Convolutional Neural Network

Convolutional Neural Networks (CNNs)

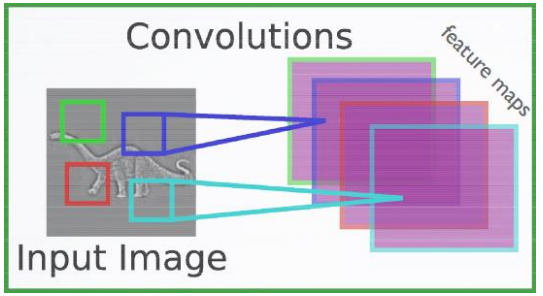
CNNs vs MLPs

- Naively using MLP to classify 224x224x3 (~ 3 x 40,000) typical ImageNet image -> parameter explosion
  - ❑ Doesn't exploit local spatial information
- Can we build special neural nets for images exploiting
  - ❑ 2D topology of pixels
  - ❑ Achieve invariance to translation ?

Category: tiger
ImageNet

Convolutional networks leverage these ideas,

- ❑ Local connectivity
- ❑ Parameter sharing
- ❑ Pooling/ Subsampling
- ❑ ReLu (rectifier) nonlinearity



*slide courtesy, Hugo Larochelle course on Neural networks

Convolutional Neural Networks (CNNs)

What we will learn about CNNs?

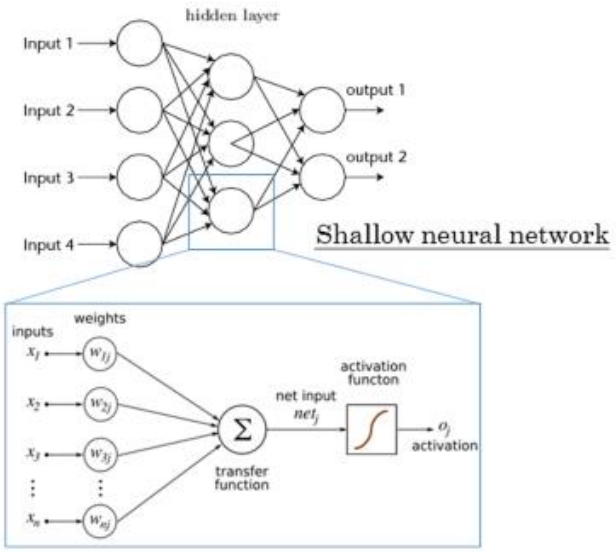
Introduction

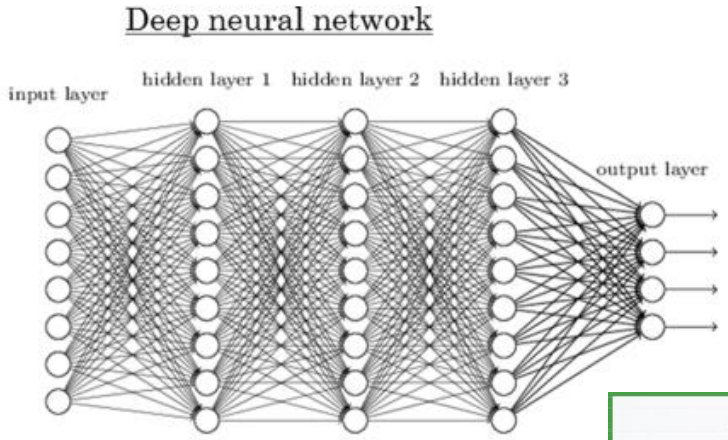
- CNNs over fully connected
- Different layers in CNN
  - ❑ Convolutional layer
  - ❑ RelU
  - ❑ Pooling layer
  - ❑ Fully connected layer

State of the art architectures

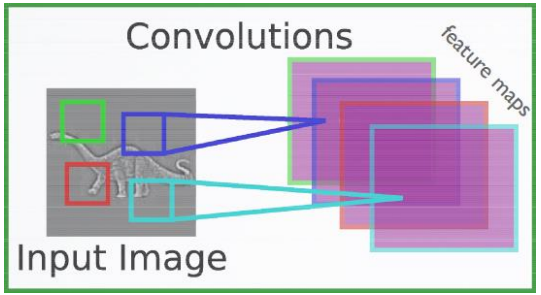
- AlexNet and VGG Net
- GoogLeNet (inception module)
- ResNets (Residual module)
  CVPR’ 17 best paper

CNNs over Multi-layer neural networks (MLNN)





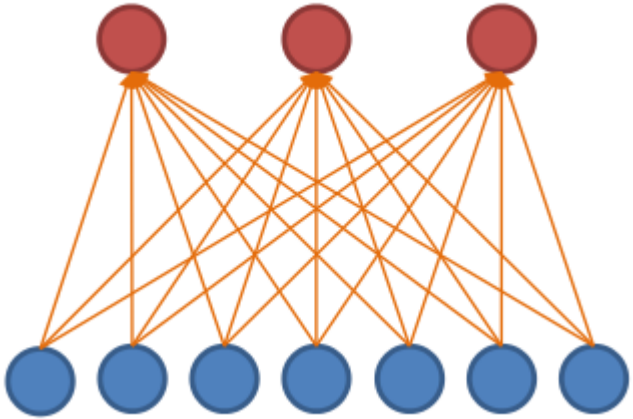
Multi-layer neural network



CNNs are **multi-layer neural network with two constraints**:

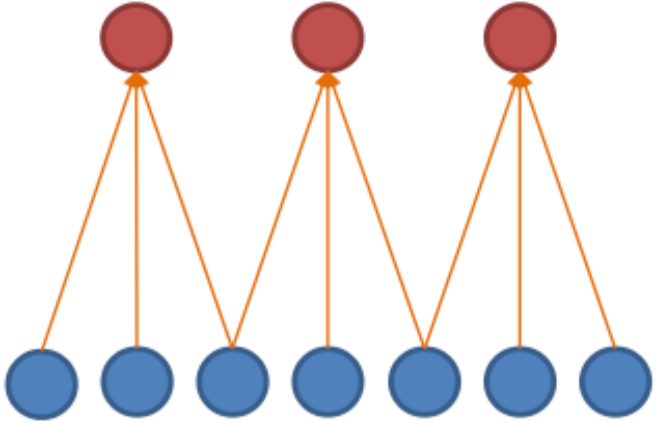
1. Local connectivity
2. Parameter sharing

CNN: Local connectivity (LC)



Hidden layer (3 nodes)

Input layer (7 nodes)



MLNN (7 X 3 = 21 parameters)

MLNN-LC (3 X 3 = 9 parameters)

2.3X runtime and storage efficient.

In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

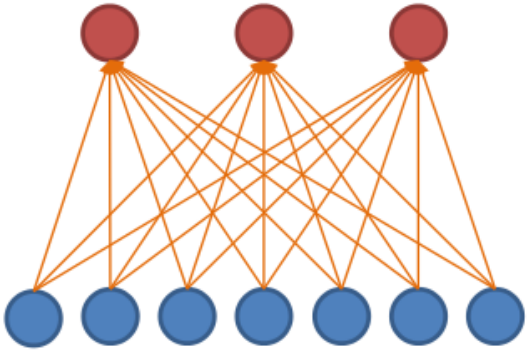
MLNN have

1. *m x n* parameters to store.
2. O(*m x n*) runtime

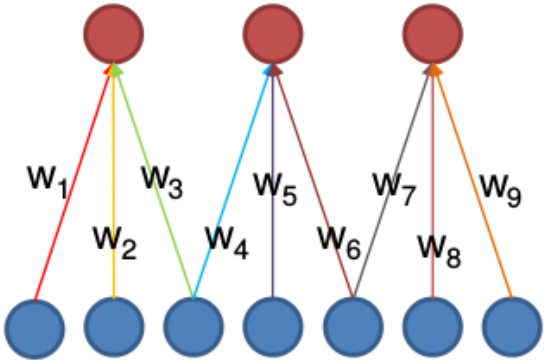
MLNN-LC have:

1. *k x n* parameters to store.
2. O(*k x n*) runtime

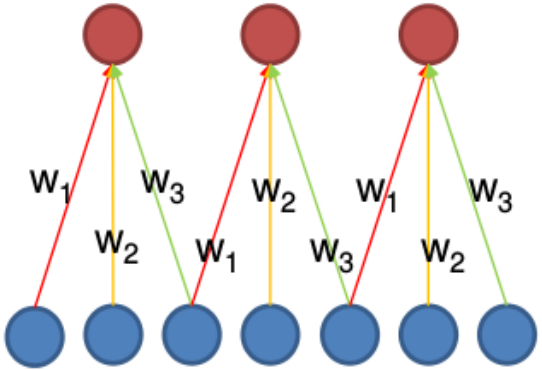
CNN: Parameter sharing (PS)



MLNN (21 parameters)



MLNN-LC (3 X 3 = 9 parameters)
2.3X runtime and storage efficient.



MLNN-LC-PS (3 parameters)
2.3X faster,
& 7X storage efficient.

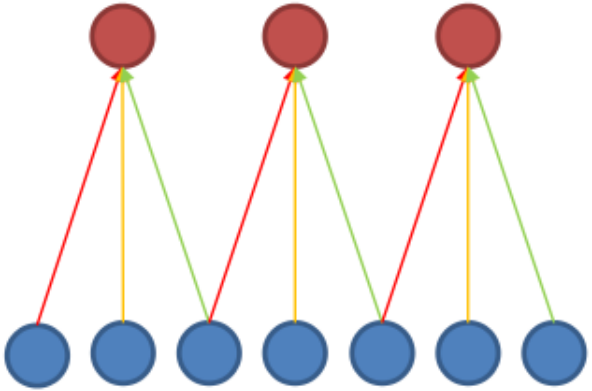
In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

MLNN have
1. *m x n* parameters to store.
2. O(*m x n*) runtime

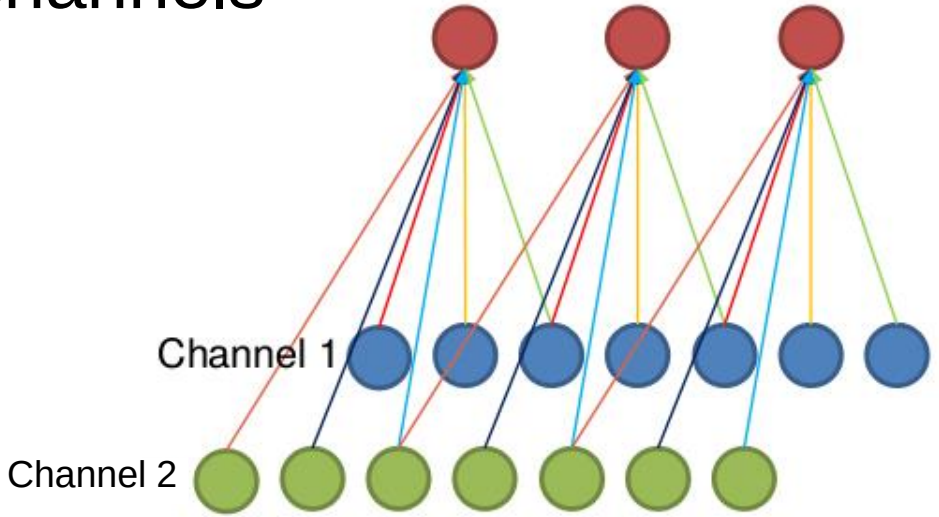
MLNN-LC have:
1. *k x n* parameters to store.
2. O(*k x n*) runtime

MLNN-LC-PS have:
1. *k* parameters to store.
2. O(*k x n*) runtime

CNN with multiple input channels



Single input channel

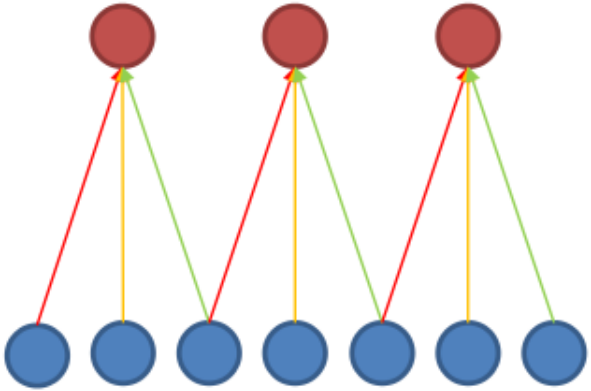


Two input channels

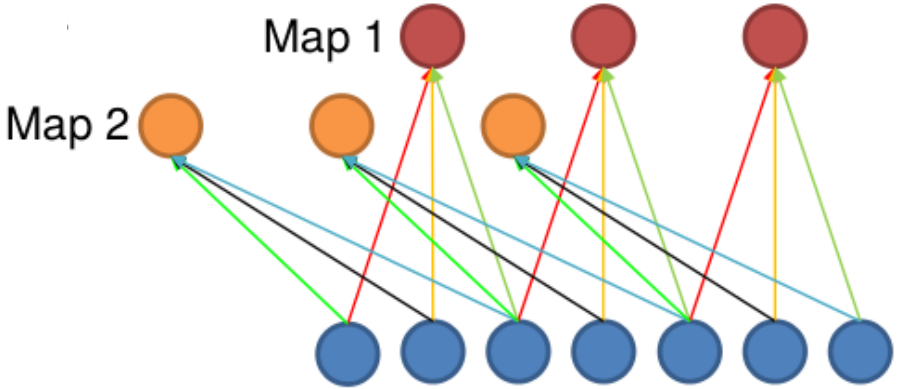
Filter weights

Filter weights

CNN with multiple output maps

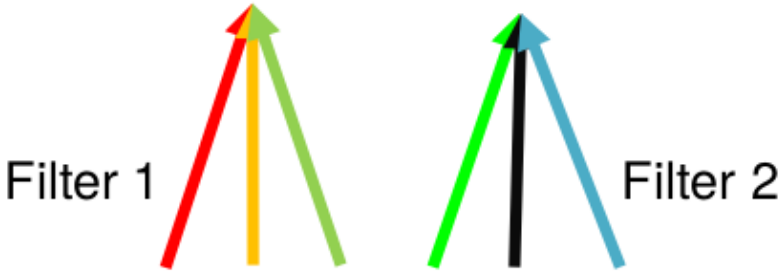


Single input map



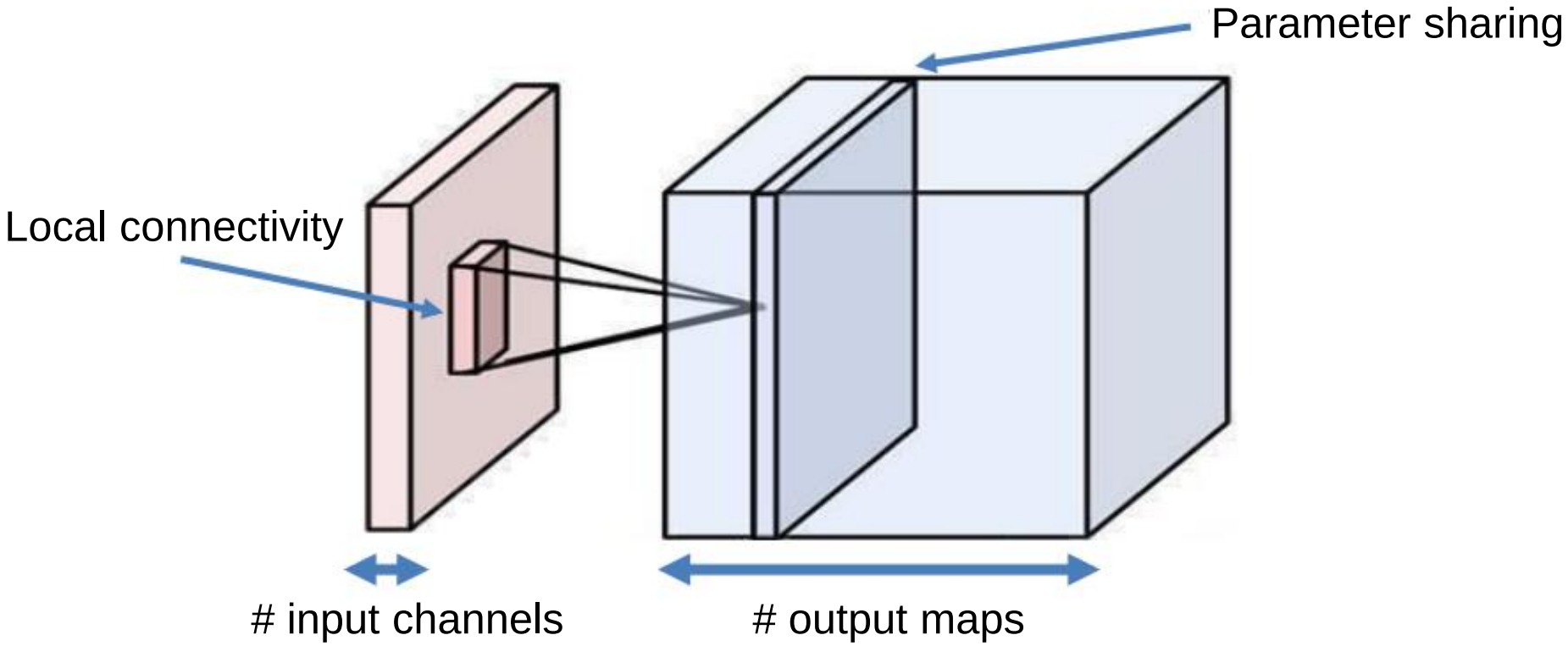
Two output maps

Filter weights



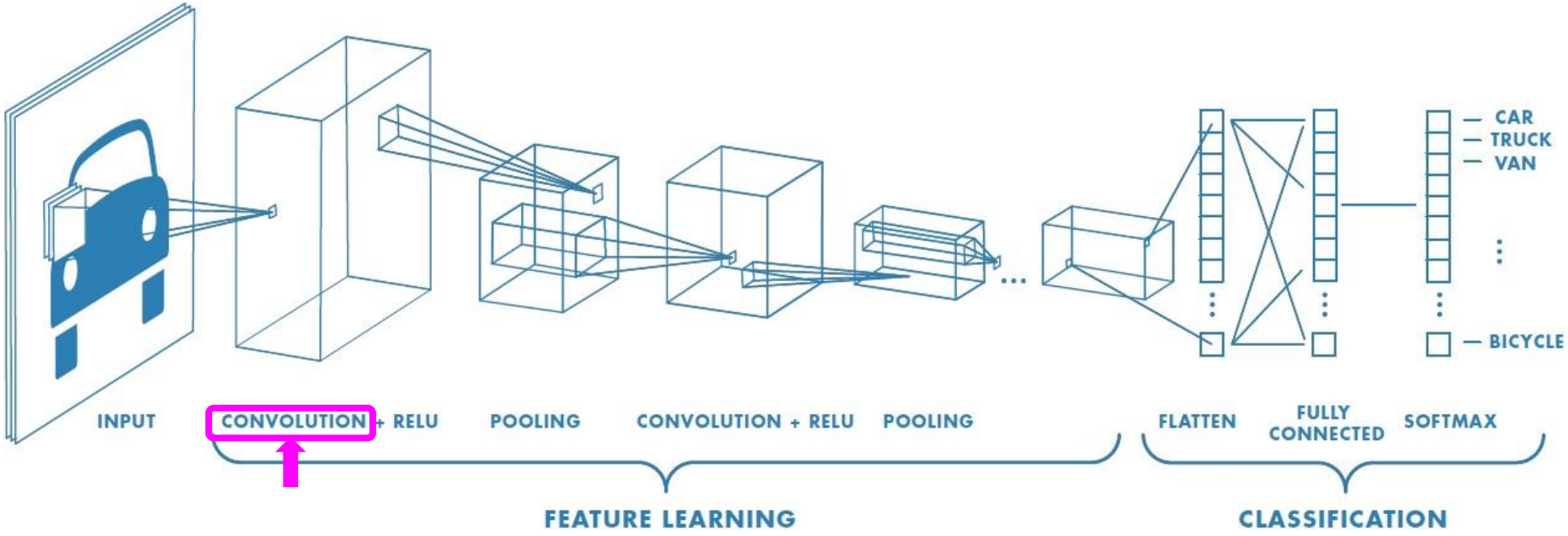
Filter weights

A generic level of CNN

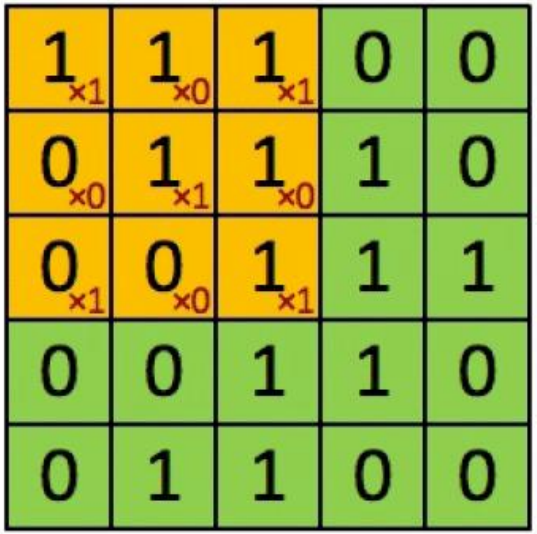


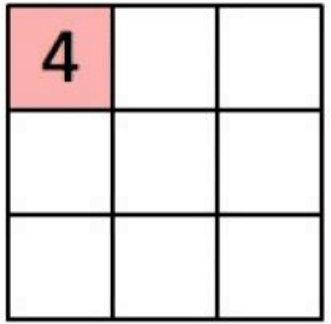
Different layers in CNN architecture (pooling, relu, etc.)

Different layers of CNN architecture



CNN: Convolutional layer

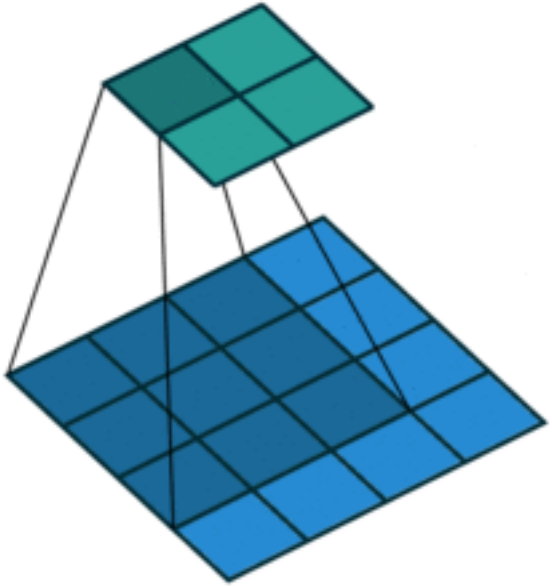




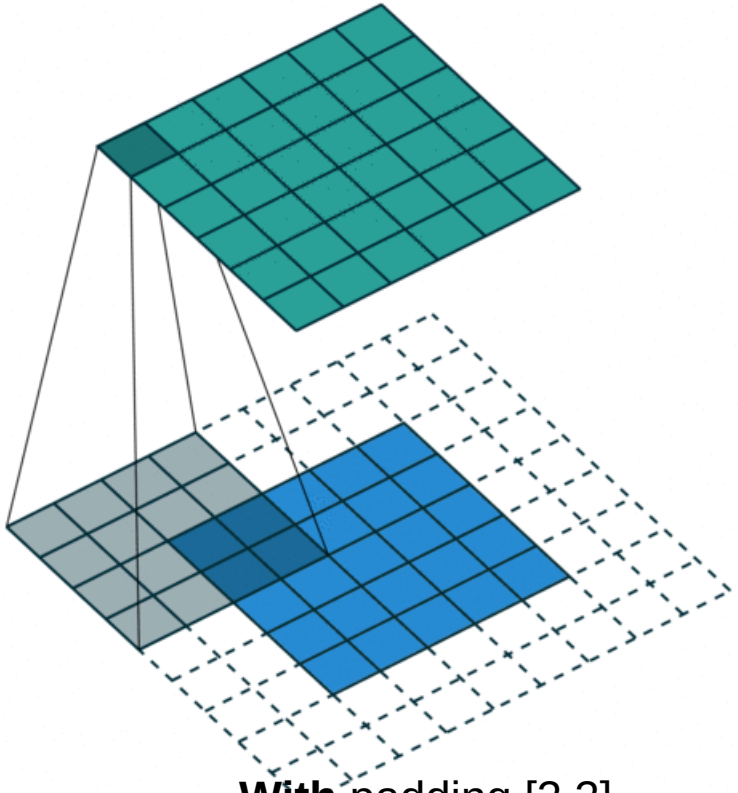
1. To reduce the number of weights (through local connectivity).
2. To provide spatial invariance (through parameter sharing).

Hyper parameters for convolutional layer.

1. Zero padding (to control input size spatially.)



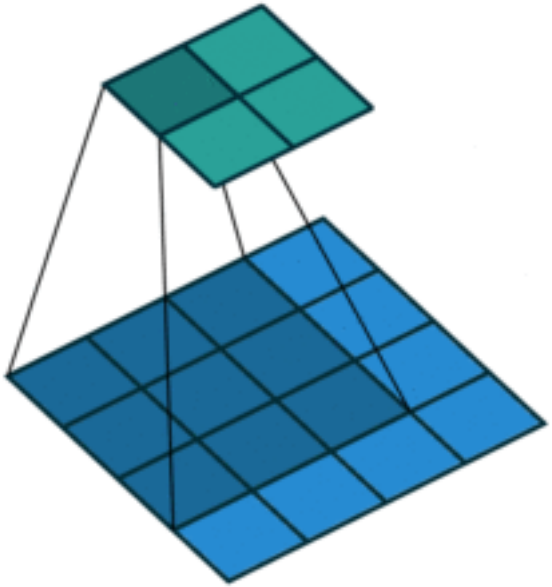
Without padding (i.e., [0,0])



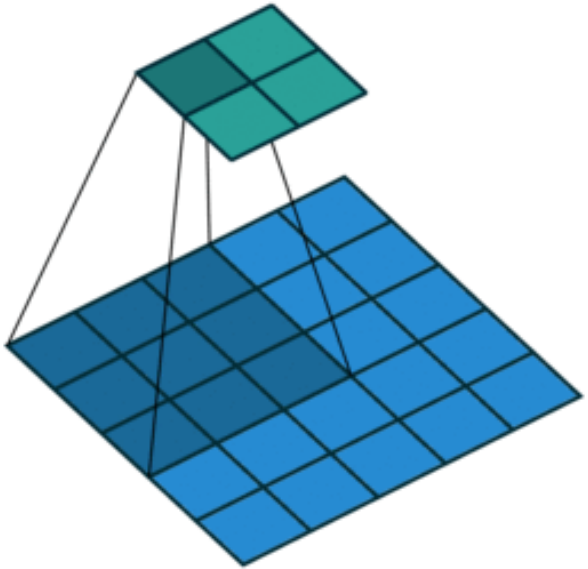
With padding [2,2]

Hyper parameters for convolutional layer.

2. Stride (to produce smaller output volumes spatially.)



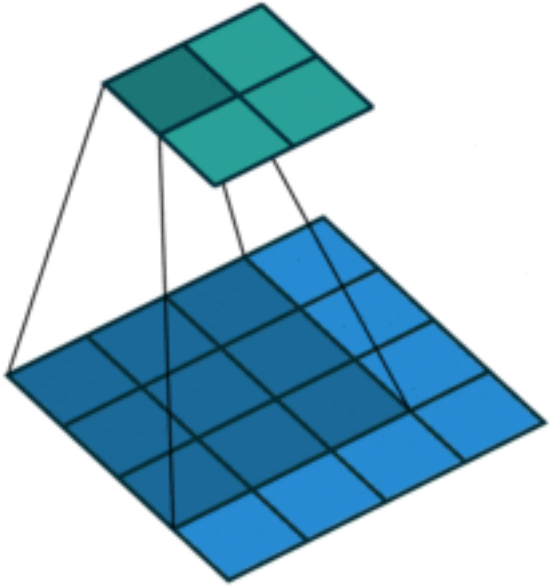
Without stride (i.e., [1,1])



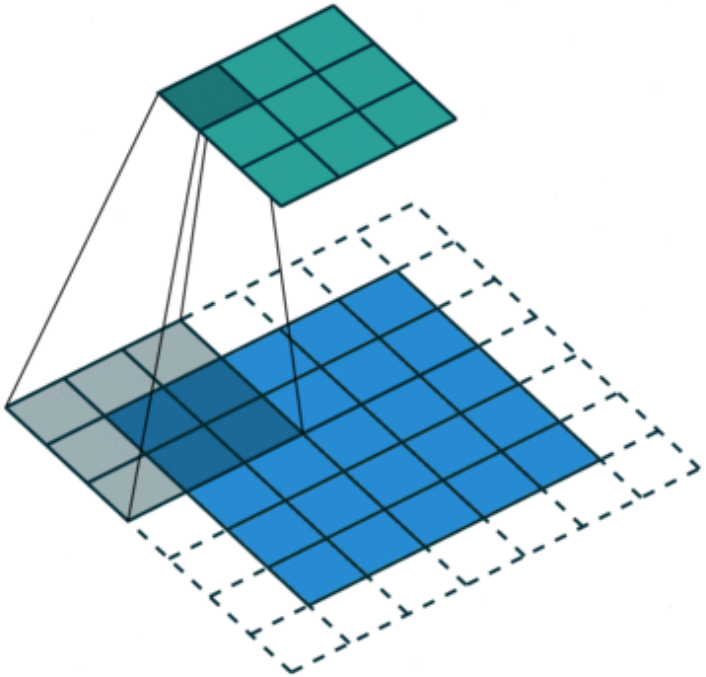
With stride [2,2]

Hyper parameters for convolutional layer.

Both padding and stride



Without padding and stride



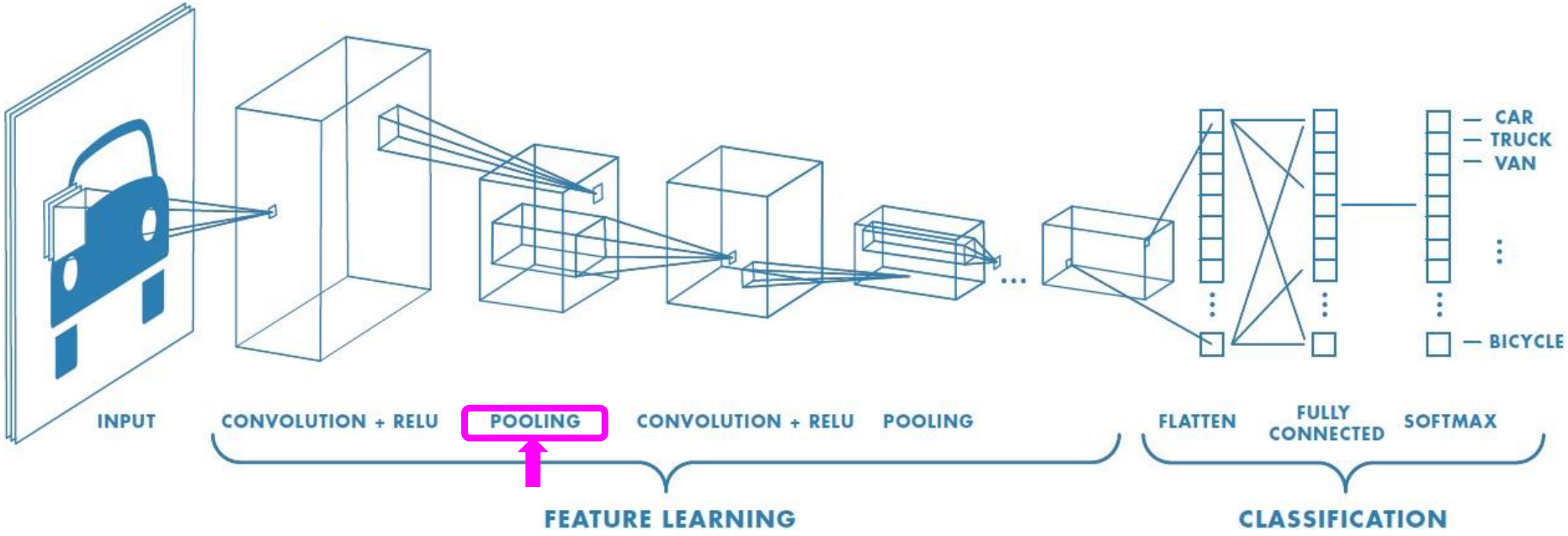
With padding [1,1] & stride [2,2]

CONVOLUTIONAL LAYER

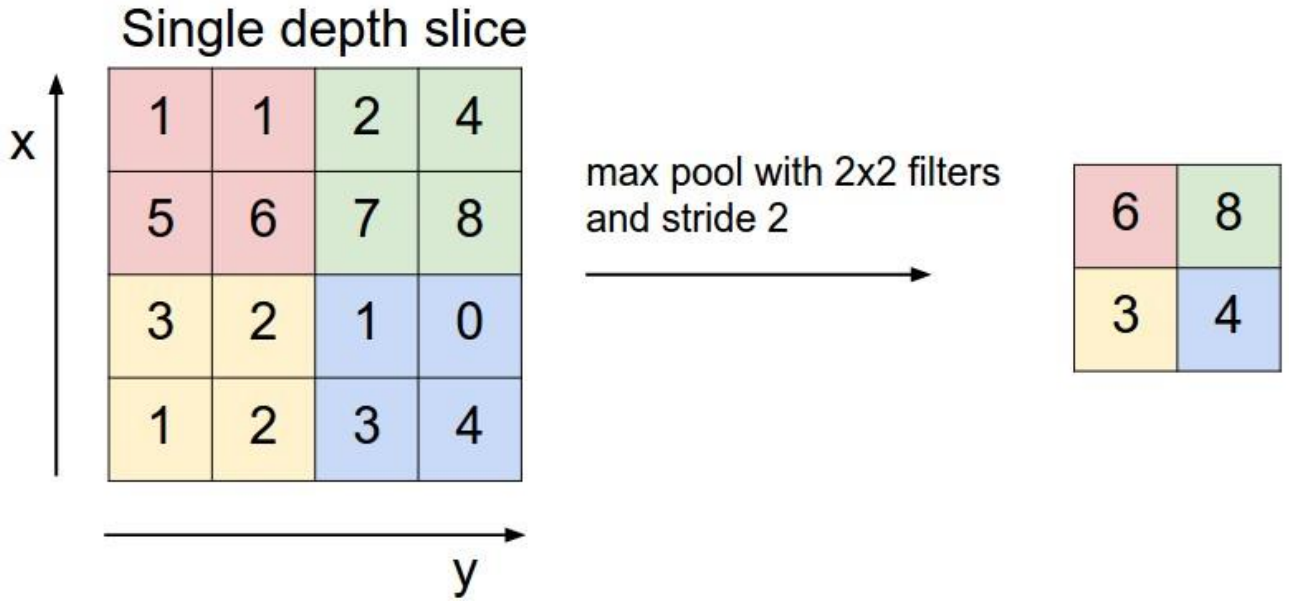
1. Accepts a volume of size ***W1*** **X** ***H1*** **X** ***D1***.
2. Requires four hyperparameters:
   a. Number of filters ***K***
   b. the spatial extent of filter ***F***
   c. their stride ***S***
   d. the amount of zero padding ***P***
3. Produces an output volume of size ***W2*** **X** ***H2*** **X** ***D2*** where:
   W2=(W1−F+2P)/S+1, H2=(H1−F+2P)/S+1, D2=K

1. With parameter sharing, it introduces ***F·F·D1*** weights per filter, for a total of ***(F·F·D1)·K*** weights and ***K*** biases.
2. In the output volume, the ***d***-th depth slice (of size ***W2*** **X** ***H2***) is the result of performing a valid convolution of the ***d***-th filter over the input volume with a stride of ***S***, and then offset by ***d***-th bias.

Different layers of CNN architecture



CNN: Pooling layer

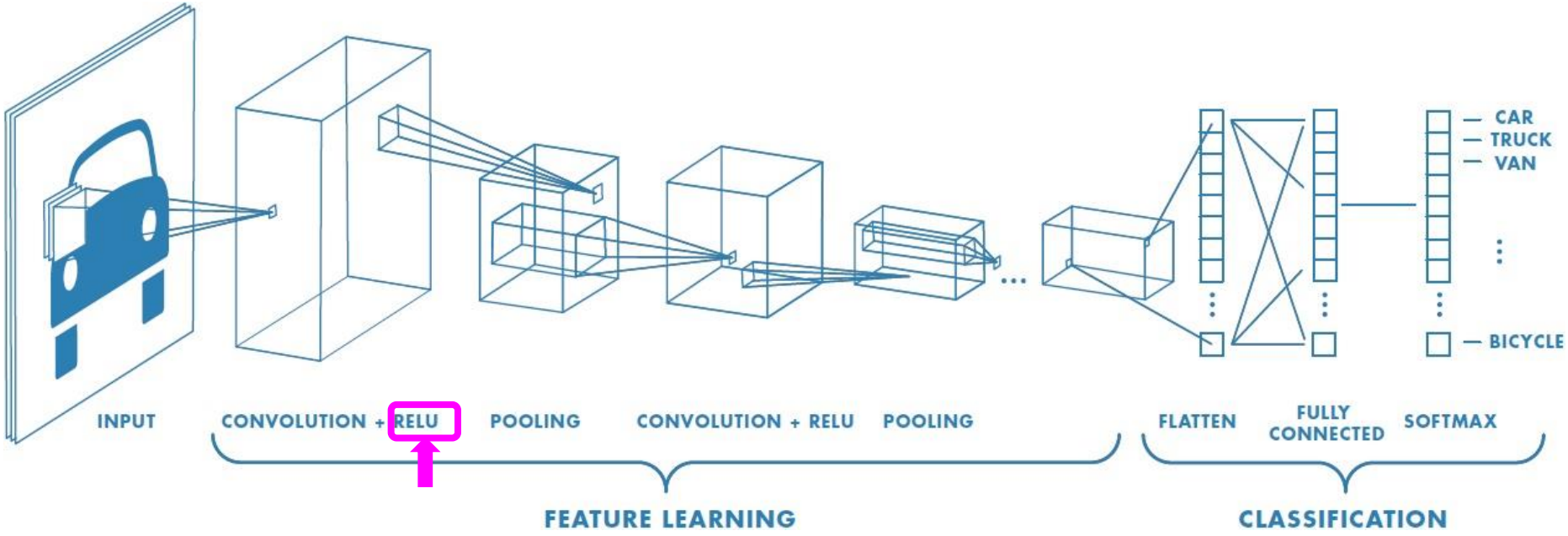


1. To reduce the spatial size of the representation to reduce the amount of parameters and computation in the network.
2. Average pooling or L2 pooling can also be used, but not popular like max pooling.

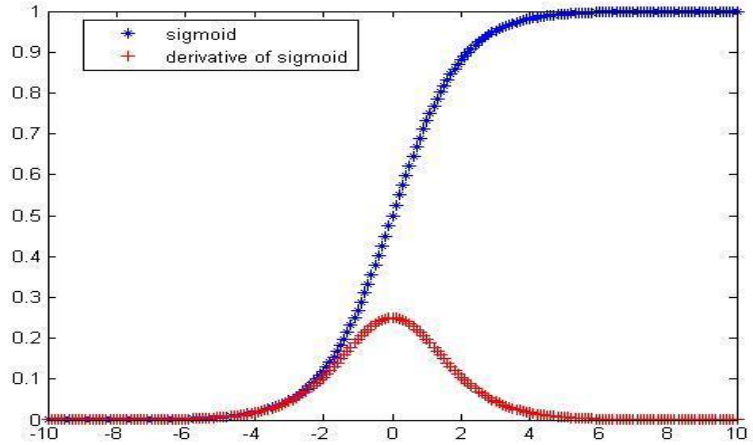
POOLING LAYER

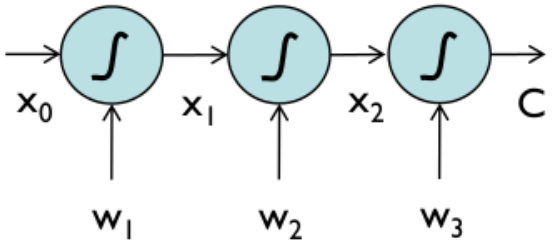
1. Accepts a volume of size ***W1*** **X** ***H1*** **X** ***D1***.
2. Requires two hyperparameters:
   a. the spatial extent of filter ***F***
   b. their stride ***S***
   c. the amount of zero padding ***P*** (commonly ***P*** = 0).
3. Produces an output volume of size ***W2*** **X** ***H2*** **X** ***D2*** where:
   W2=(W1−F+2P)/S+1, H2=(H1−F+2P)/S+1, D2=K

Different layers of CNN architecture



Activation functions: Sigmoidal function





$$x_i = \sigma'(w_i{}^T x_{i-1})$$

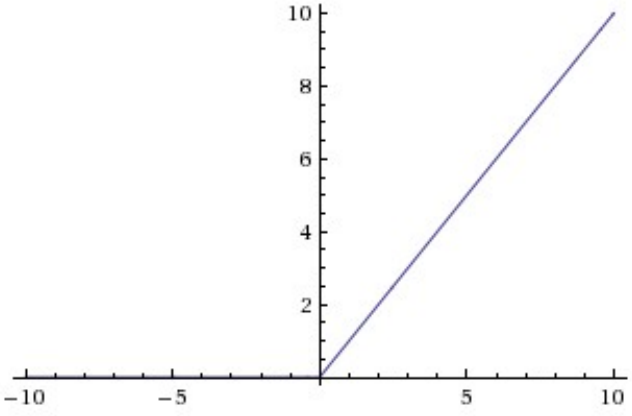
$$\frac{\partial C}{\partial w_1(j)} = \sigma'(w_3{}^T x_2) \cdot x_2(j) \cdot \sigma'(w_2{}^T x_1) \cdot x_1(j) \cdot \sigma'(w_1{}^T x_0) \cdot x_0(j)$$

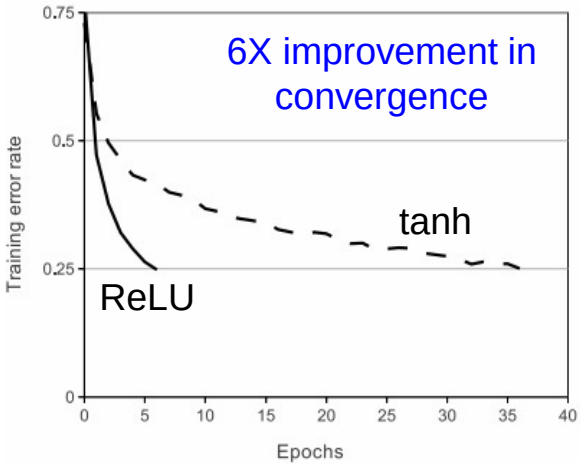
Sigmoids saturate and kill gradients (when the neuron's activation saturates at either tail of 0 or 1).

0 ⇒ fails to update weights while back-prop.

$$\frac{\partial C}{\partial w_1(i)} = \sigma'(w_3{}^T x_2) \cdot x_2(i) \cdot \sigma'(w_2{}^T x_1) \cdot x_1(i) \cdot \sigma'(w_1{}^T x_0) \cdot x_0(i)$$

Activation functions: Rectified Linear Unit (very popular).



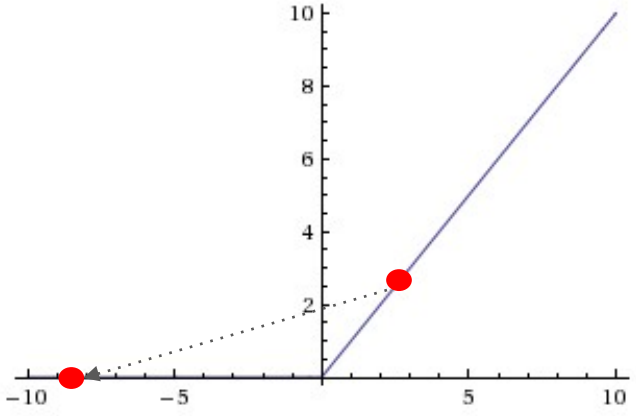


Advantage 1: **Eliminate** *saturation and killing of gradients for positive inputs*.

Advantage 2: Greatly **accelerate** convergence of SGD. (*Krizhevsky et al.* argued that this is due to its linear, non-saturating form.)

Advantage 3: tanh/sigmoid neurons involve expensive operations (exponentials, etc.), whereas ReLU can be implemented by **simply thresholding** activations at zero.

Activation functions: Rectified Linear Unit (very popular).



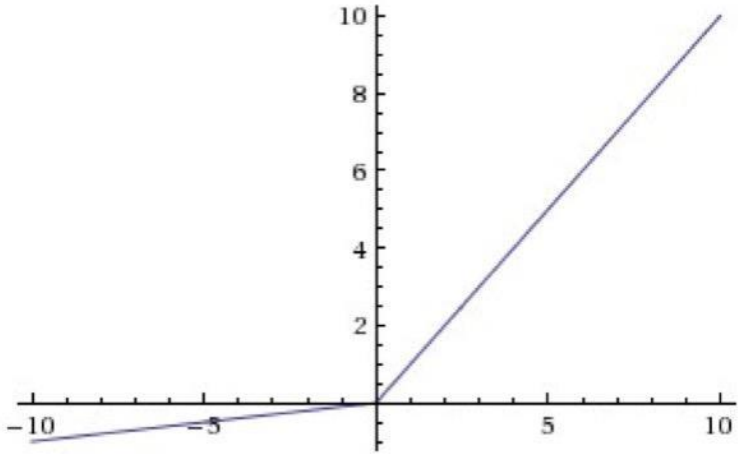
Drawback: ReLU units can **irreversibly die** during training. A large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again.

0 ⇒ fails to update weights while back-prop.

$$\frac{\partial C}{\partial w_1(i)} = \sigma'({w_3}^T x_2) \cdot x_2(i) \cdot \sigma'({w_2}^T x_1) \cdot x_1(i) \cdot \sigma'({w_1}^T x_0) \cdot x_0(i)$$

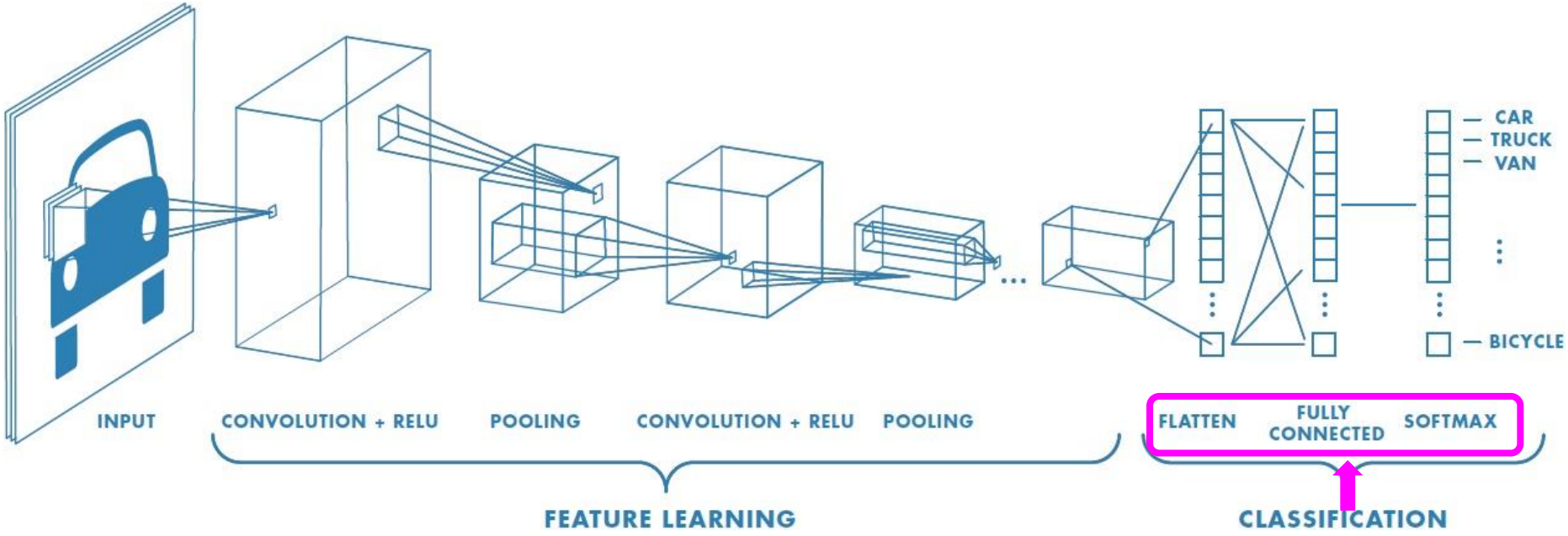
Activation functions: Leaky ReLU.

Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (a hyper-parameter).

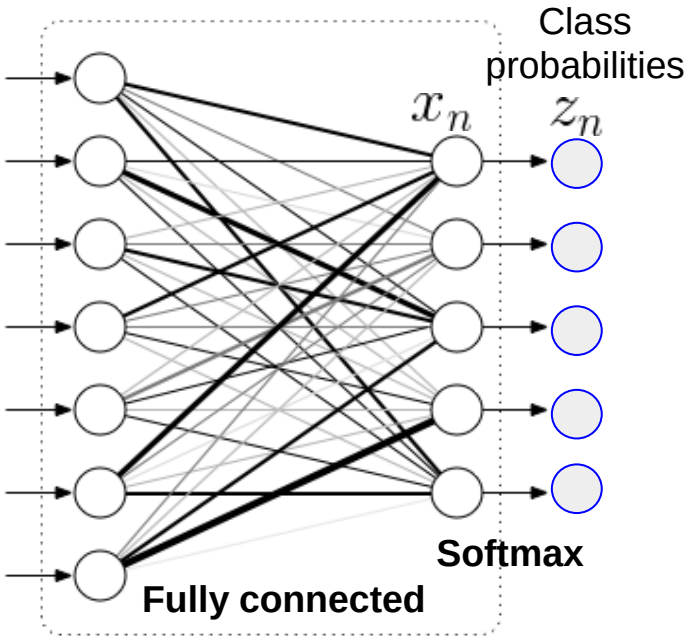


Advantages: **Eliminate** irreversible dying of neurons, as in ReLU.
Have all the advantages of ReLU.

Different layers of CNN architecture



Flattening, fully connected (FC) layer and softmax



Flattening

1. Vectorization (converting ***M X N X D*** tensor to a ***MND X 1*** vector).

FC layer

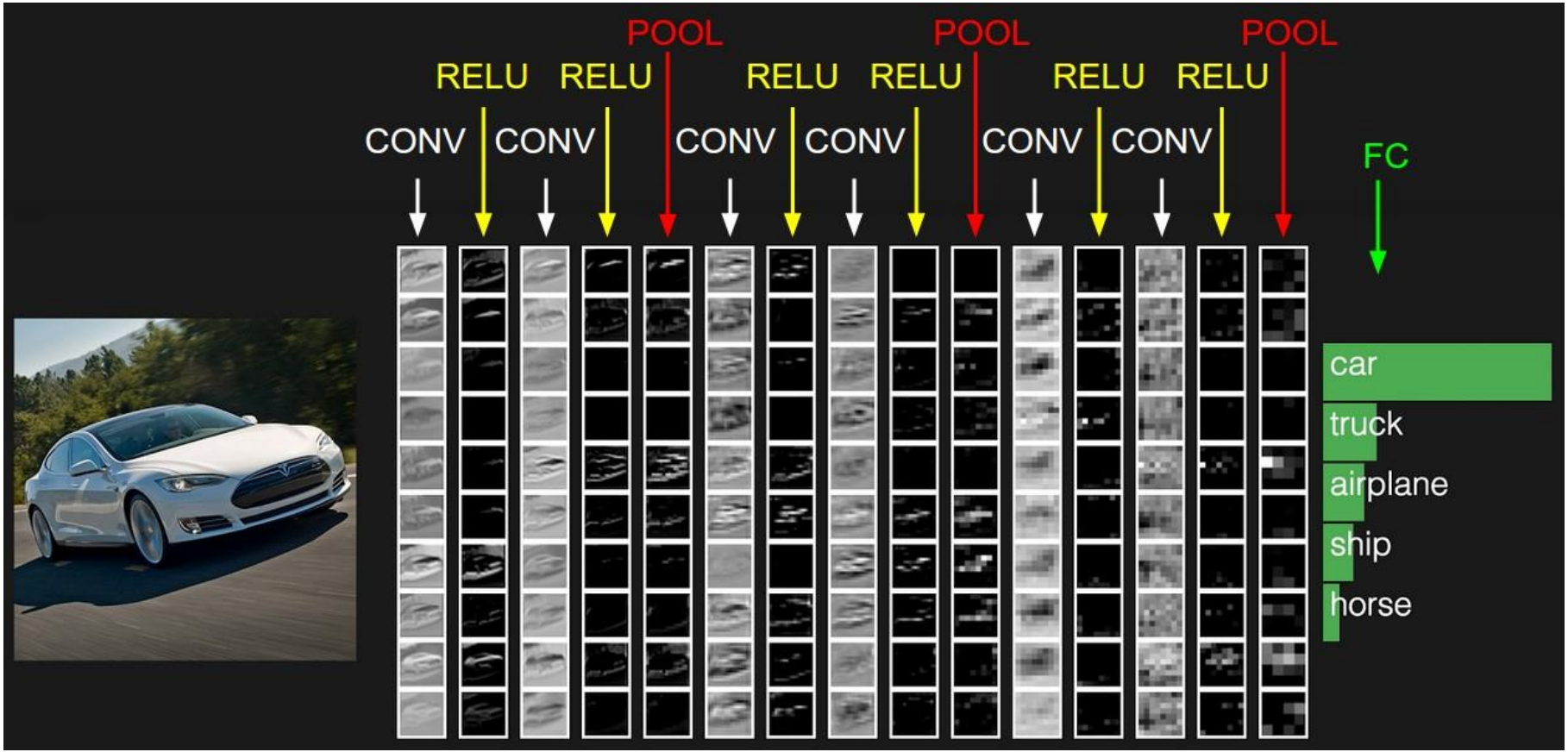
1. Multilayer perceptron.
2. Generally used in final layers to classify the object.
3. Role of a classifier.

Softmax layer

1. Normalize output as discrete class probabilities.

$$z_n = \frac{e^{x_n}}{\sum_{i=1}^{K} e^{x_i}}$$

A CNN Example



Basics of CNN:

- Advantage of CNN over MLP
  - Local connectivity
  - Parameter sharing
  - Efficient runtime and storage

- Different layers in CNN
  - Convolutional layer
  - ReLU
  - Pooling
  - Fully connected layer

Convolutional Neural Networks (CNNs)

Introduction

- CNNs over fully connected
- Different layers in CNN
  - Convolutional layer
  - RelU
  - Pooling layer
  - Fully connected layer

State of the art architectures

- AlexNet
- VGG Net
- GoogLeNet (inception module)
- ResNets (Residual module)
  CVPR’ 17 best paper

CNN Architectures

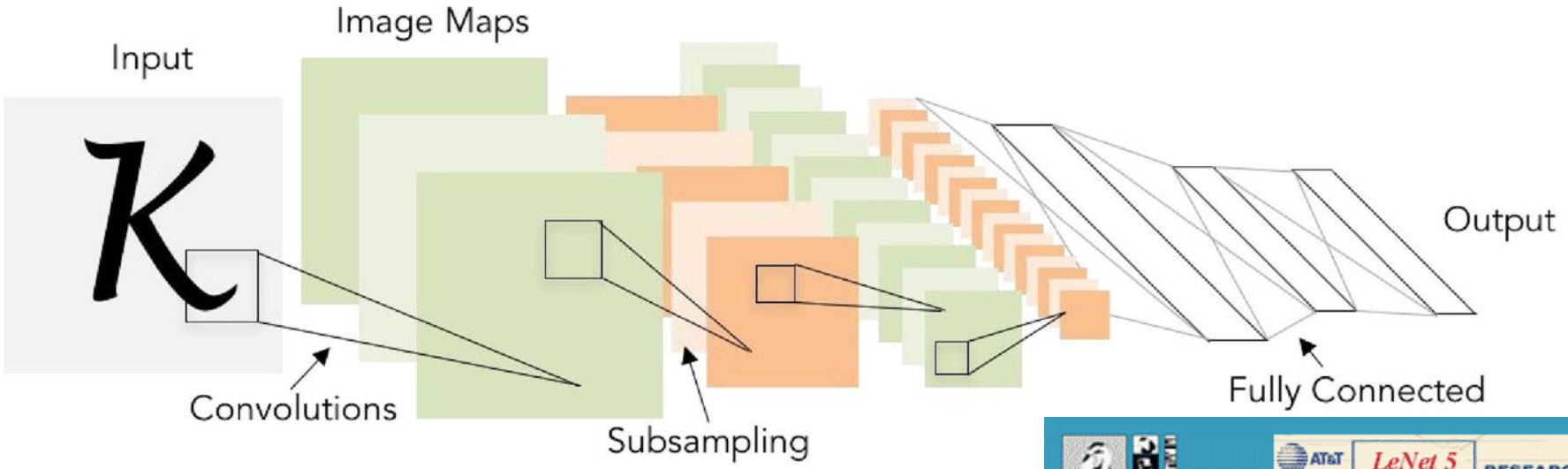
Slides taken from CS231n lecture slides

Case study of following CNN architectures

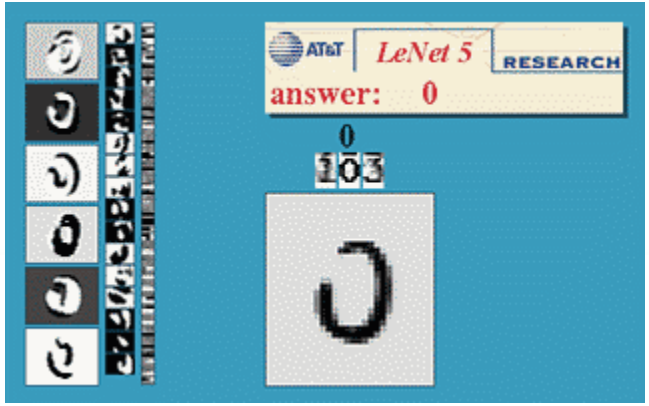
- LeNet
- AlexNet
- VGG
- GoogLeNet
- ResNet

Review: LeNet-5

[LeCun et al., 1998]



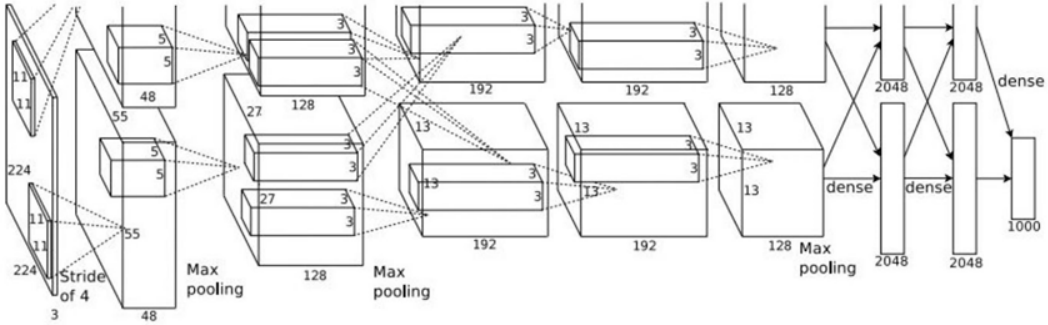
Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]



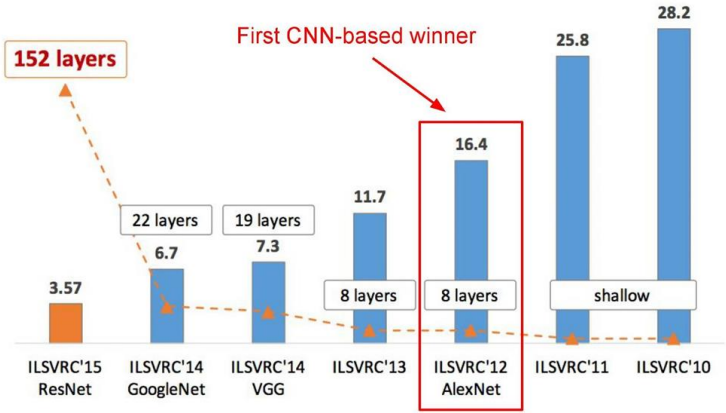
Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

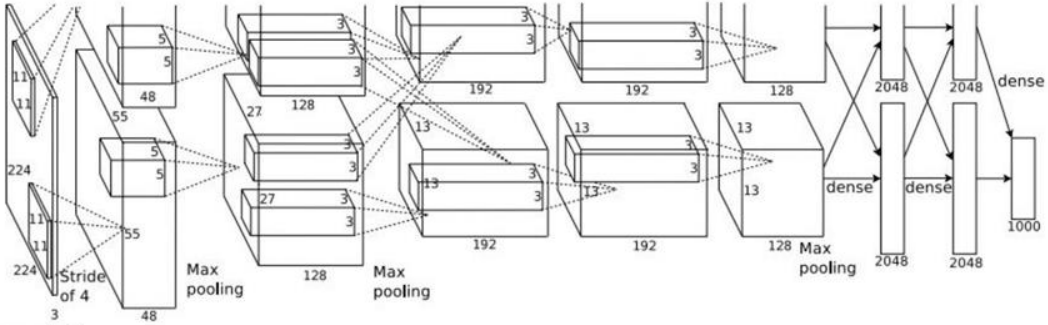


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: AlexNet

[Krizhevsky et al. 2012]

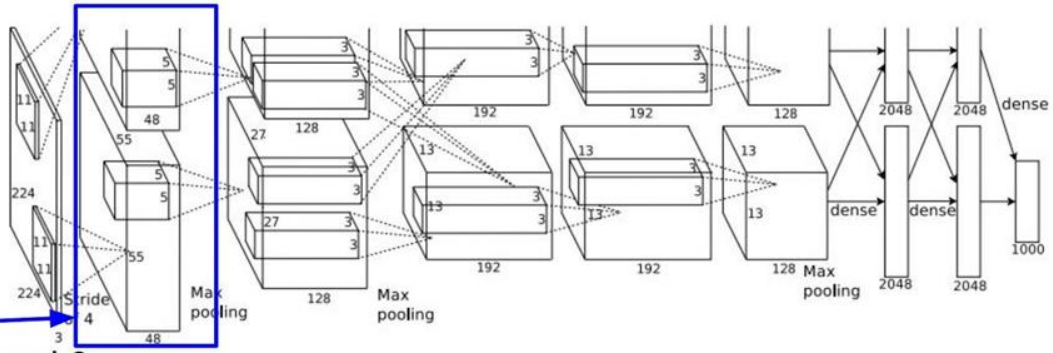


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Case Study: AlexNet

[Krizhevsky et al. 2012]



[55x55x48] x 2

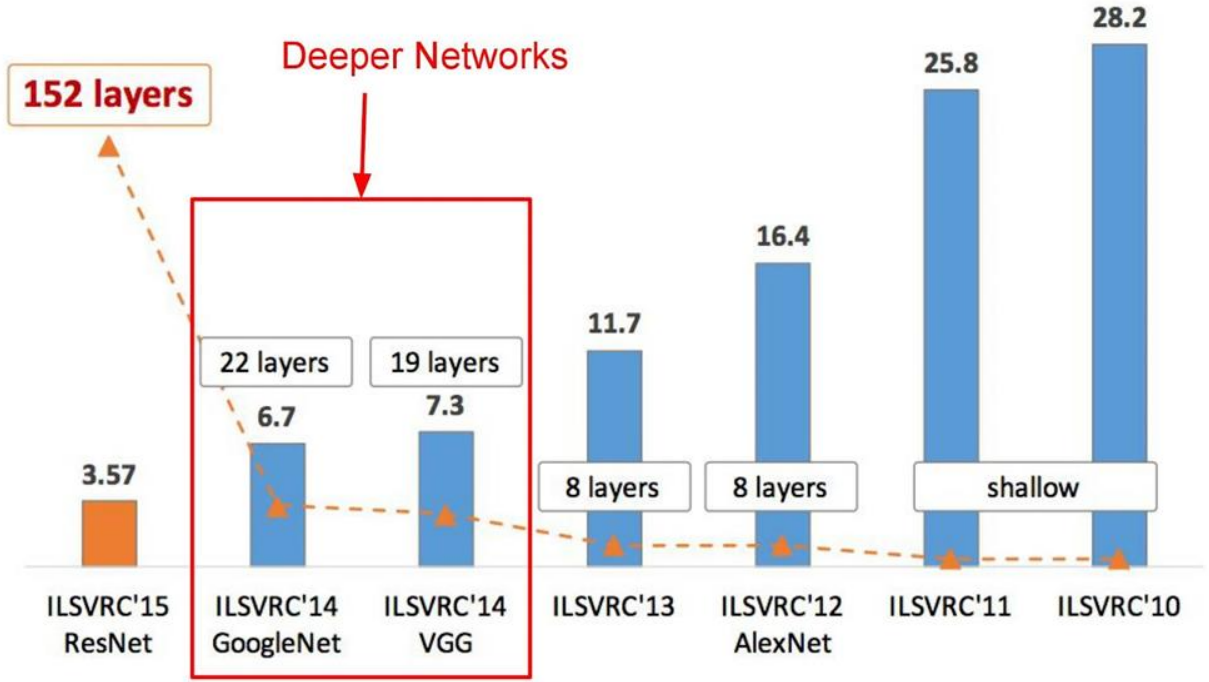
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

Note on Alexnet

- Notice the use of filters of size 11x11 in the initial layer.
- It increases the number of parameters that need to be trained
- Recent CNNs use cascade of small filters of size 3*3 or 5*5.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: VGGNet

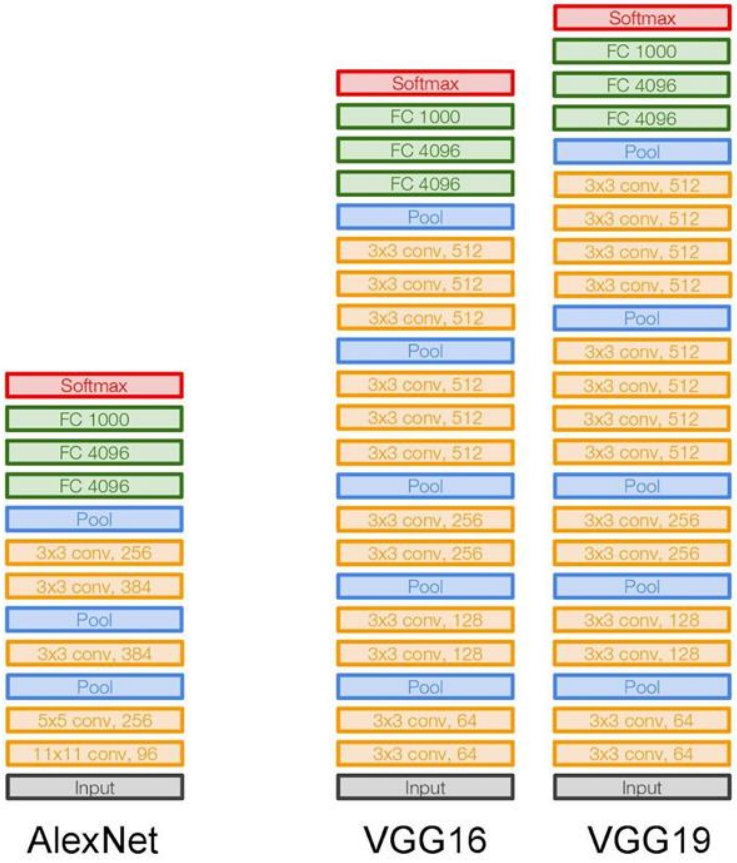
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

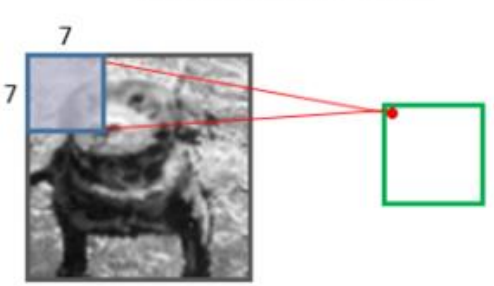
Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: 3 * (3^2C^2) vs. 7^2C^2 for C channels per layer

« Conventional » approach

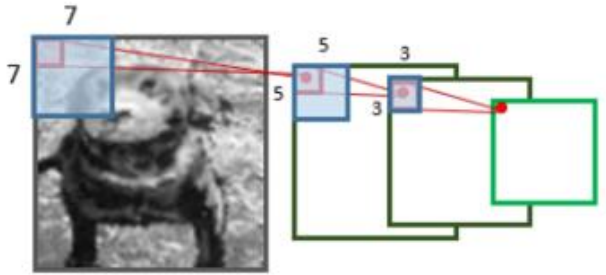
Using one convolutional layer with a large receptive field (7x7)



Effective receptive field

« VGG net » approach

Stacking three (3x3) convolutional layers



Convolution filter

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

Note:

Most memory is in early CONV

Most params are in late FC

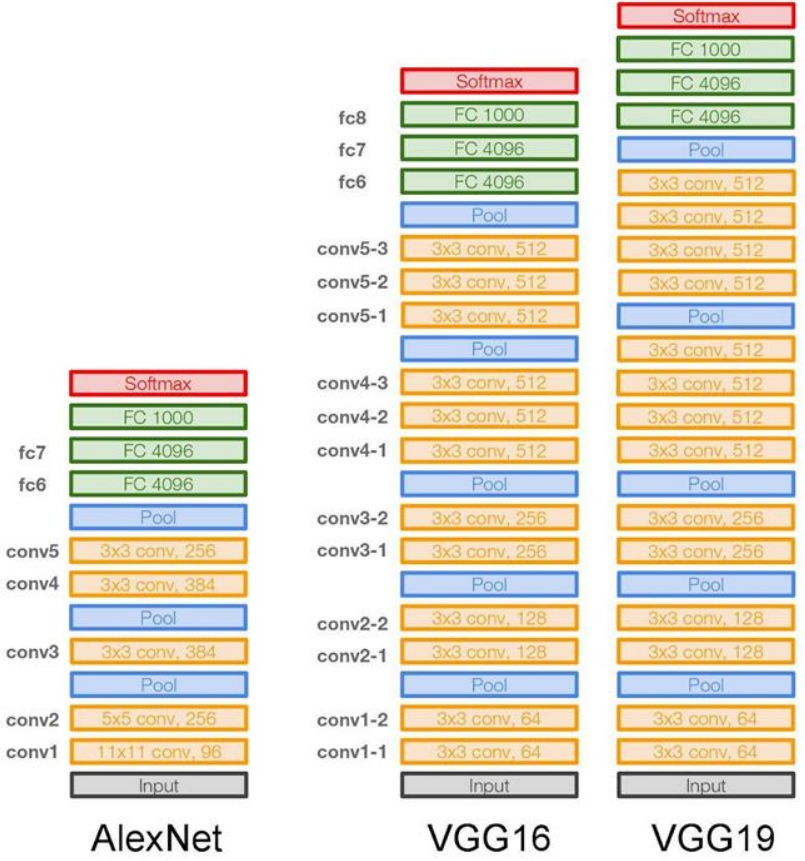
TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Case Study: VGGNet

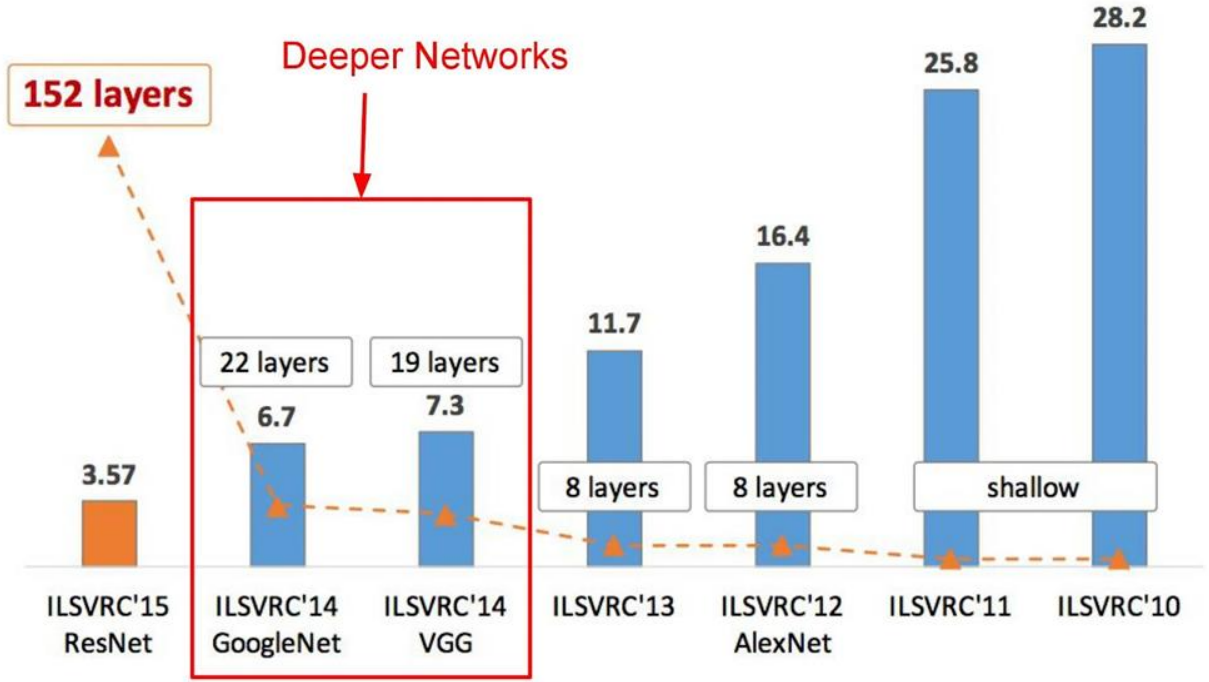
[Simonyan and Zisserman, 2014]

Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

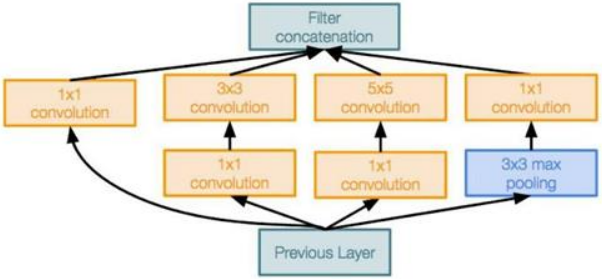


Case Study: GoogLeNet

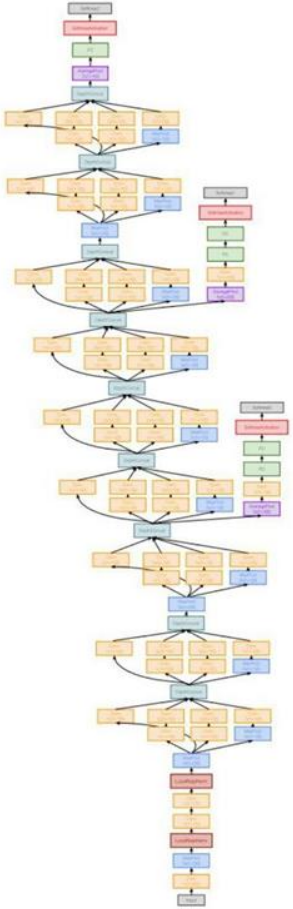
[Szegedy et al., 2014]

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)



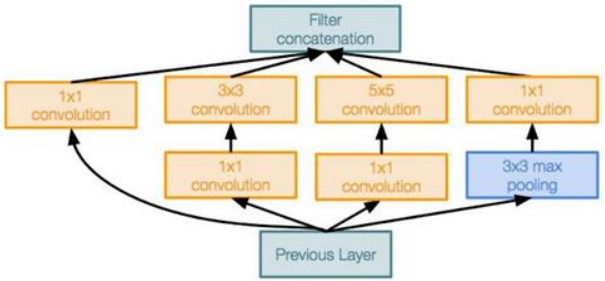
Inception module



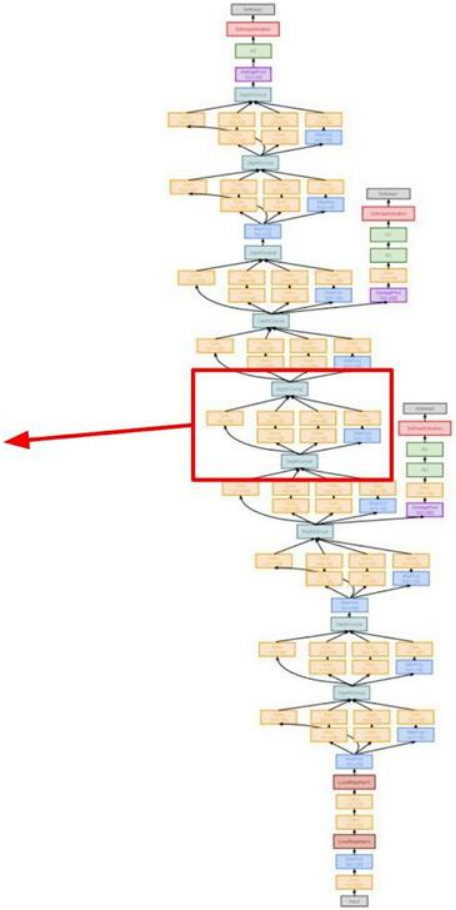
Case Study: GoogLeNet

[Szegedy et al., 2014]

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

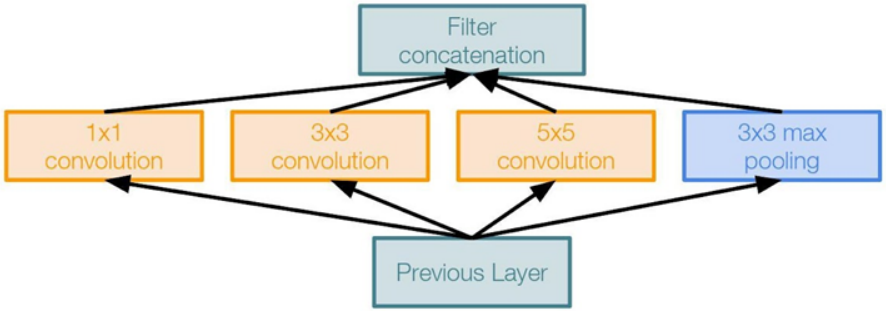


Inception module



Case Study: GoogLeNet

[Szegedy et al., 2014]



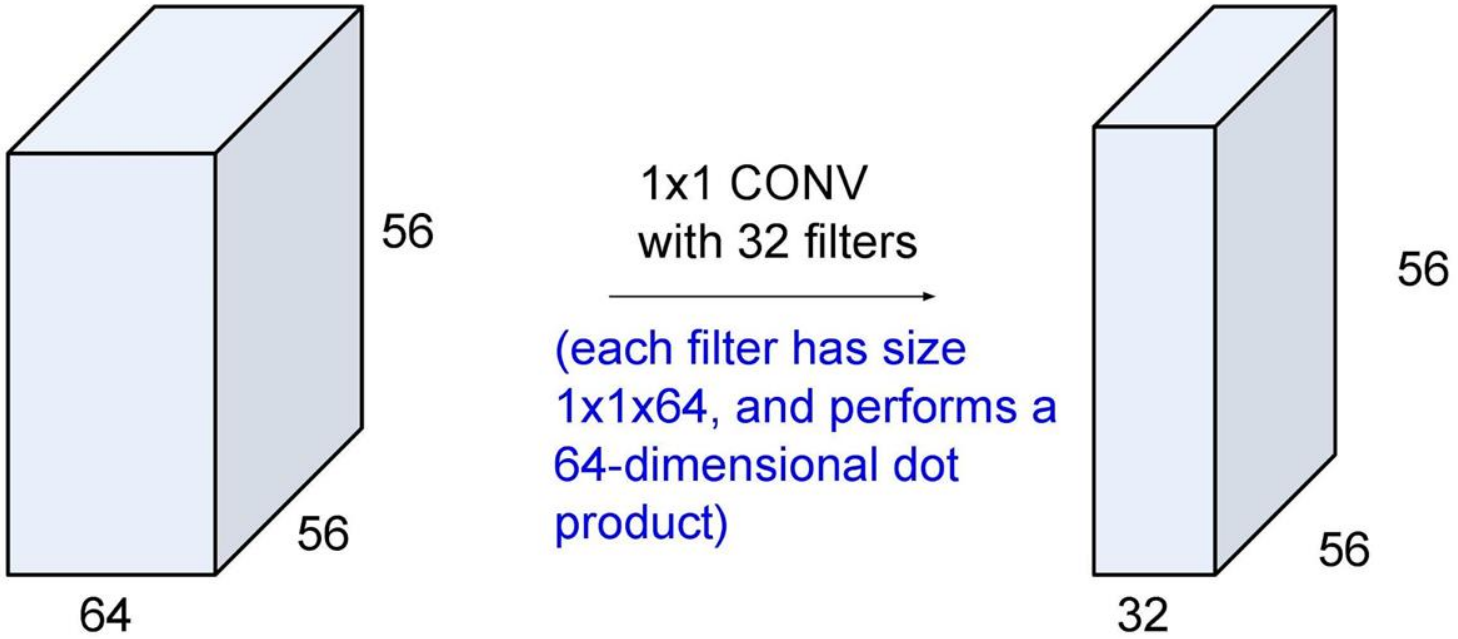
Naive Inception module

Apply parallel filter operations on the input from previous layer:

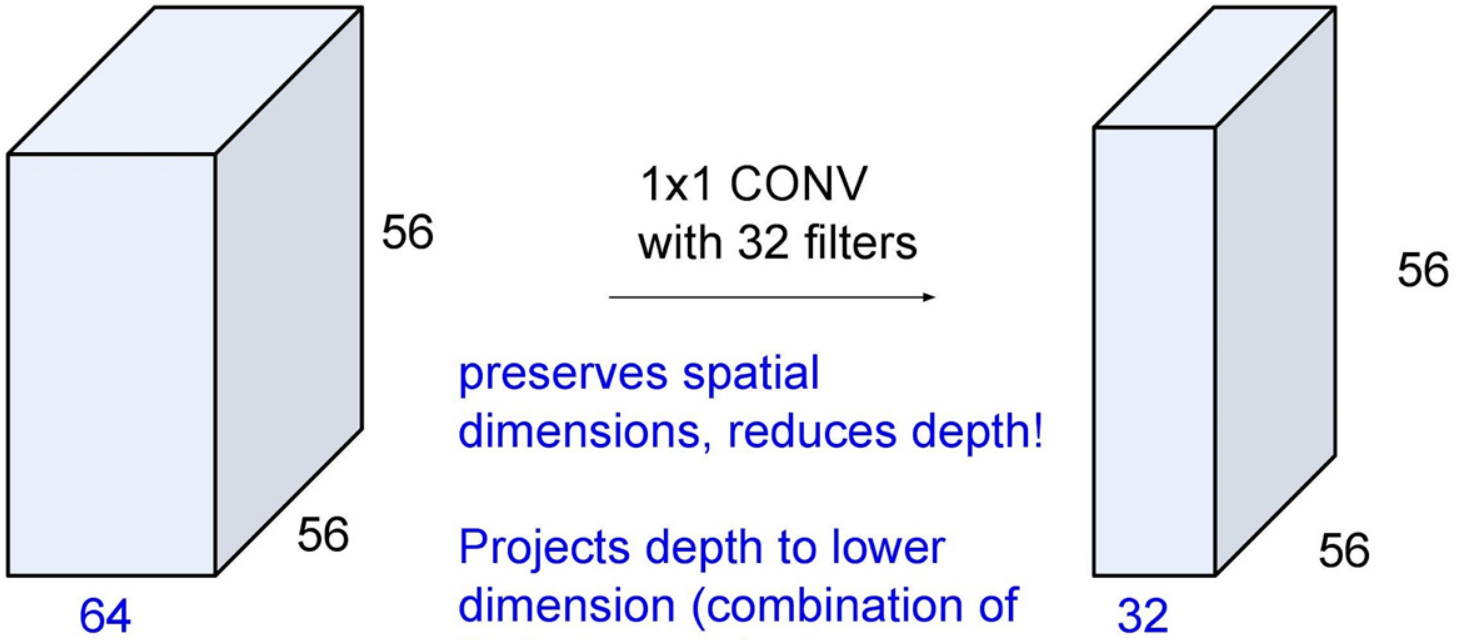
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

1x1 Convolutions

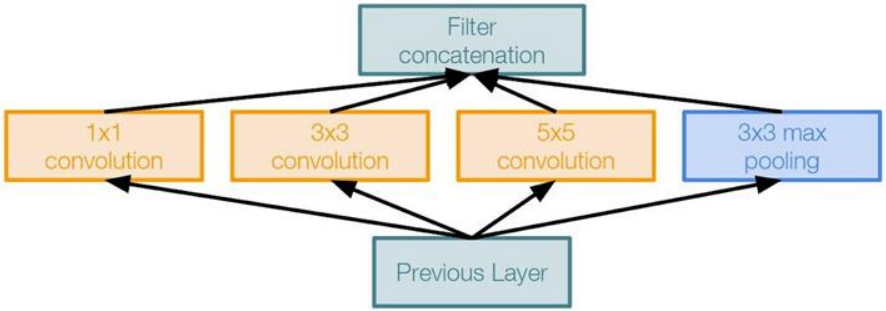


1x1 Convolutions



Case Study: GoogLeNet

[Szegedy et al., 2014]



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

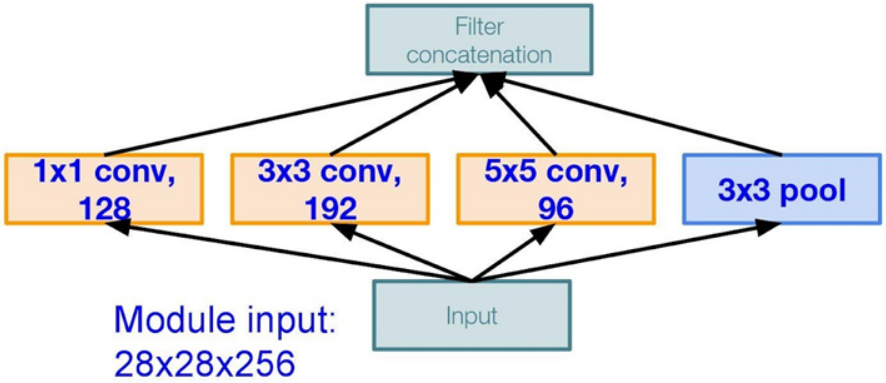
Q: What is the problem with this?
[Hint: Computational complexity]

Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:



Naive Inception module

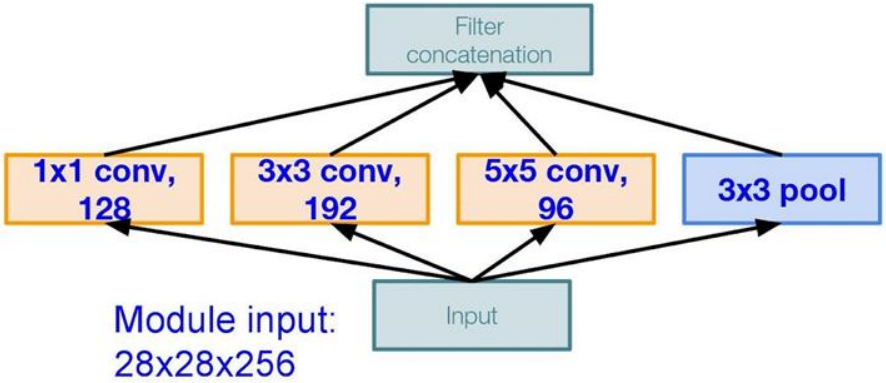
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?



Naive Inception module

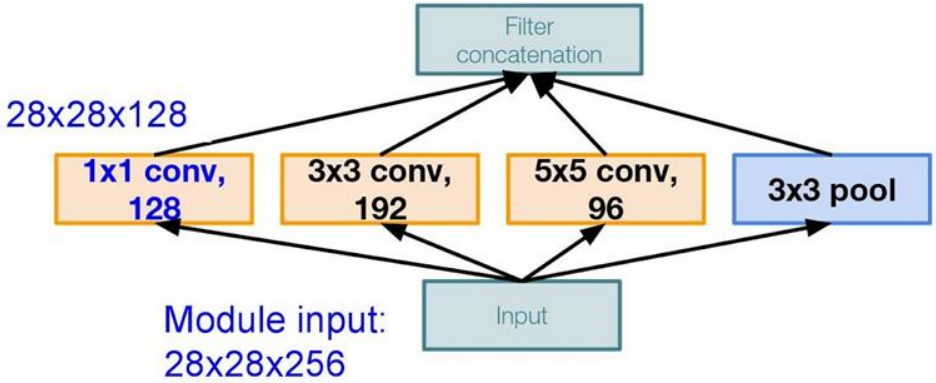
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?



Naive Inception module

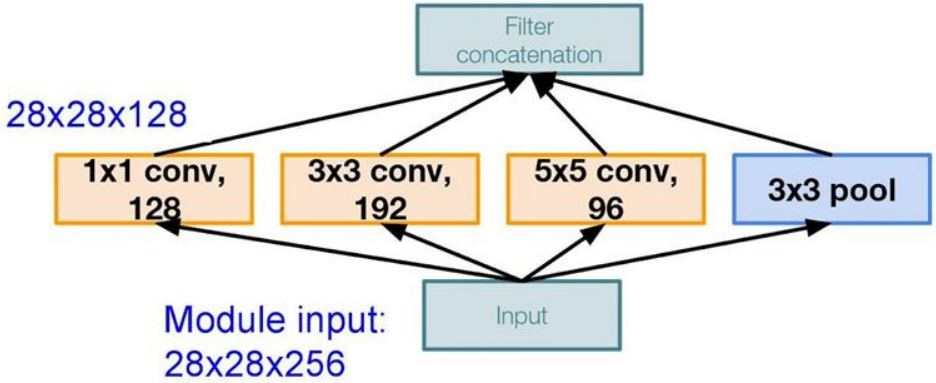
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?



Naive Inception module

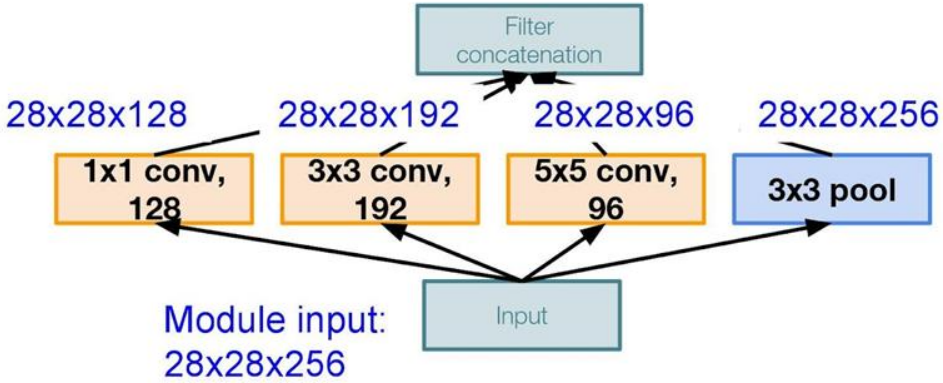
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?



Naive Inception module

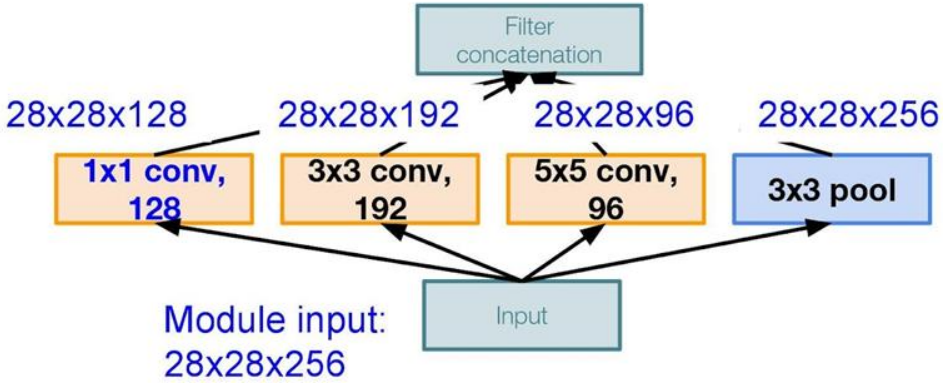
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

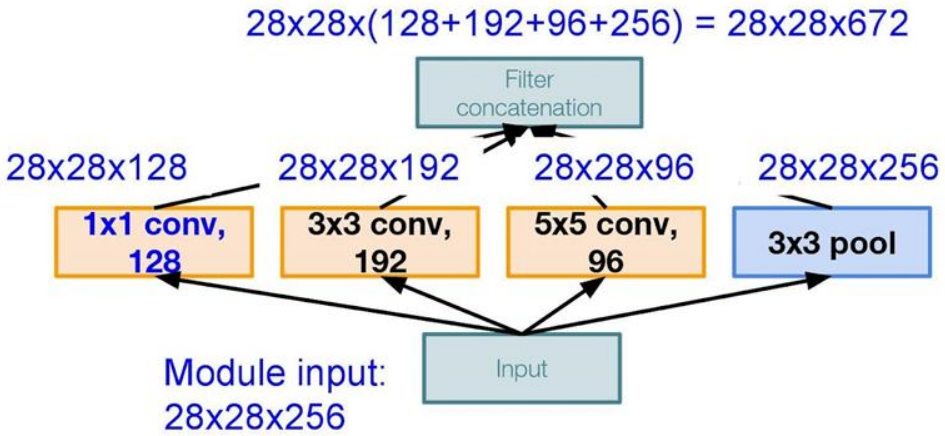
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?



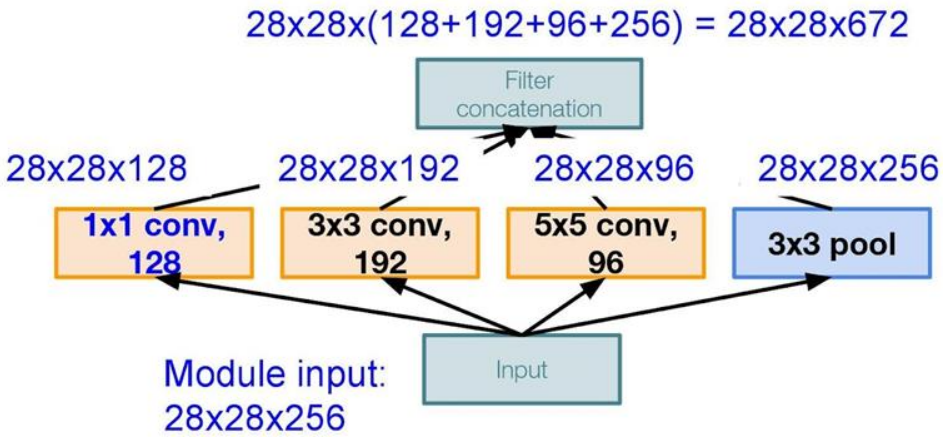
Naive Inception module

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

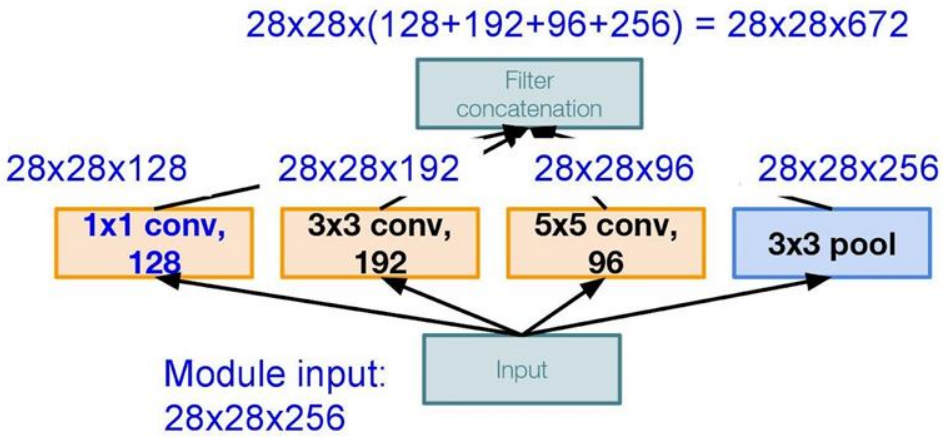
Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

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

Very expensive compute

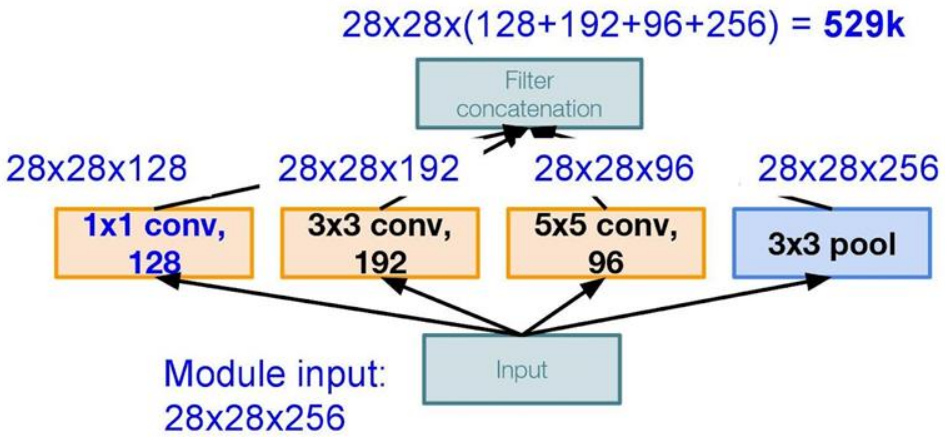
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



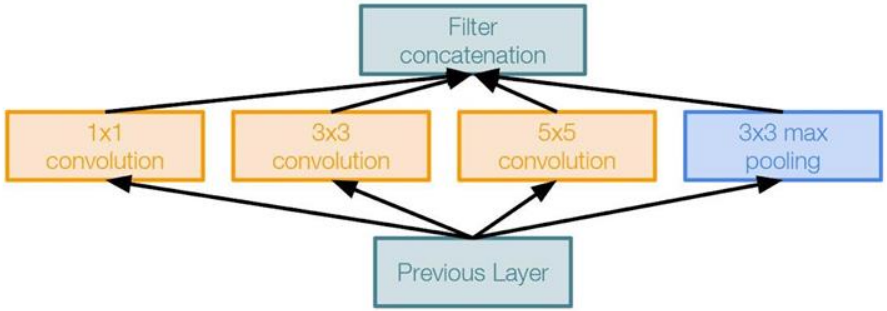
Naive Inception module

Q: What is the problem with this? [Hint: Computational complexity]

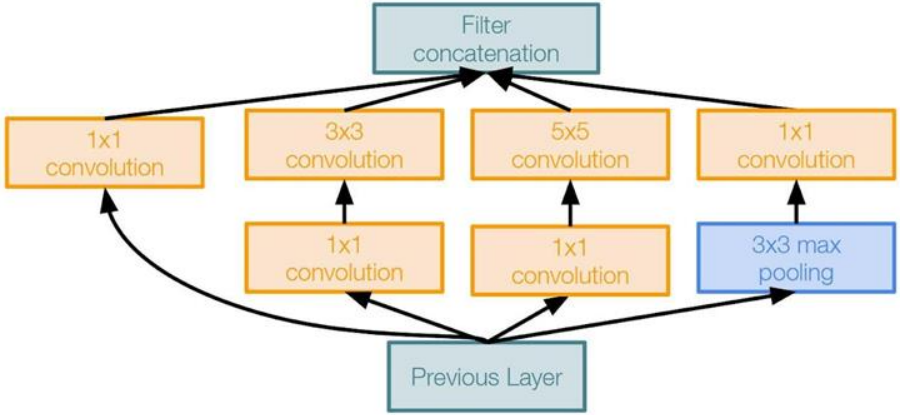
Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature depth

Case Study: GoogLeNet

[Szegedy et al., 2014]



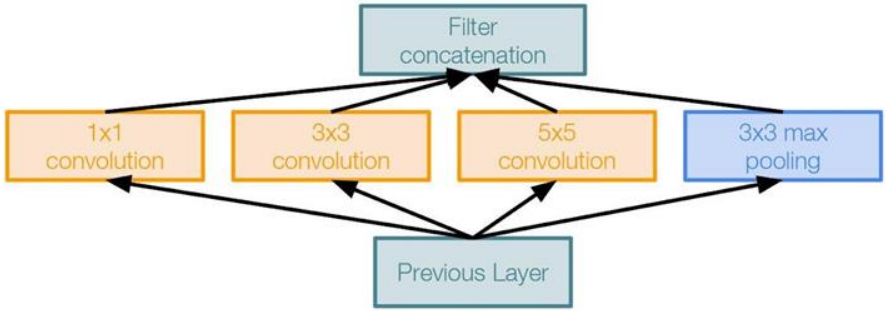
Naive Inception module



Inception module with dimension reduction

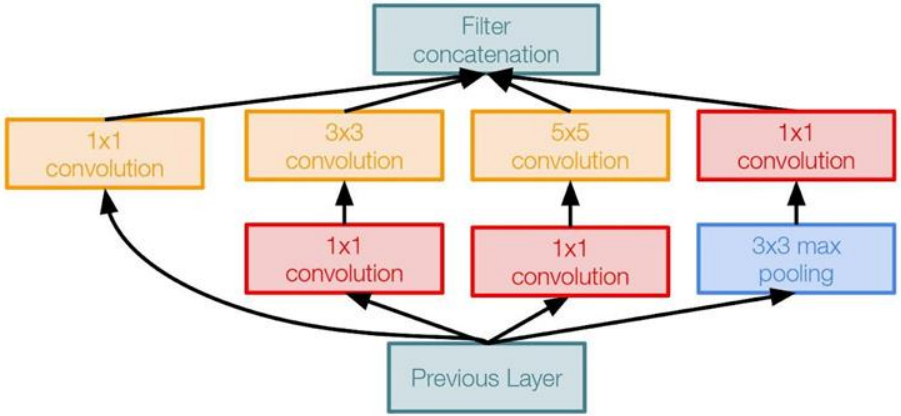
Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

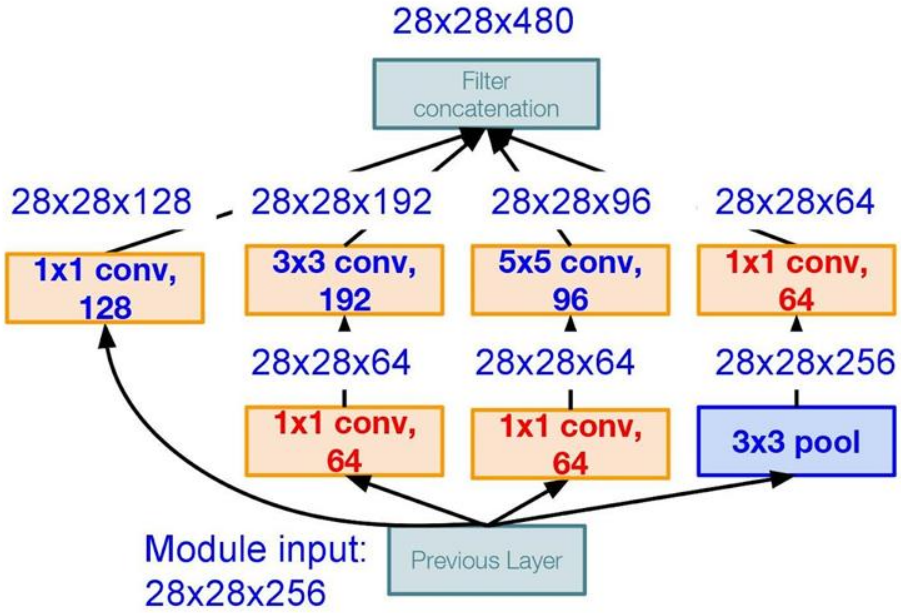
1x1 conv "bottleneck" layers



Inception module with dimension reduction

Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

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

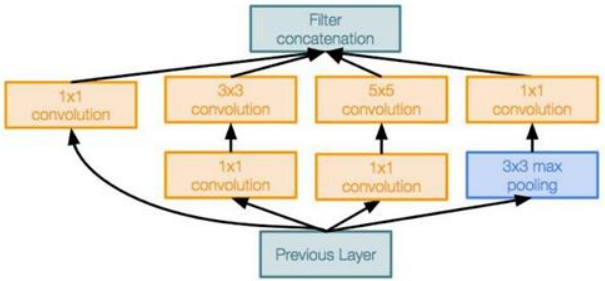
Conv Ops:
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

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

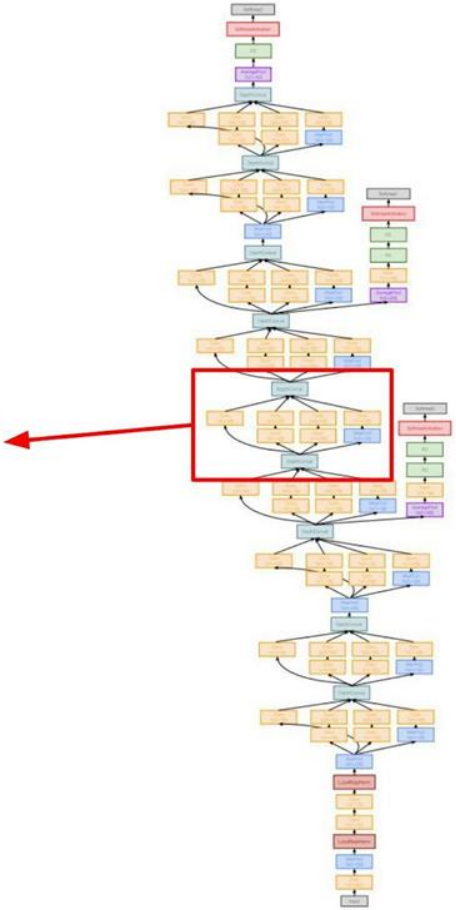
Case Study: GoogLeNet

[Szegedy et al., 2014]

Stack Inception modules with dimension reduction on top of each other



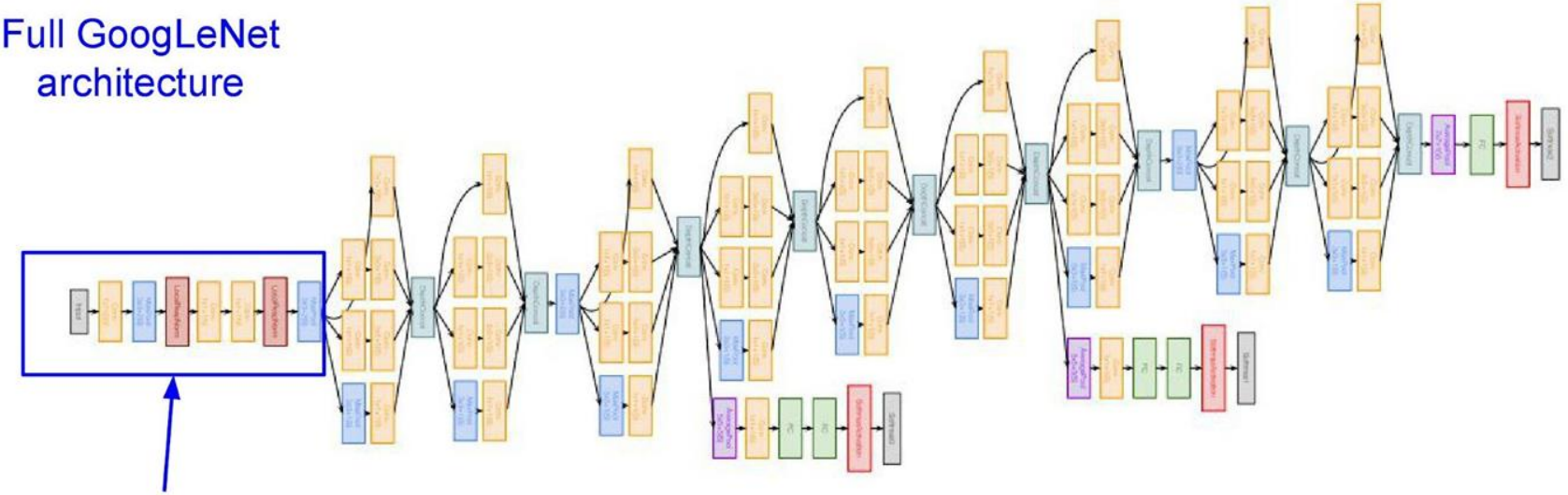
Inception module



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

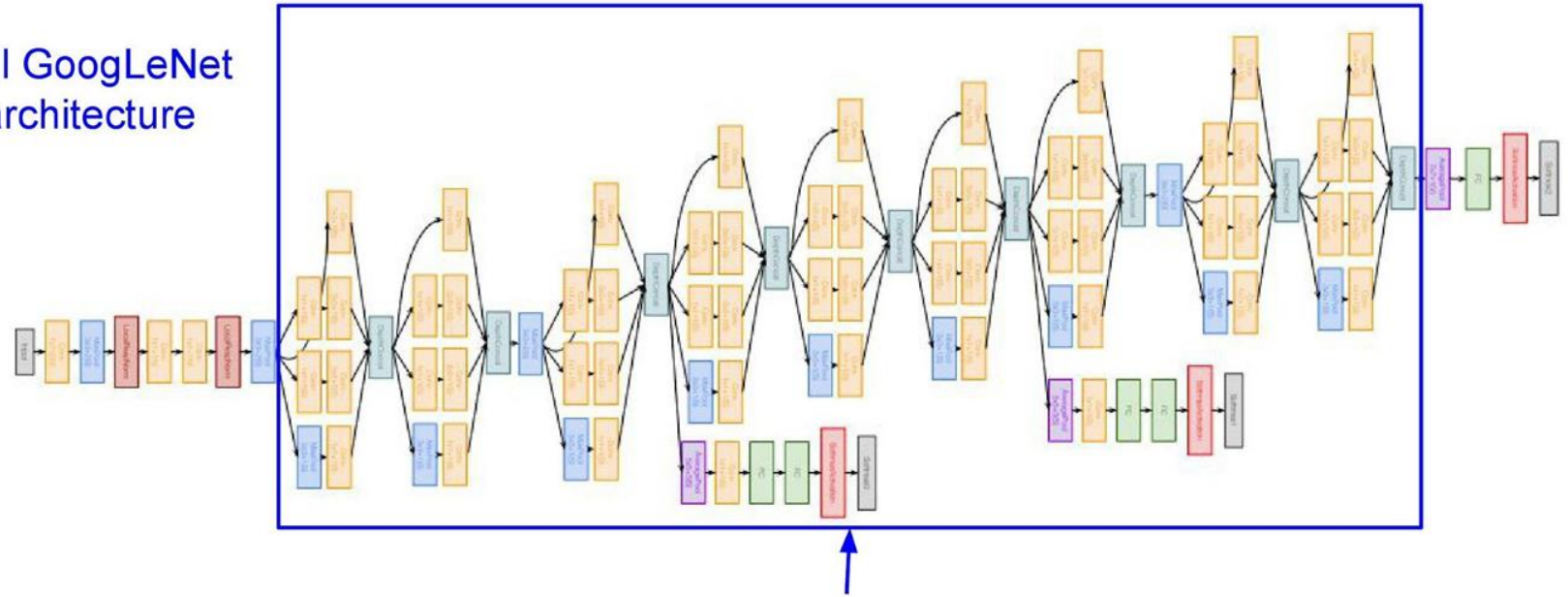


Stem Network: Conv-Pool- 2x Conv-Pool

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

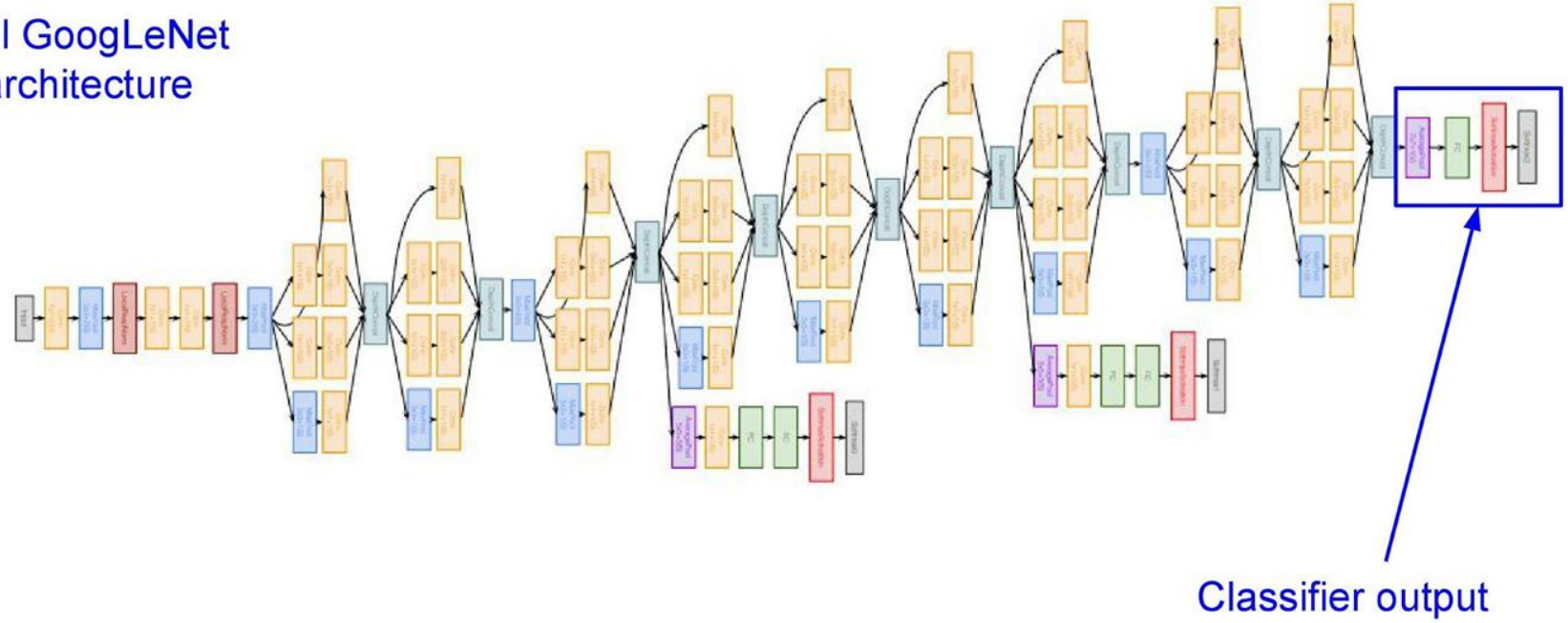


Stacked Inception Modules

Case Study: GoogLeNet

[Szegedy et al., 2014]

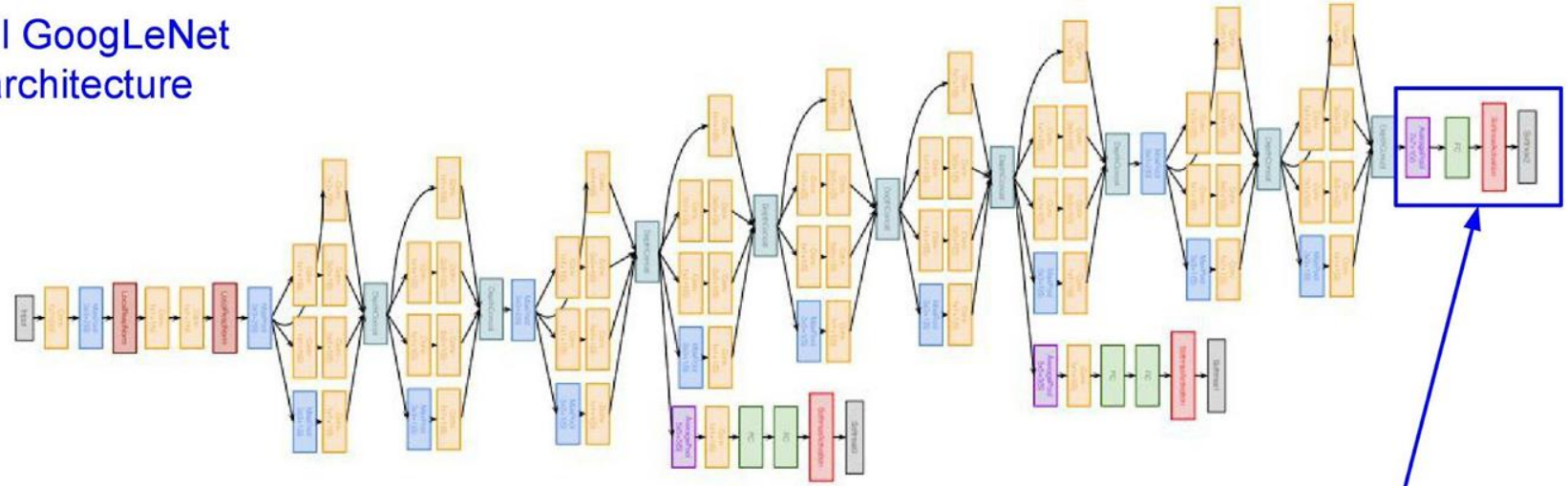
Full GoogLeNet architecture



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

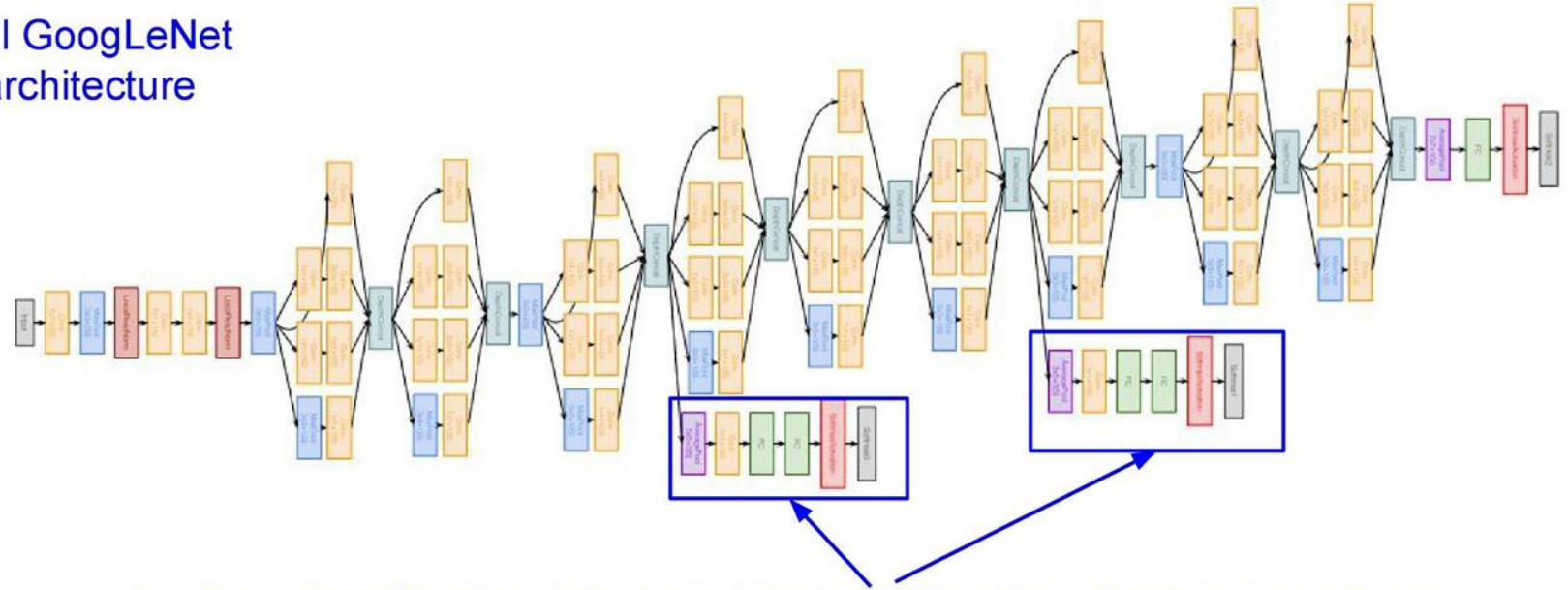


Classifier output
(removed expensive FC layers!)

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

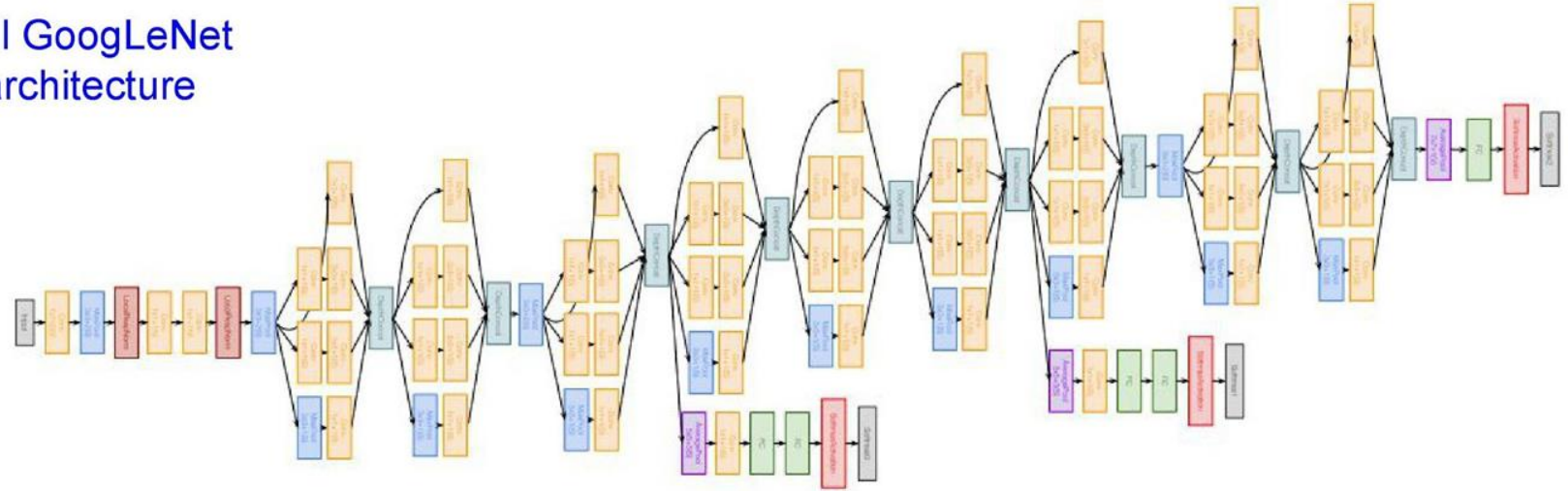


Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



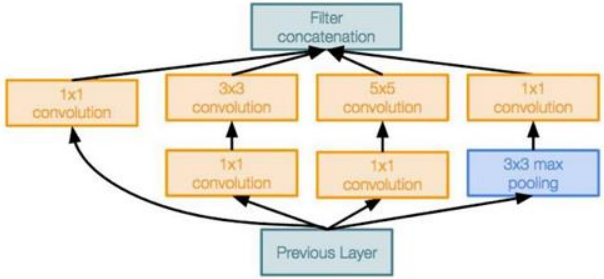
22 total layers with weights (including each parallel layer in an Inception module)

Case Study: GoogLeNet

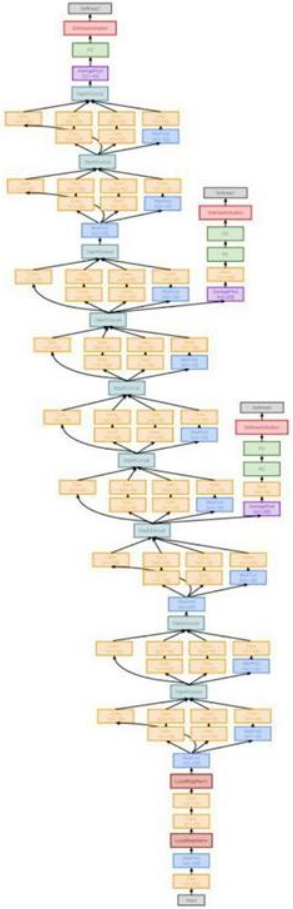
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

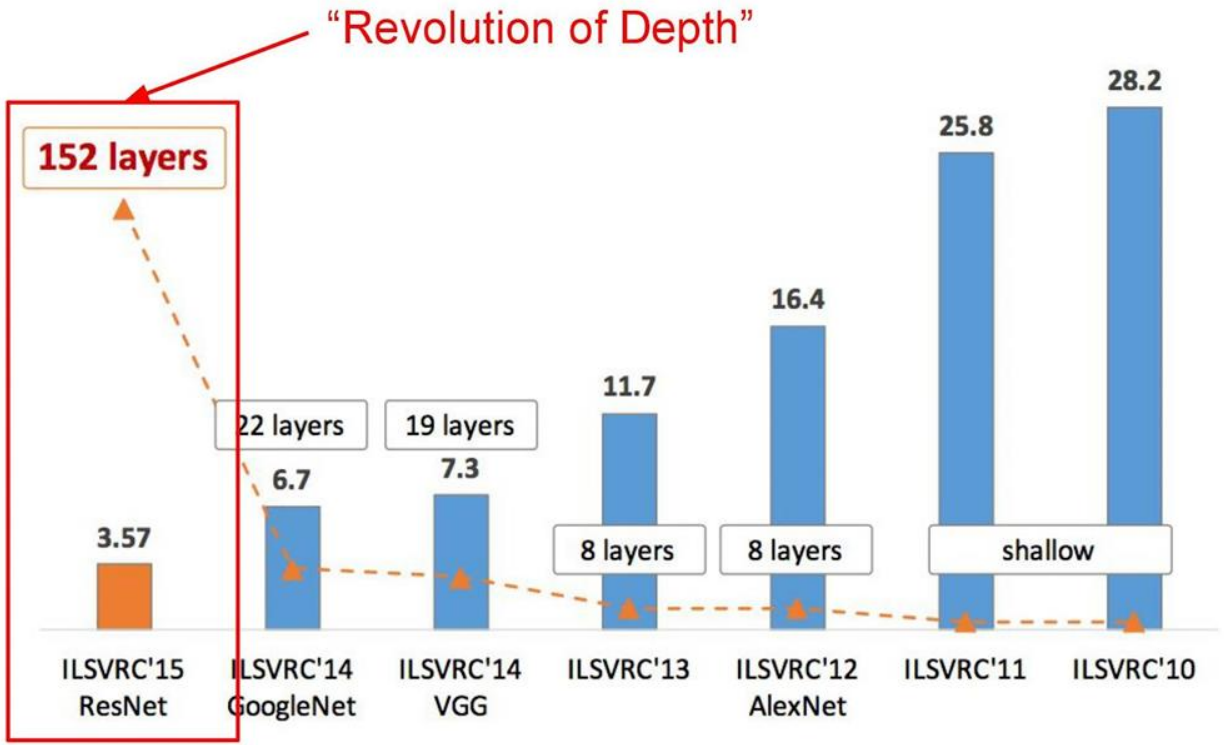
- 22 layers
- Efficient “Inception” module
- No FC layers
- 12x less params than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)



Inception module



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

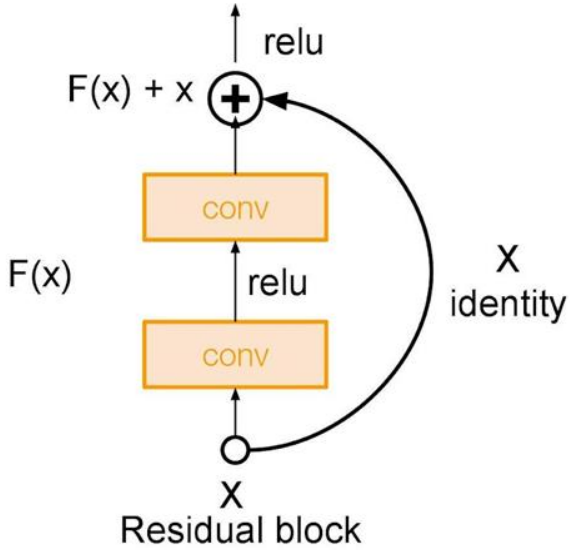


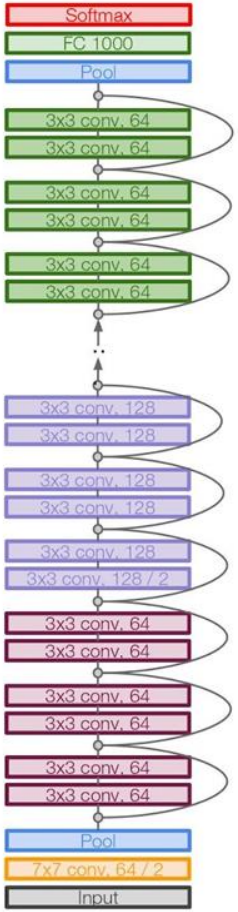
Case Study: ResNet

[He et al., 2015]

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!





Case Study: ResNet

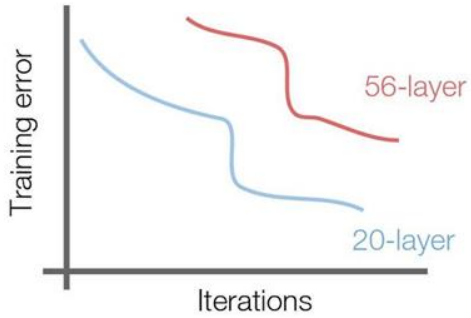
[He et al., 2015]

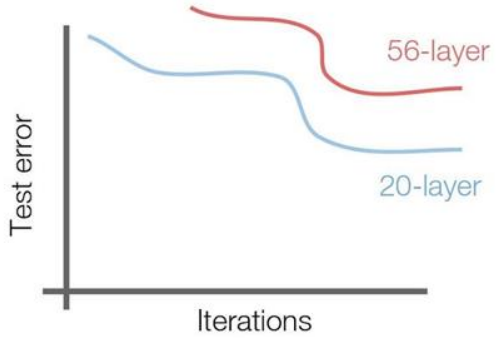
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

Case Study: ResNet

[He et al., 2015]

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





Q: What’s strange about these training and test curves?
[Hint: look at the order of the curves]

Case Study: ResNet

[He et al., 2015]

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

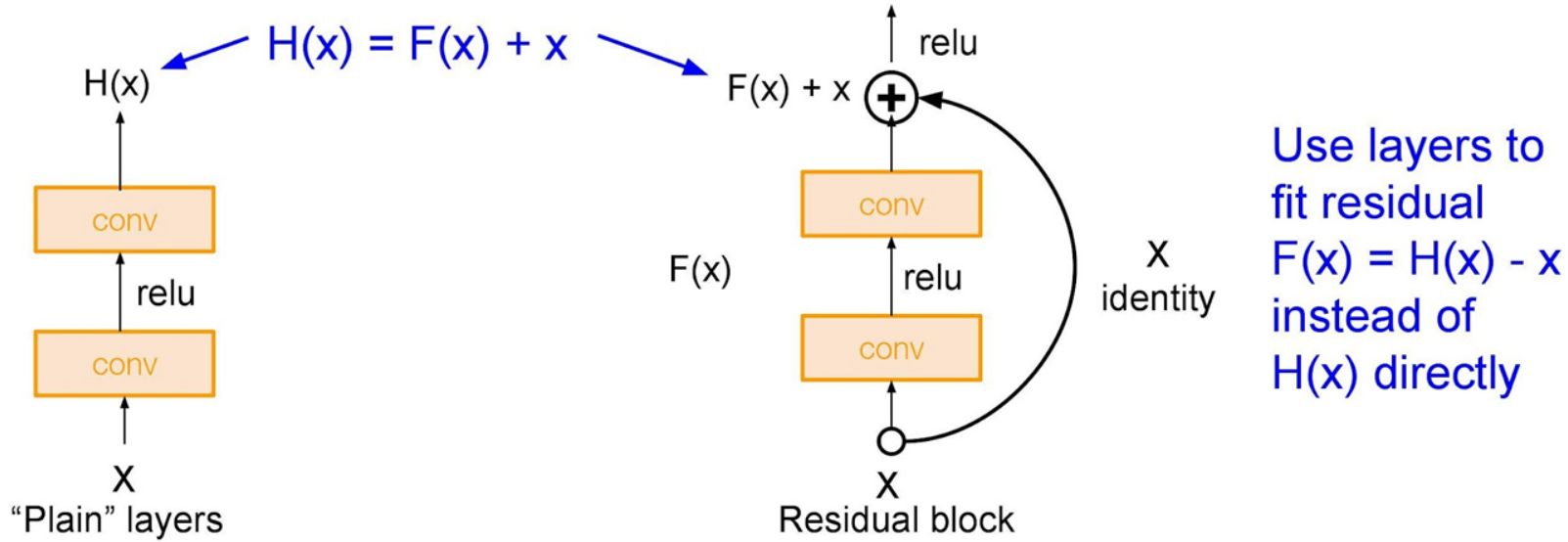
The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

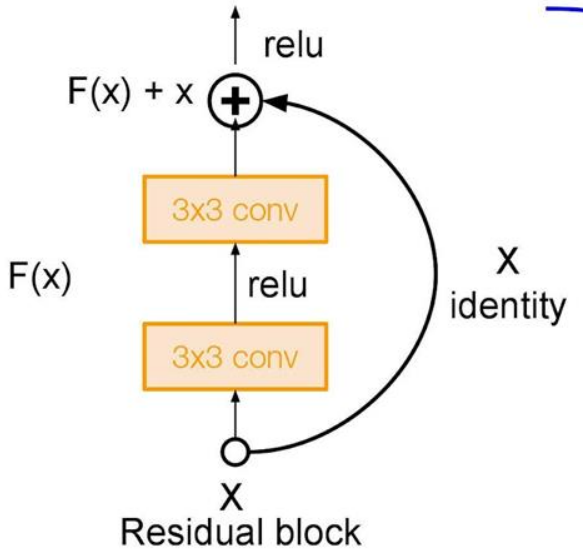


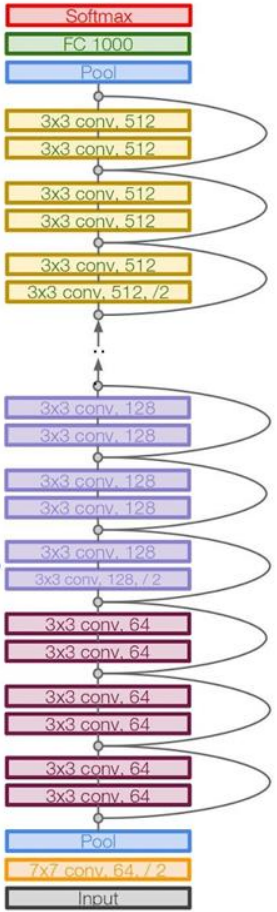
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers



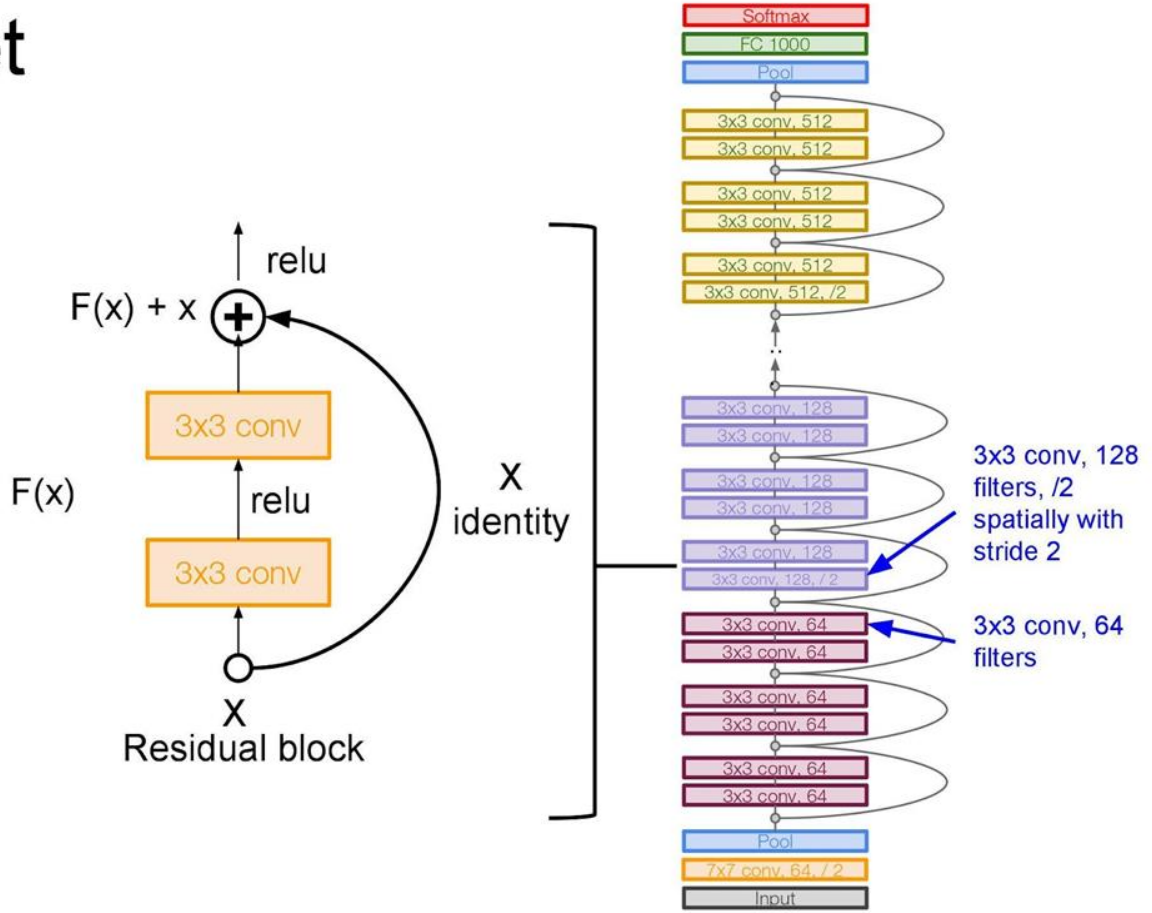


Case Study: ResNet

[He et al., 2015]

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)

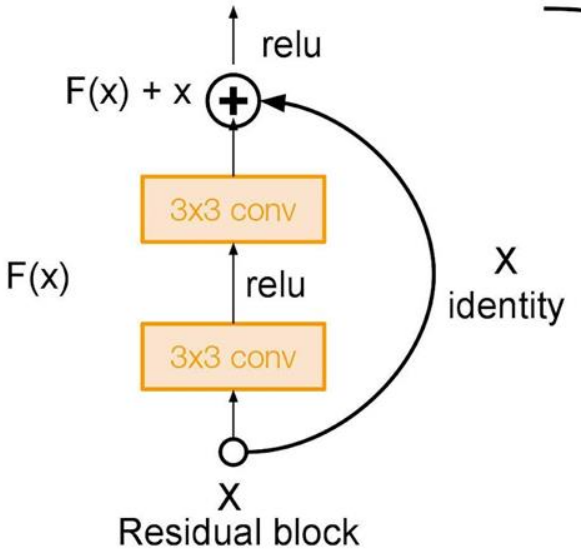


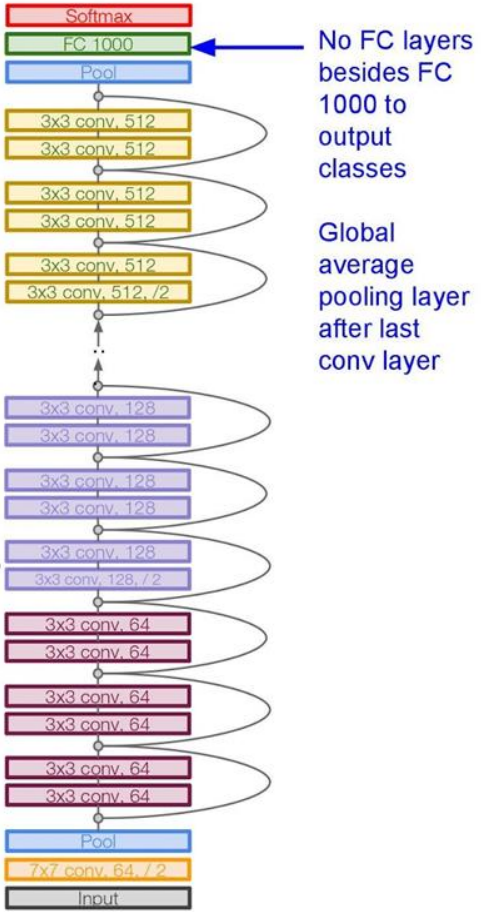
Case Study: ResNet

[He et al., 2015]

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)

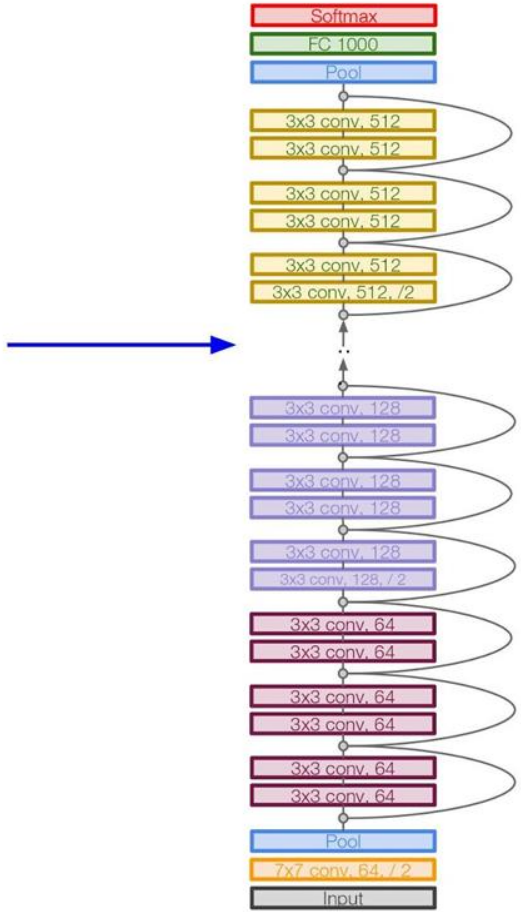




Case Study: ResNet

[He et al., 2015]

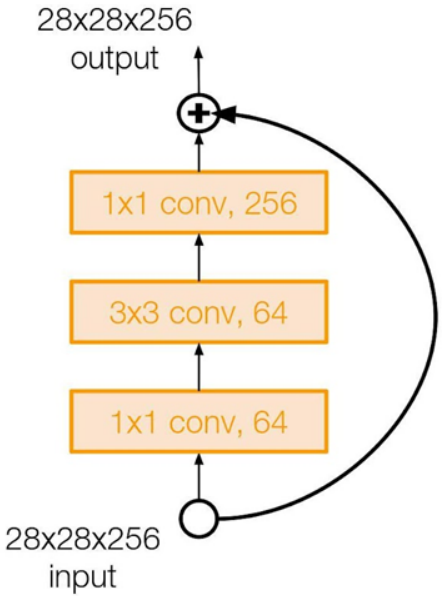
Total depths of 34, 50, 101, or 152 layers for ImageNet



Case Study: ResNet

[He et al., 2015]

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



Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

Case Study: ResNet

[He et al., 2015]

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: "*Ultra-deep*" (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

Summary: CNN Architectures

- VGG, GoogLeNet, ResNet all in wide use, available in model zoos
- ResNet current best default
- Trend towards extremely deep networks
- Significant research centers around design of layer / skip connections and improving gradient flow
- Even more recent trend towards examining necessity of depth vs. width and residual connections

Convolutional Neural Networks (CNNs)

What we learnt about CNNs?

Introduction

- CNNs over fully connected
- Different layers in CNN
  - ❑ Convolutional layer
  - ❑ RelU
  - ❑ Pooling layer
  - ❑ Fully connected layer

State of the art architectures

- AlexNet and VGG Net
- GoogLeNet (inception module)
- ResNets (Residual module)
  CVPR’ 17 best paper