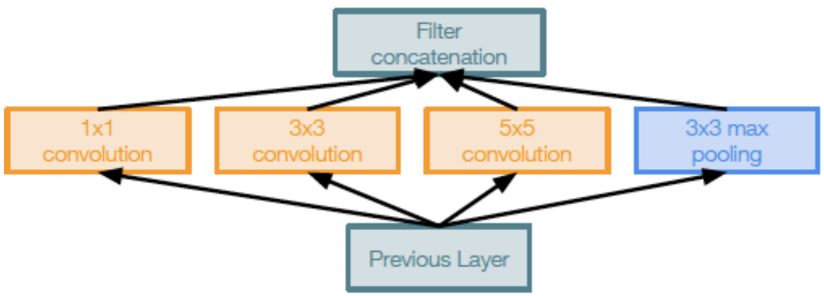
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

Very expensive compute

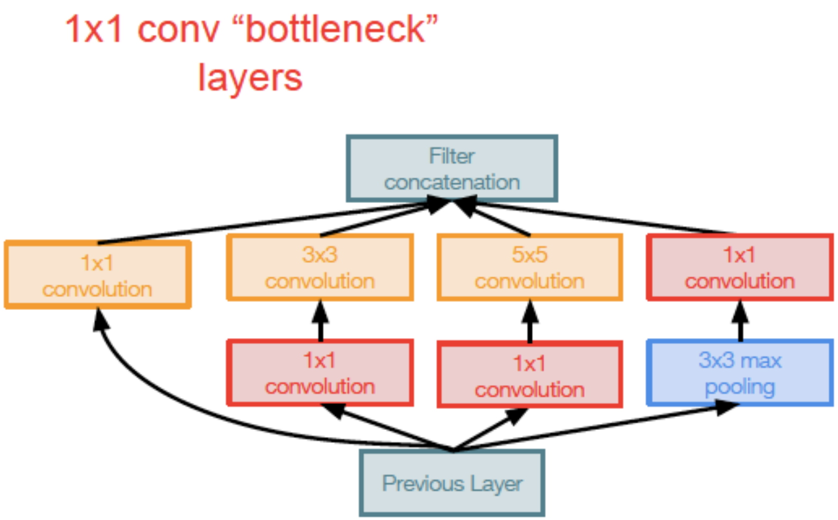
Google LeNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

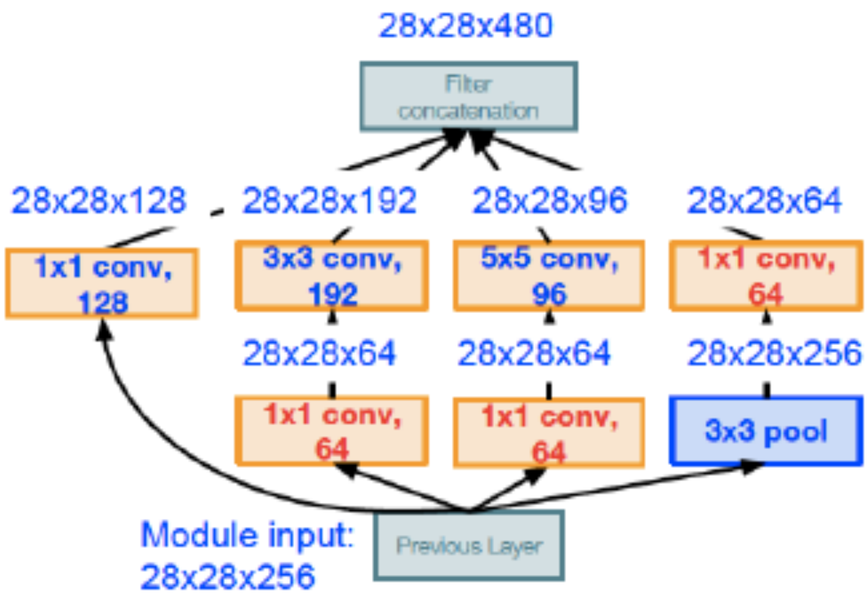


Naive Inception module



Inception module with dimension reduction

Google LeNet



Inception module with dimension reduction

Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:

Conv Ops:

[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

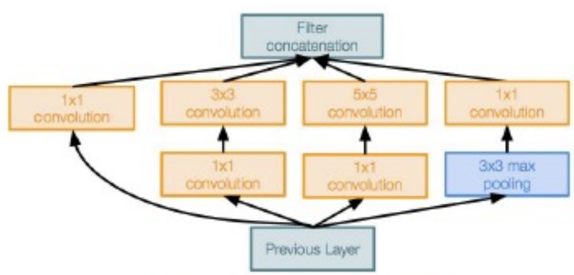
Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

Google LeNet

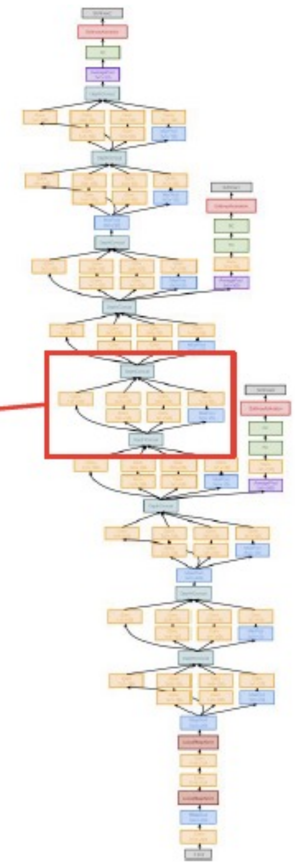
Case Study: GoogLeNet

[Szegedy et al., 2014]

Stack Inception modules
with dimension reduction
on top of each other



Inception module

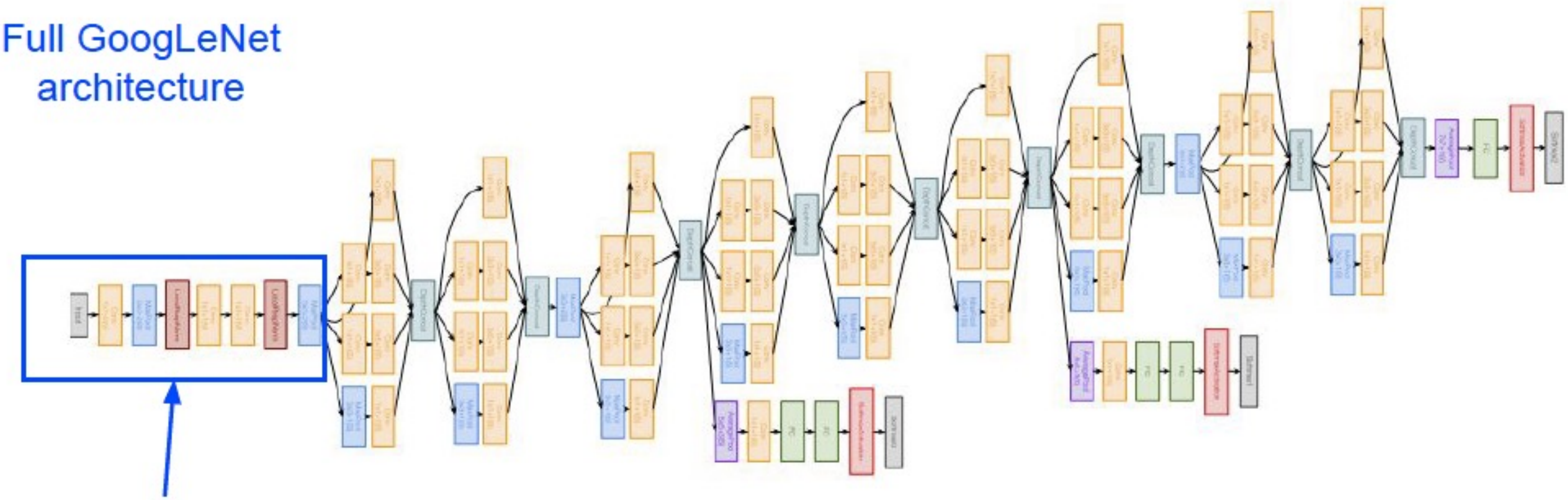


Google LeNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

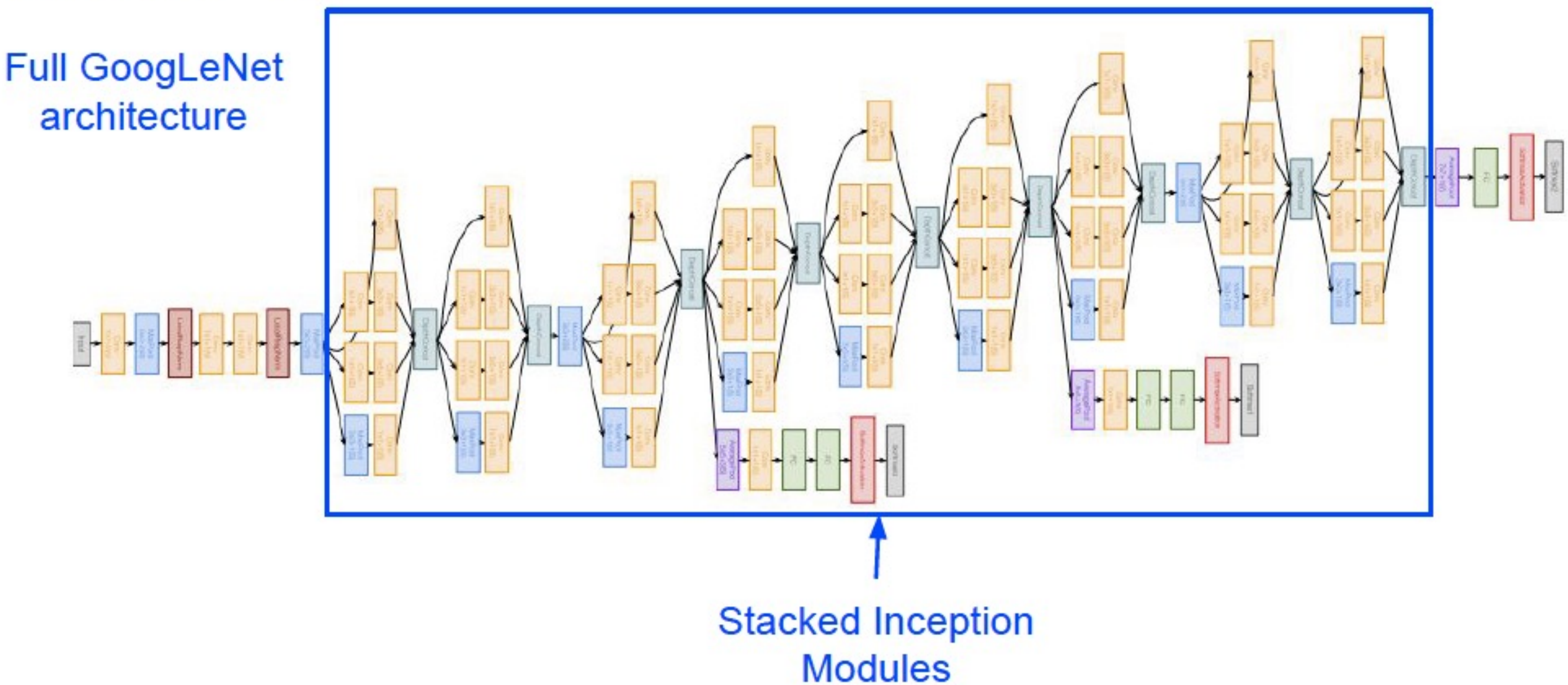


Stem Network:
Conv-Pool-
2x Conv-Pool

Google LeNet

Case Study: GoogLeNet

[Szegedy et al., 2014]



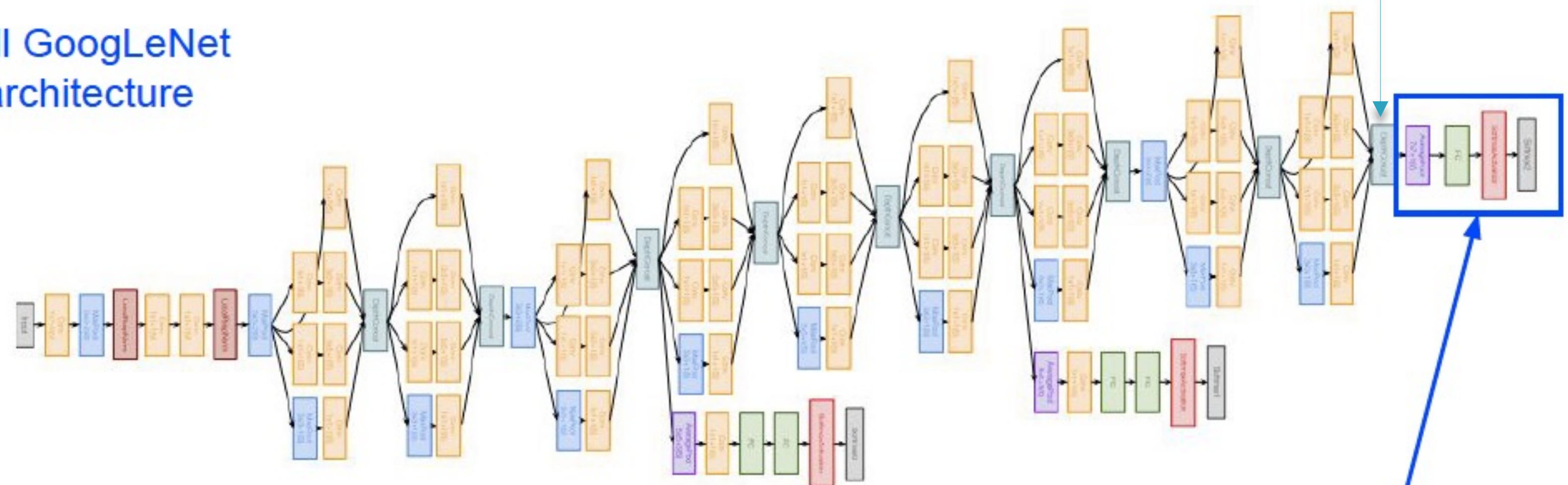
Google LeNet

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

Global Average Pooling



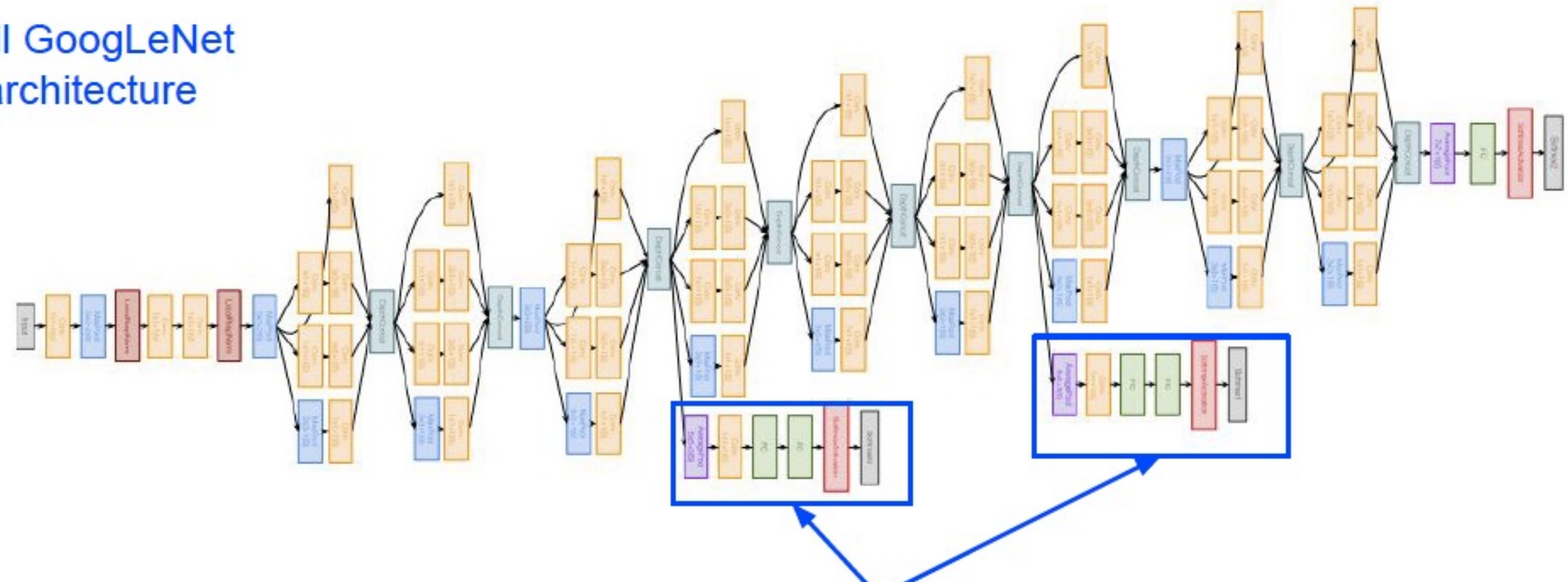
Classifier output
(removed expensive FC layers!)

Google LeNet

Case Study: GoogLeNet

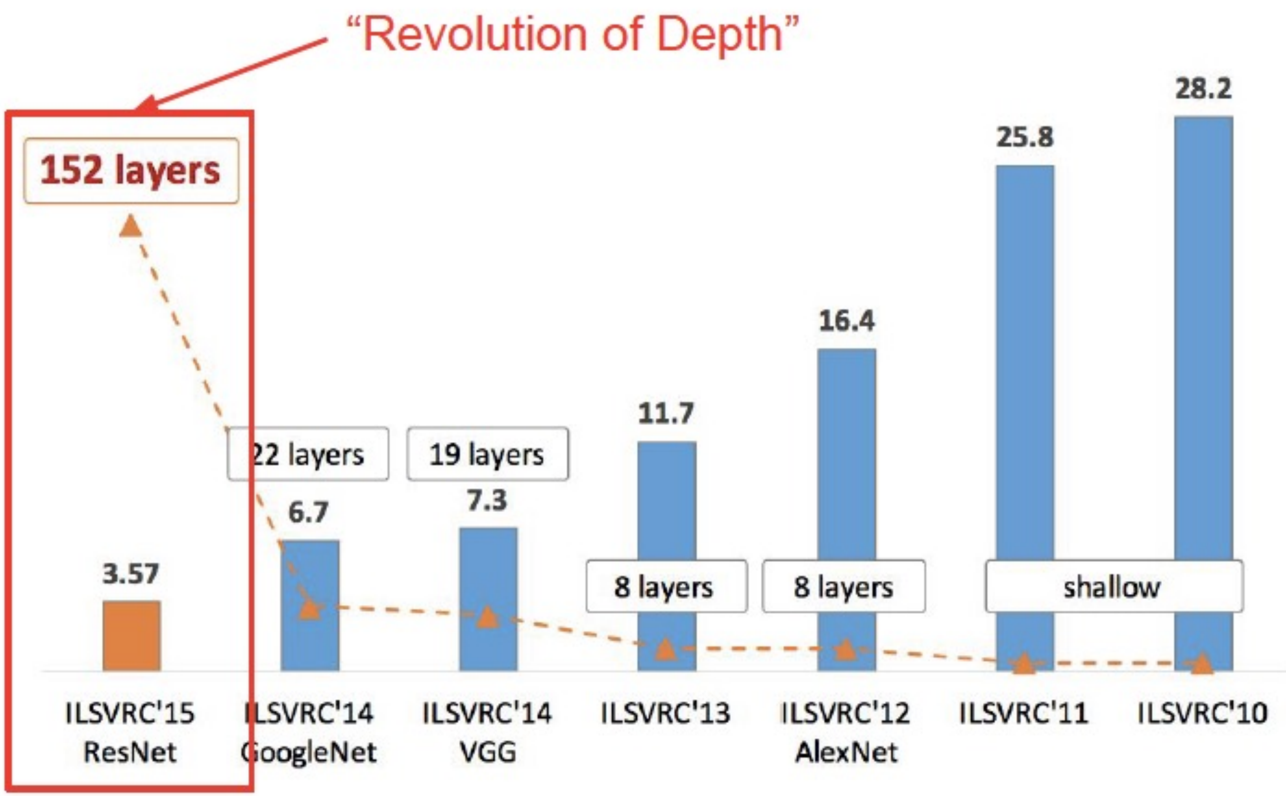
[Szegedy et al., 2014]

Full GoogLeNet architecture



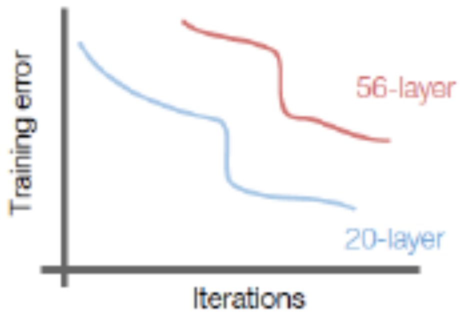
Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)

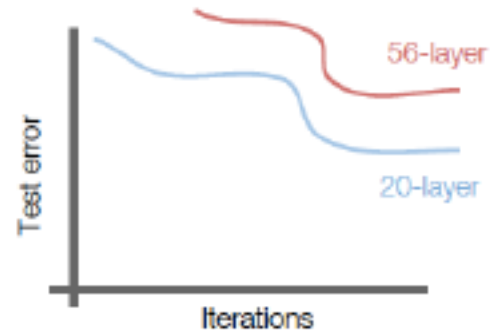
ResNet (2015)


"Revolution of Depth"
152 layers
22 layers
19 layers
8 layers
8 layers
shallow
3.57
6.7
7.3
11.7
16.4
25.8
28.2
ILSVRC'15
ResNet
ILSVRC'14
GoogleNet
ILSVRC'14
VGG
ILSVRC'13
ILSVRC'12
AlexNet
ILSVRC'11
ILSVRC'10

Deep Models

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



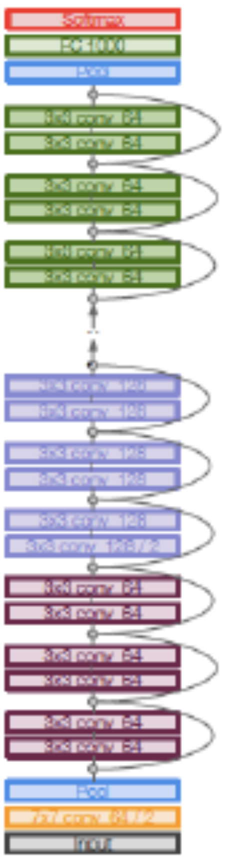


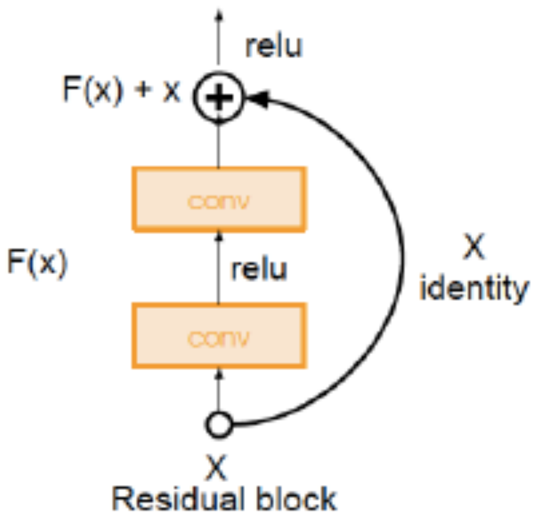
56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it’s not caused by overfitting!

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

ResNet

Fundamental New Idea: Residual Connections

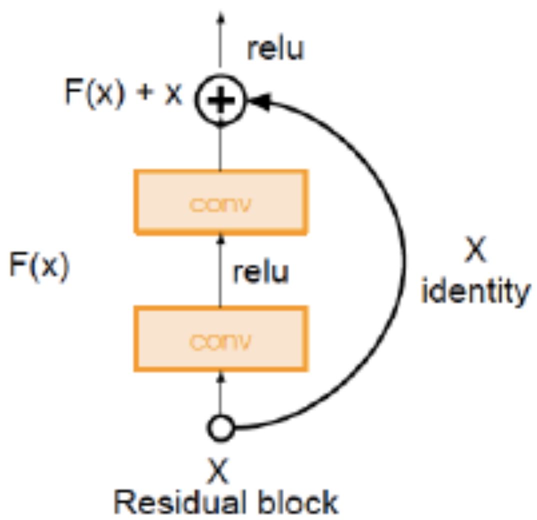




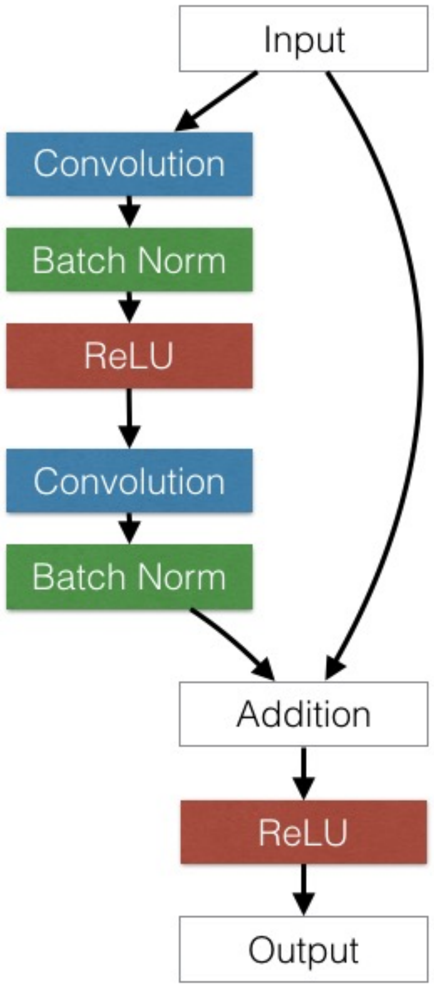
Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!

ResNet

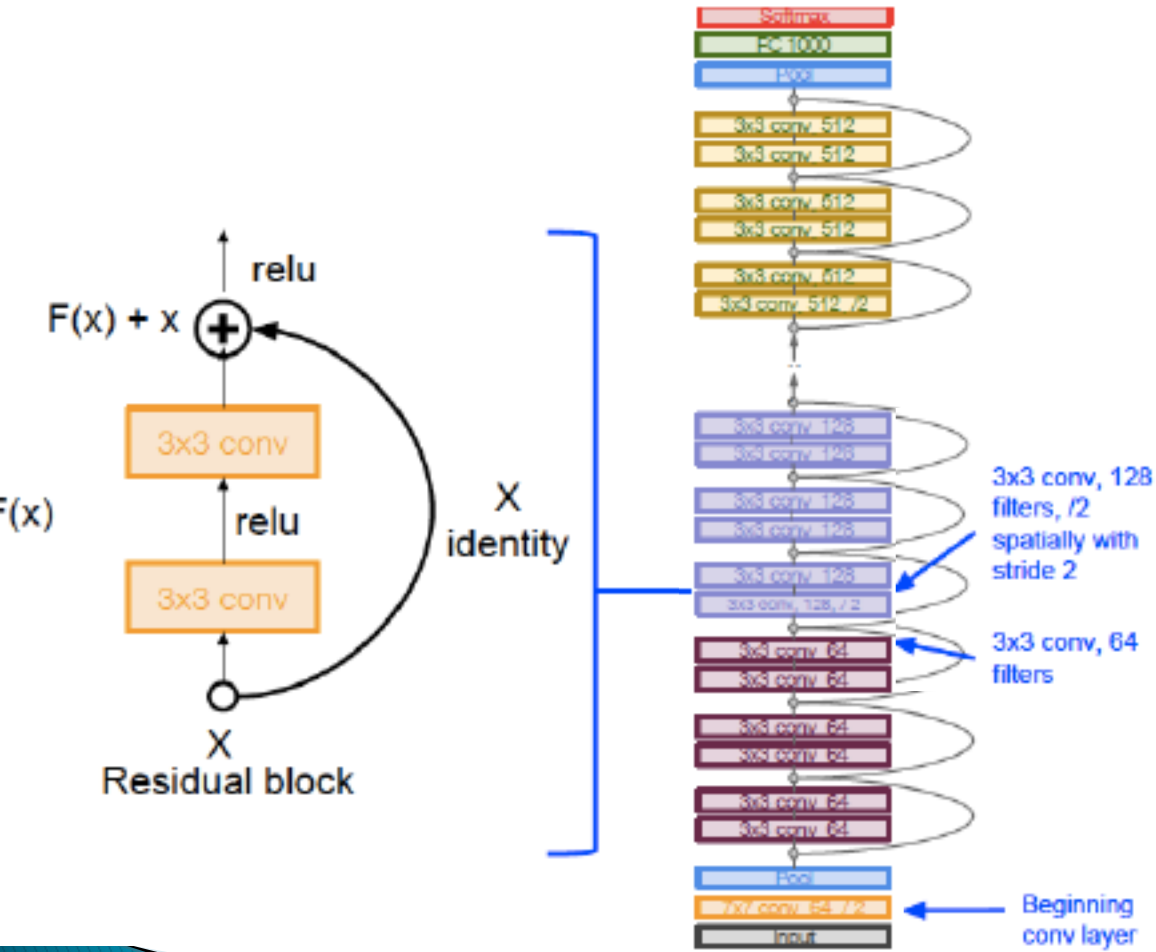


relu
F(x) + x
conv
F(x)
relu
X
identity
conv
X
Residual block



Input
Convolution
Batch Norm
ReLU
Convolution
Batch Norm
Addition
ReLU
Output

ResNet



Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)

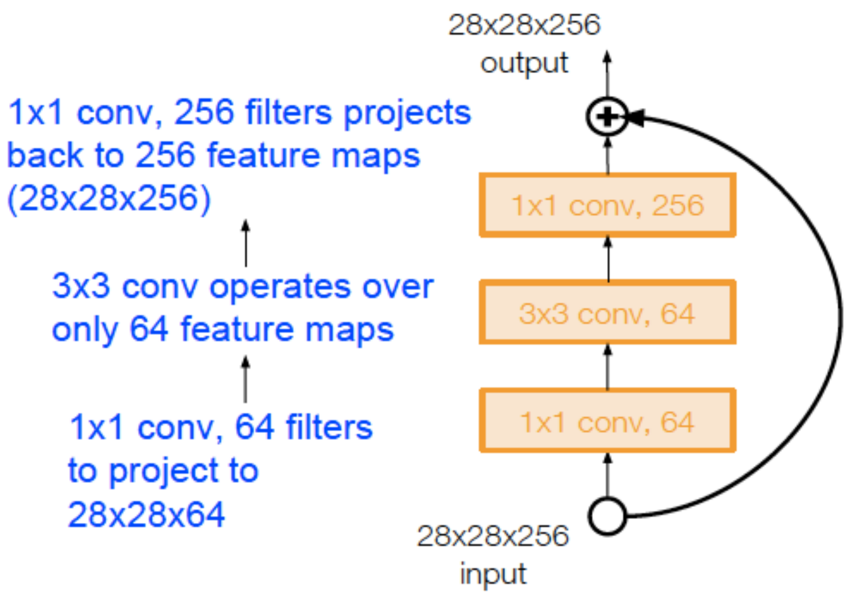
Global average pooling layer after last conv layer

Improved ResNets: Use 1x1 Layer

Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use "bottleneck" layer to improve efficiency (similar to GoogLeNet)



Effect of Skip Connections on Loss Surfaces

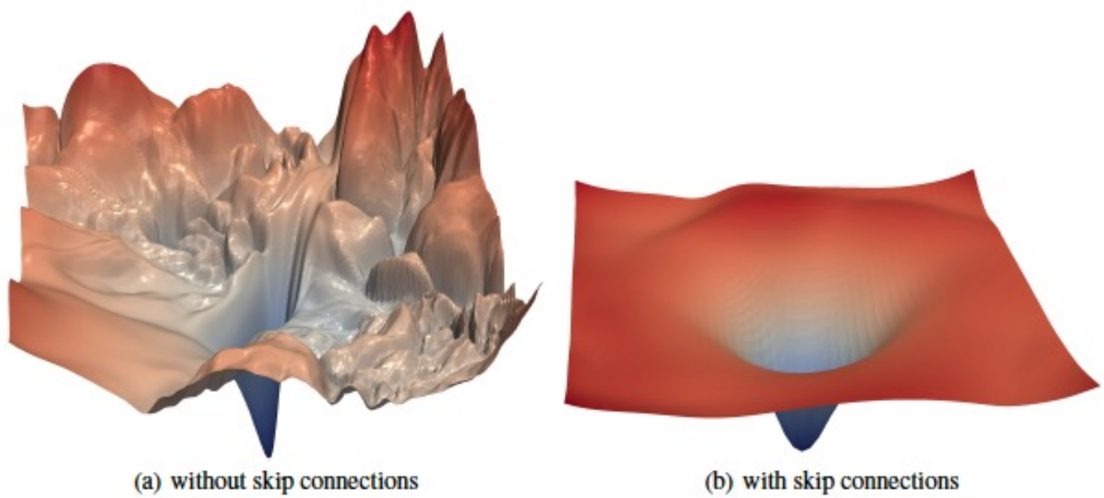


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

- The Loss Surface has become much more smooth and convex in shape, which makes it easier to run the SGD algorithm on it.
- In contrast, the chaotic shape of the Loss Surface without residual connections makes it very easy for the SGD algorithm to get caught in local minimums.

[Li, Xu et.al.] (https://arxiv.org/pdf/1712.09913.pdf)

Effect of Skip Connections on Loss Surfaces

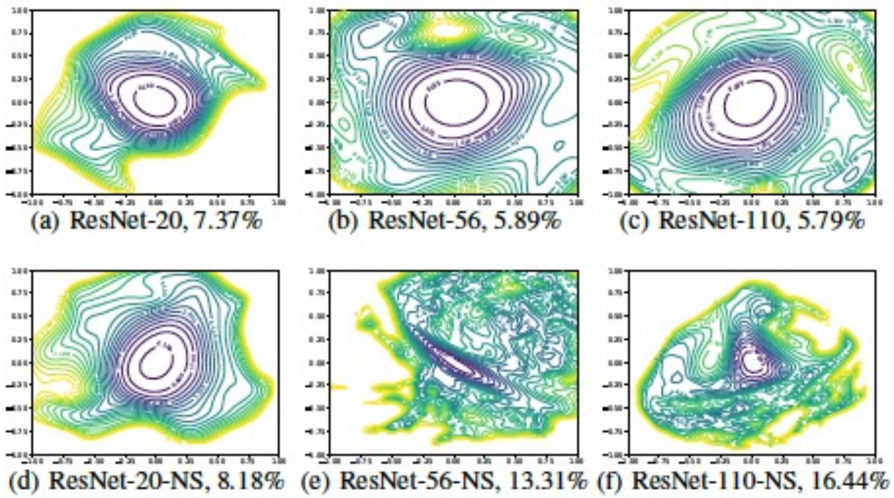
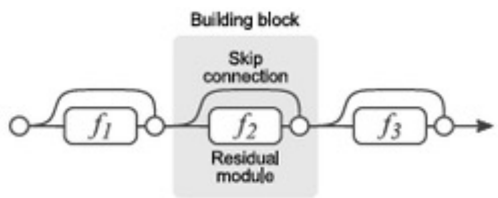


Figure 5: 2D visualization of the loss surface of ResNet and ResNet-noshort with different depth.

- Networks with a smaller number of layers, such as the 20 layer ResNet on the LHS, exhibit fairly well behaved Loss Surfaces, even in the absence of Residual Connections. Hence Residual Connections are not essential for smaller networks.
- Networks with a larger number of layers on the other hand, start to exhibit chaotic non-convex behavior in their Loss Surface in the absence of Residual Connections.

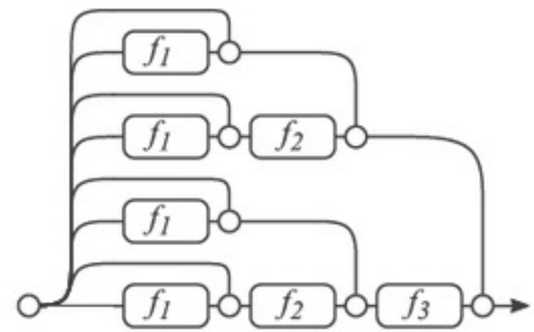
[Li, Xu et.al.] (https://arxiv.org/pdf/1712.09913.pdf)

Effect of Skip Connections on Forward Paths



(a) Conventional 3-block residual network

=

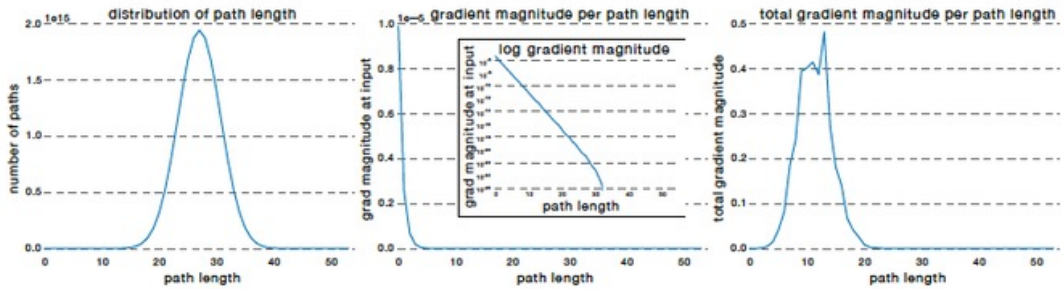


(b) Unraveled view of (a)

- In a network with no Residual Connections, there exists only a single forward path and all data flows along it. However in a network with n Residual Connections, there exist 2^n forward paths. This is illustrated for the case $n=3$. The 8 separate forward paths that exist in this network are shown in Part (b) of this figure.
- As a result, the network decisions are effectively made by all of these 8 forward paths, that are operating parallel. This is very much like what is done in the Ensemble method, in which multiple models operate in parallel to improve model accuracy.

[Veit, Wilbur, Belongie] (https://arxiv.org/pdf/1605.06431.pdf)

Effect of Skip Connections on Forward Paths



(a) (b) (c)

- They furthermore showed that gradient flow in the backwards direction is dominated by a few shorter paths. This is illustrated in the figure which has results for a network with 54 Residual Connections. Part (a) of this figure shows the distribution of the path lengths in this network, while Part (b) plots the gradient magnitudes.
- They further multiplied the gradient magnitudes with the number of pathlengths for a particular path, and and obtained the graph in Part (c). As can be seen the majority of the gradients are contributed by path lengths of 5 to 17, while the higher path lengths contribute no gradient at all.
- From this they concluded that in very deep networks with hundreds of layers, Residual Connections avoid the vanishing gradient problem by introducing short paths which can carry the gradient throughout the extent of these networks.

[Veit, Wilbur, Belongie] (https://arxiv.org/pdf/1605.06431.pdf)

Implementing Residual Connections

Listing 9.2 Residual block where the number of filters changes

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set ide the esidual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.

Listing 9.3 Case where the target block includes a max pooling layer

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

Chollet Page 253

Implementing Modular Architectures

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)

x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=5, use_bias=False)(x)

for size in [32, 64, 128, 256, 512]:
    residual = x

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

    residual = layers.Conv2D(
        size, 1, strides=2, padding="same", use_bias=False)(residual)
    x = layers.add([x, residual])

x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

We use the same data augmentation configuration as before.

Don't forget input scaling!

In the original model, we used a Flatten layer before the Dense layer. Here, we go with a GlobalAveragePooling2D layer.

Like in the original model, we add a dropout layer for regularization.

We apply a series of convolutional blocks with increasing feature depth. Each block consists of two batch-normalized depthwise separable convolution layers and a max pooling layer, with a residual connection around the entire block.

Note that the assumption that underlies separable convolution, "feature channels are largely independent," does not hold for RGB images! Red, green, and blue color channels are actually highly correlated in natural images. As such, the first layer in our model is a regular Conv2D layer. We'll start using SeparableConv2D afterwards.

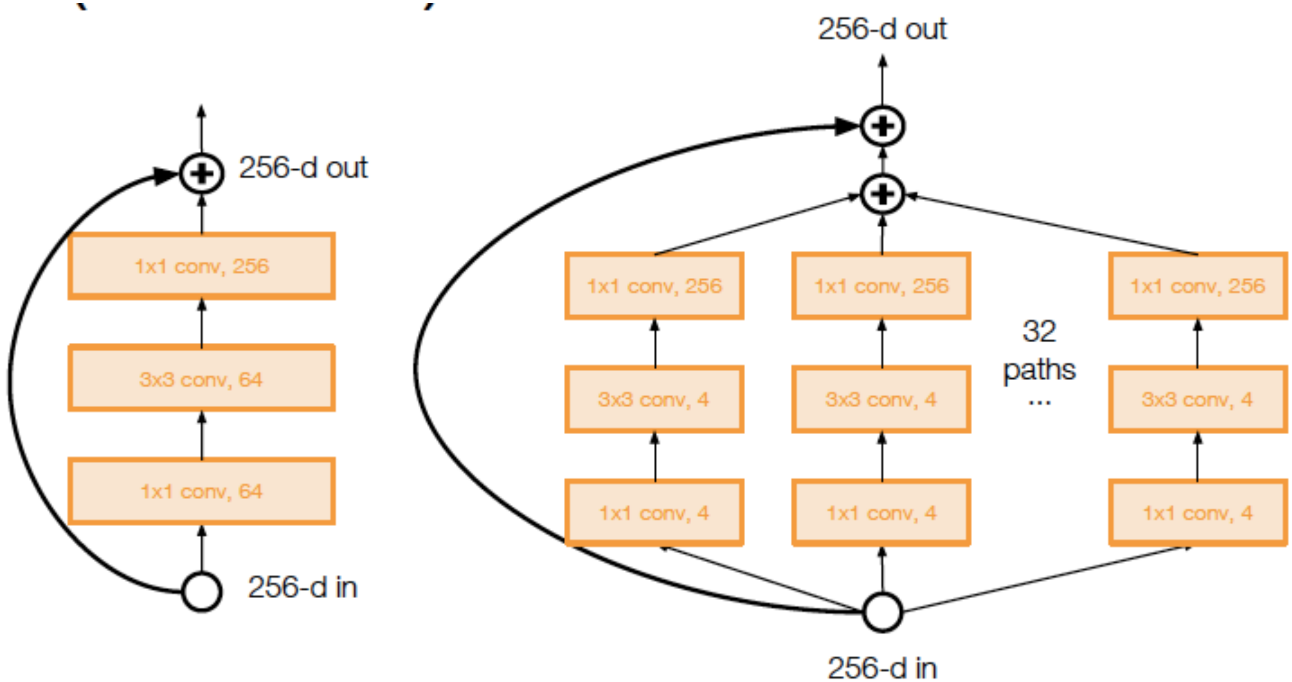
Chollet: Page 260

Beyond ResNets

Improving ResNets: ResNext

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways (“cardinality”)
- Parallel pathways similar in spirit to Inception module



Grouped Convolutions

101 Layer ResNext has better Accuracy than a 200 Layer ResNet

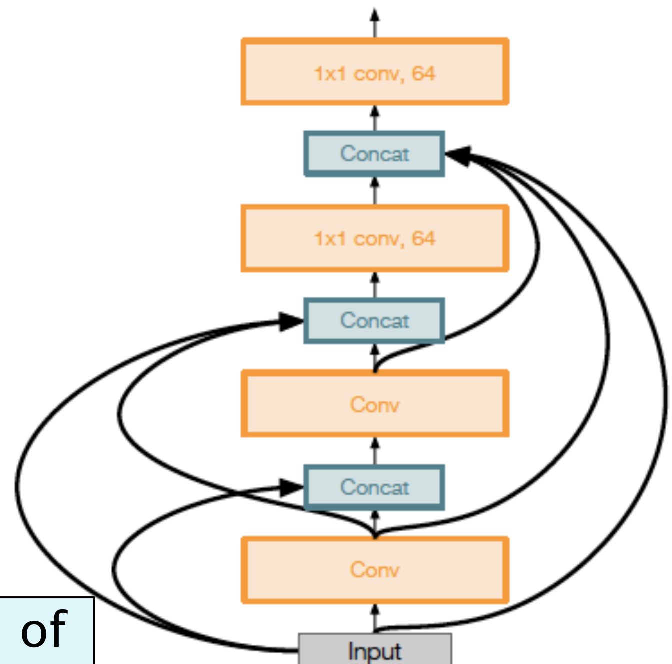
Improving ResNets: DenseNet

Densely Connected Convolutional Networks

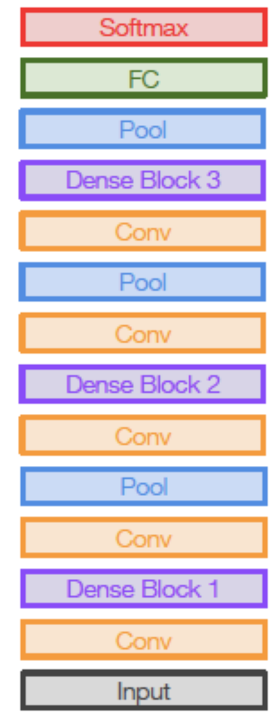
[Huang et al. 2017]

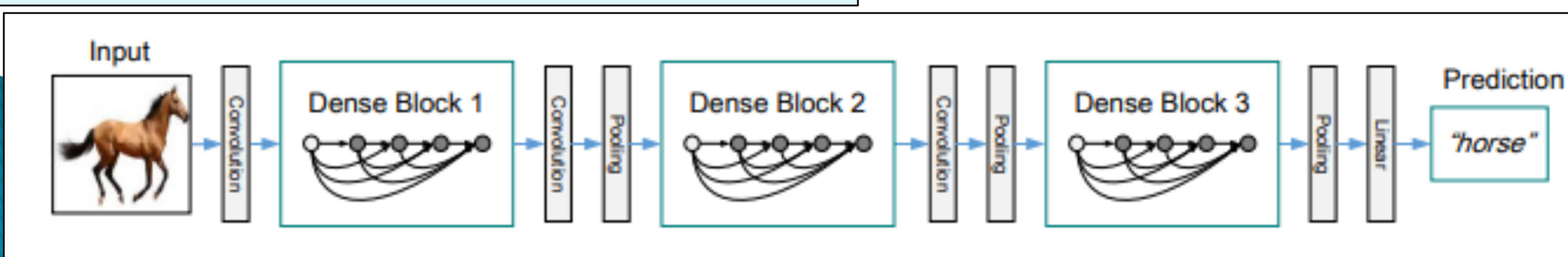
- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse

Instead of addition, uses concatenation of activation maps to combine layers

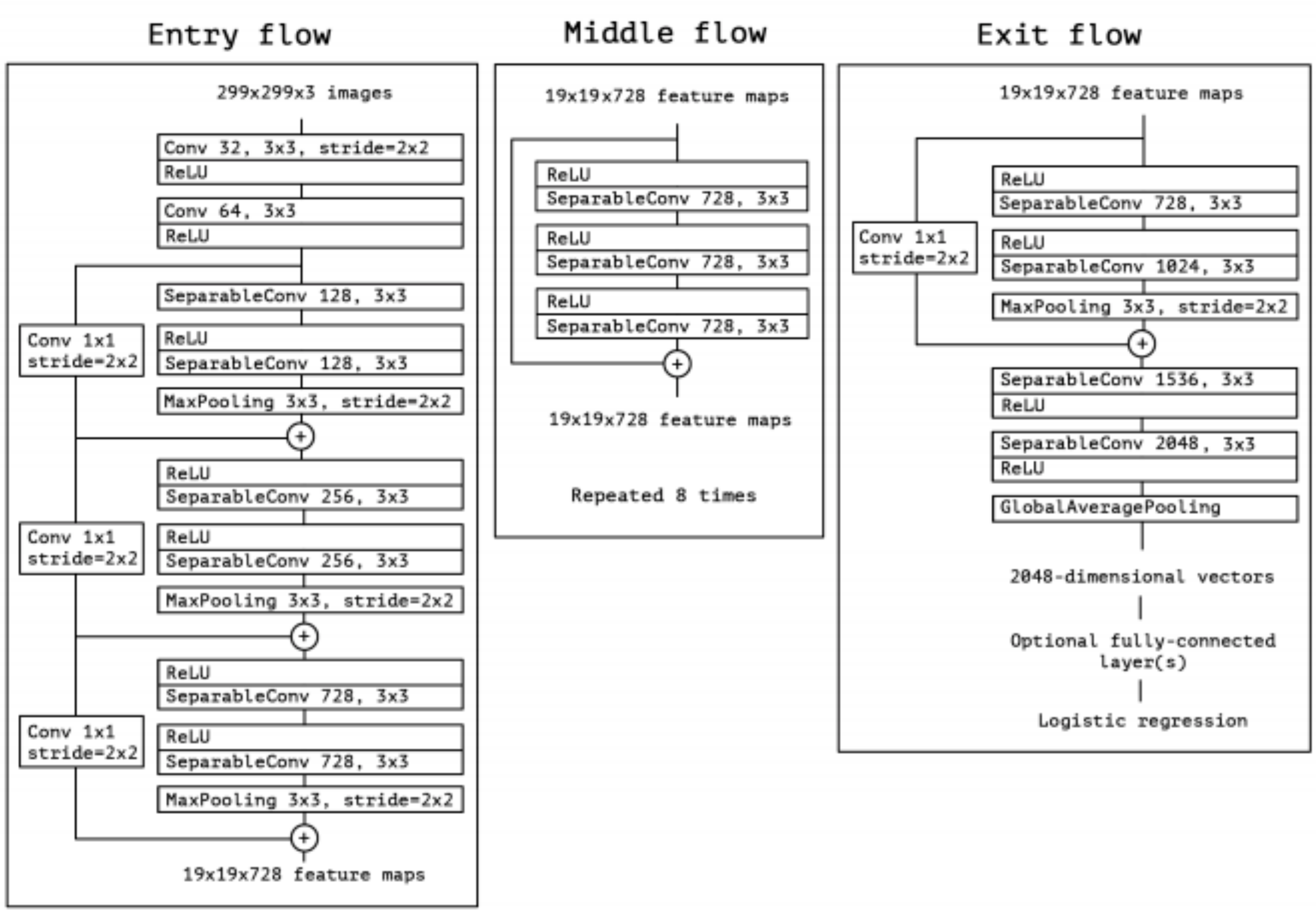


Dense Block





Improving ResNets – XceptionNet



Improving ResNets – MobileNet

- No pooling, downsampling done using strided convolutions
- 95% of the computations done in 1x1 convolutions, which can be implemented very efficiently

Table 3. Resource usage for modifications to standard convolution. Note that each row is a cumulative effect adding on top of the previous row. This example is for an internal MobileNet layer with $D_K = 3$, $M = 512$, $N = 512$, $D_F = 14$.

Layer/Modification	Million Mult-Adds	Million Parameters
Convolution	462	2.36
Depthwise Separable Conv	52.3	0.27
$\alpha = 0.75$	29.6	0.15
$\rho = 0.714$	15.1	0.15

Table 4. Depthwise Separable vs Full Convolution MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

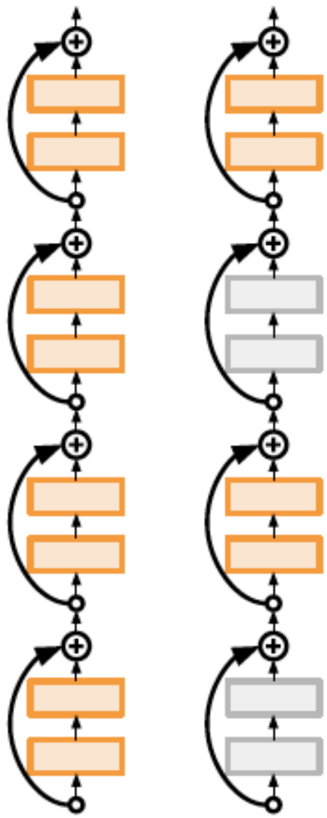
Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Improving ResNets: Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time



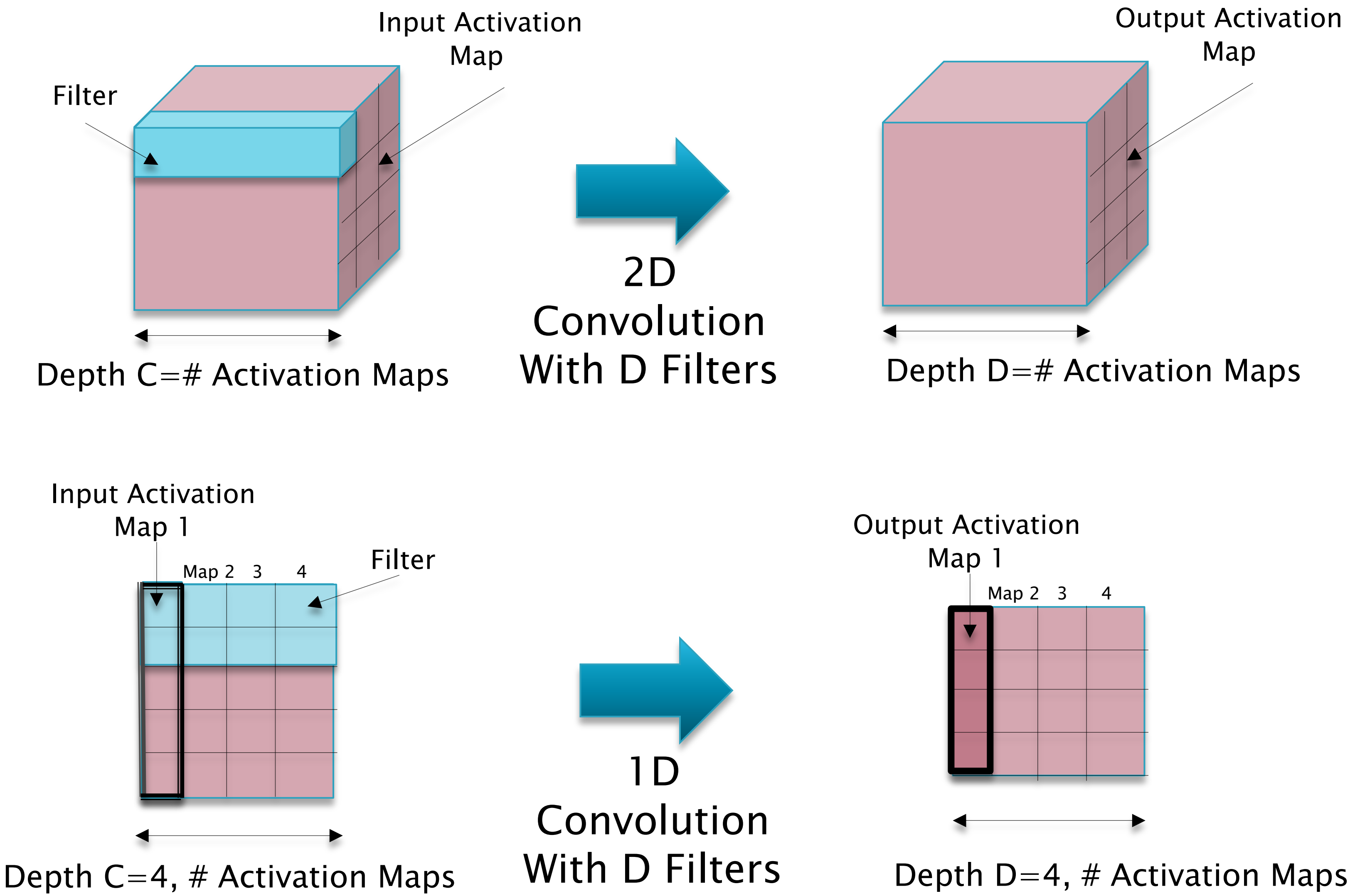
This is a type of Regularization!

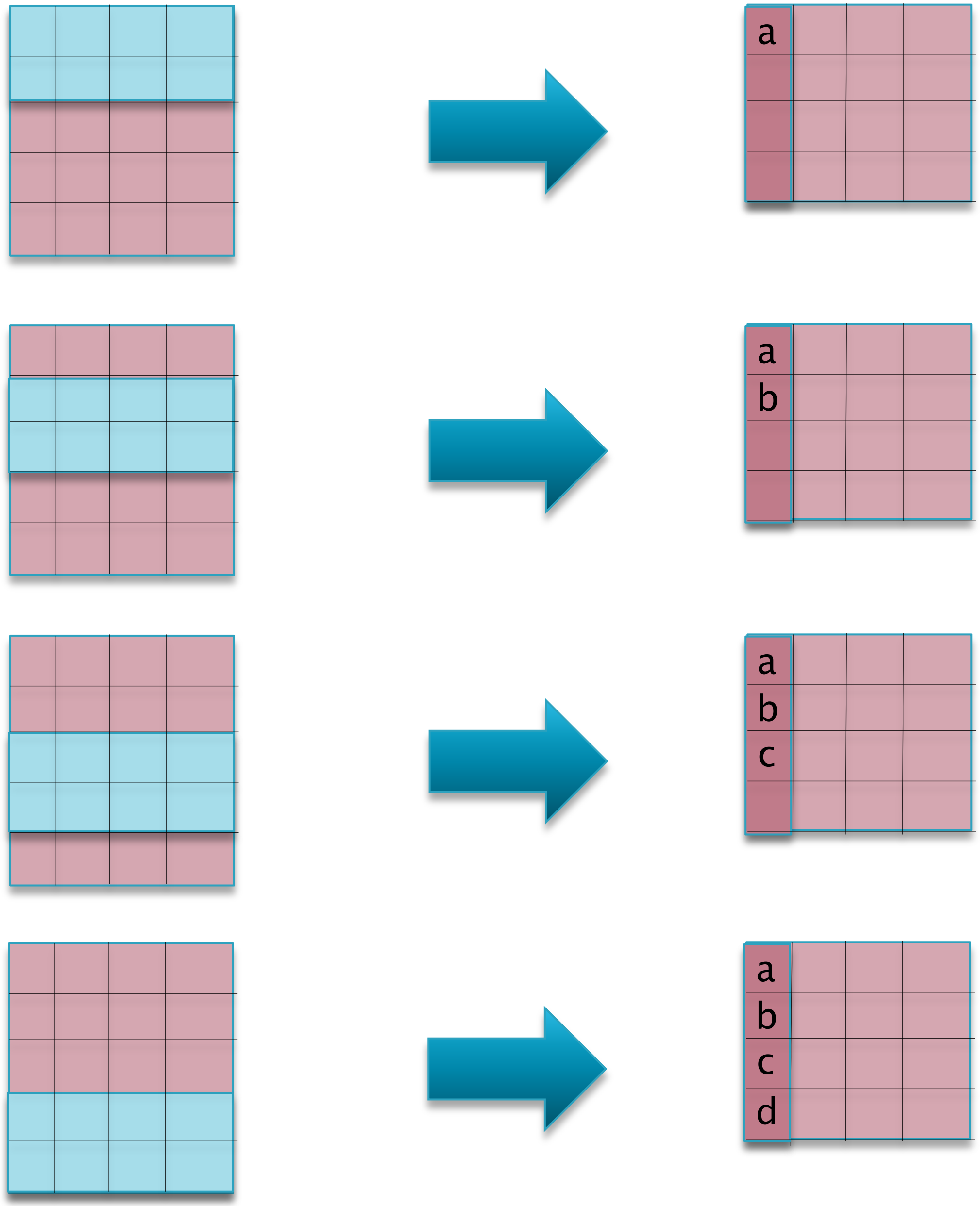
One Dimensional Convolutions

Why Use 1D Convolutions?

A way in which Convolutional Networks can be used for:

- Natural Language Processing
- Tabular Data (CSV or Excel)

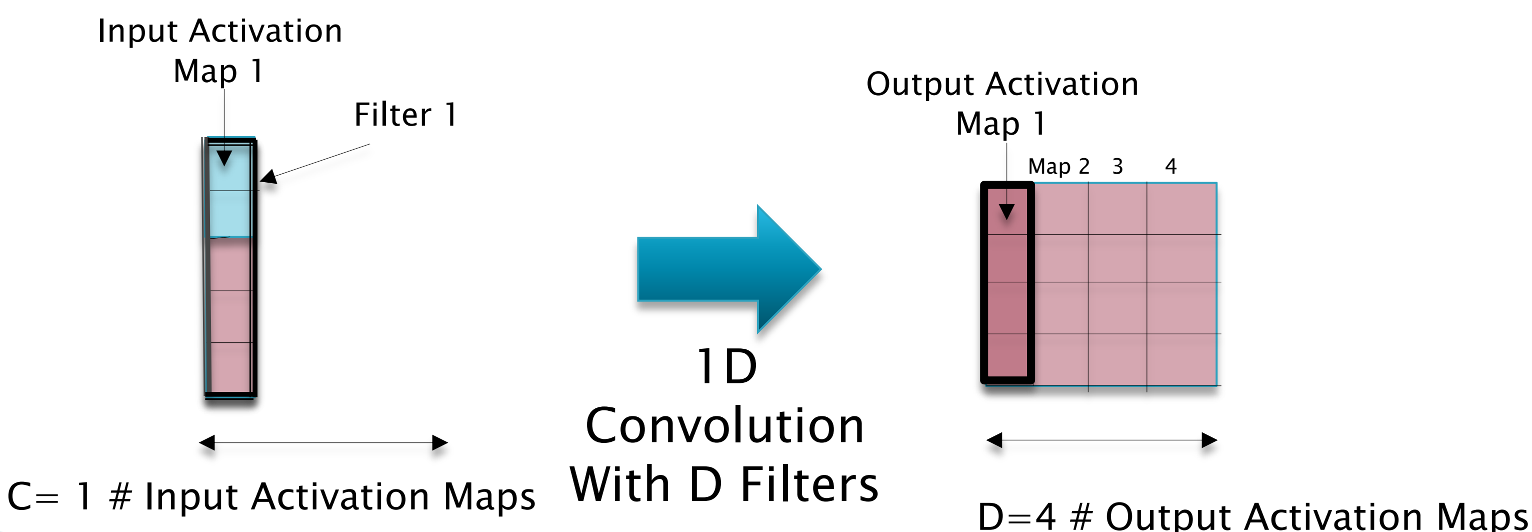

Filter
Input Activation Map
2D Convolution With D Filters
Output Activation Map
Depth C=# Activation Maps
Depth D=# Activation Maps
Input Activation Map 1
Map 2 3 4
Filter
1D Convolution With D Filters
Output Activation Map 1
Map 2 3 4
Depth C=4, # Activation Maps
Depth D=4, # Activation Maps



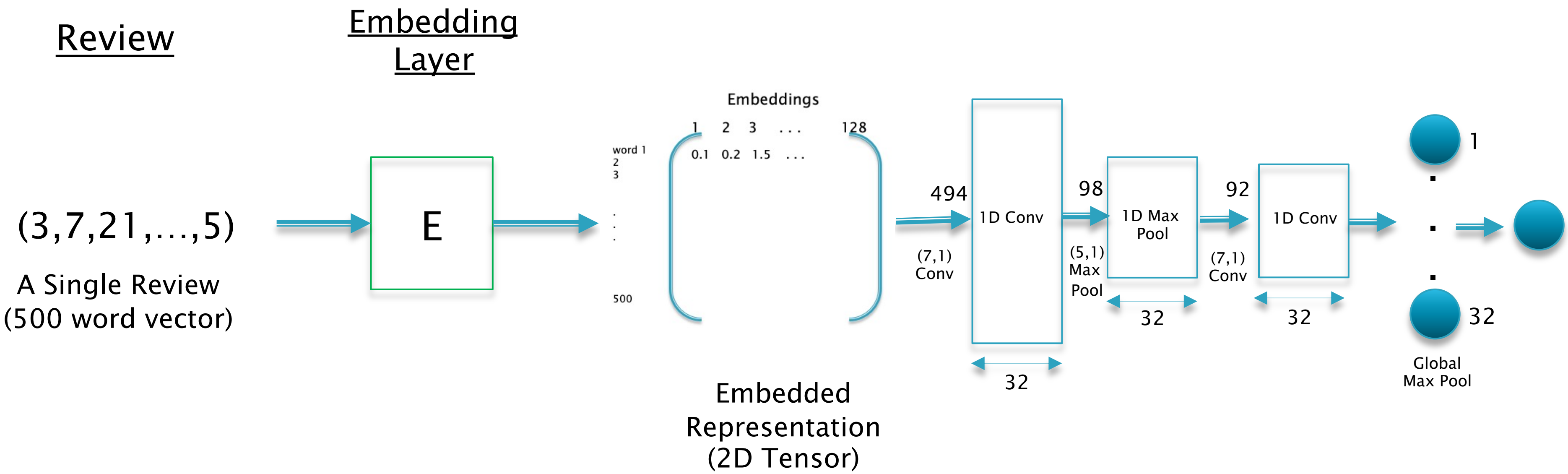
1D Convolution
With 2x1 Filter
With Depth 4

Why are 1D Convolutions Useful?

- 1-D Convolutions can be used to process 1D input data. Examples:
  - NLP
  - Tabular Data: Good alternative to using Dense Feed Forward Networks



Processing IMDB Reviews with 1D ConvNets



$$N2 = \frac{N1 - F + 2P}{S} + 1$$

1D Convolutions in Keras

```
from keras.models import Sequential
from keras import layers
from tensorflow.keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(learning_rate=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Further Reading

- Das and Varma: Chapter ConvNetsPart2
- Chollet Chapter 9, Section 9.3