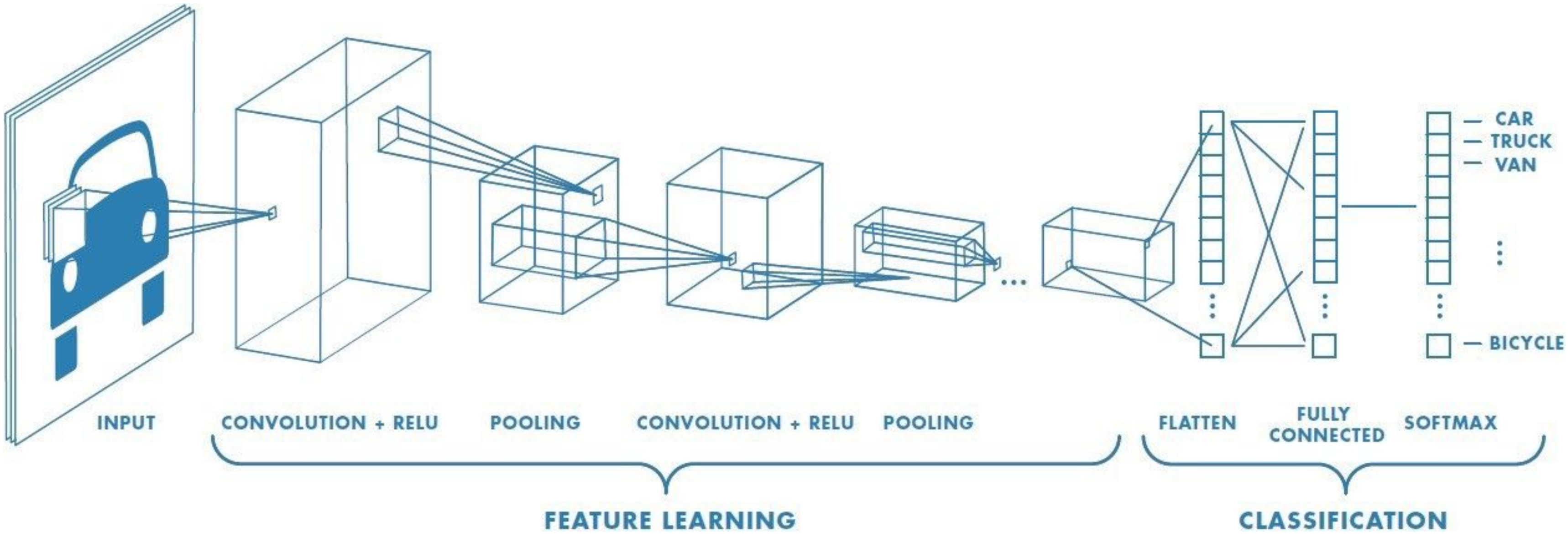
CNNs for image classification

Instructor: Seunghoon Hong

Announcement

- We have a complete list of teams for the final project.
    - Check your new teammates and contact them as soon as possible.
- Assignment 1 will be released tonight!
    - **Due date: midnight September 23** (late submission due: midnight September 25)
    - We provide you a colab example that walks you through the image classification process using CNN
    - If you are not familiar with PyTorch yet, it may take some time! Start it ASAP.

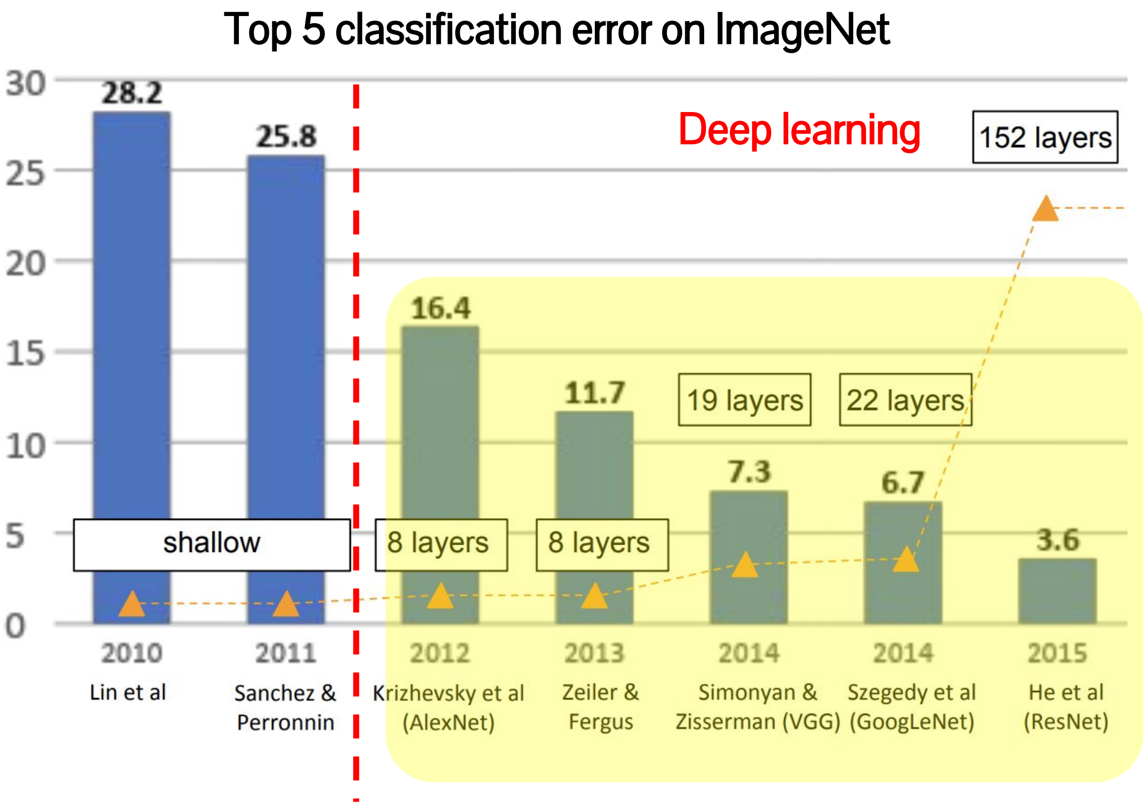
Review: CNN for image classification



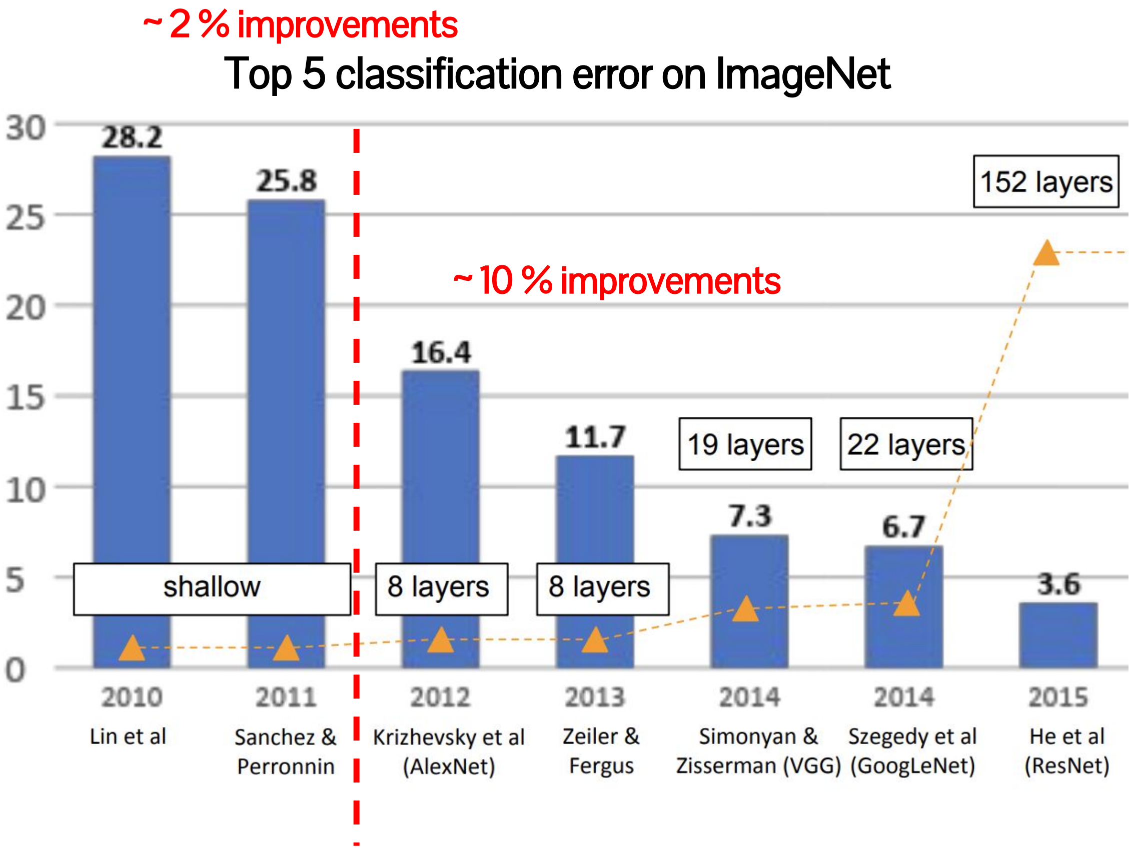
Today’s agenda

- CNN architectures for image classification
  - AlexNet, ZFNet, VGGNet, Resnet, DenseNet
- Training tips for CNN
  - data augmentation, fine-tuning

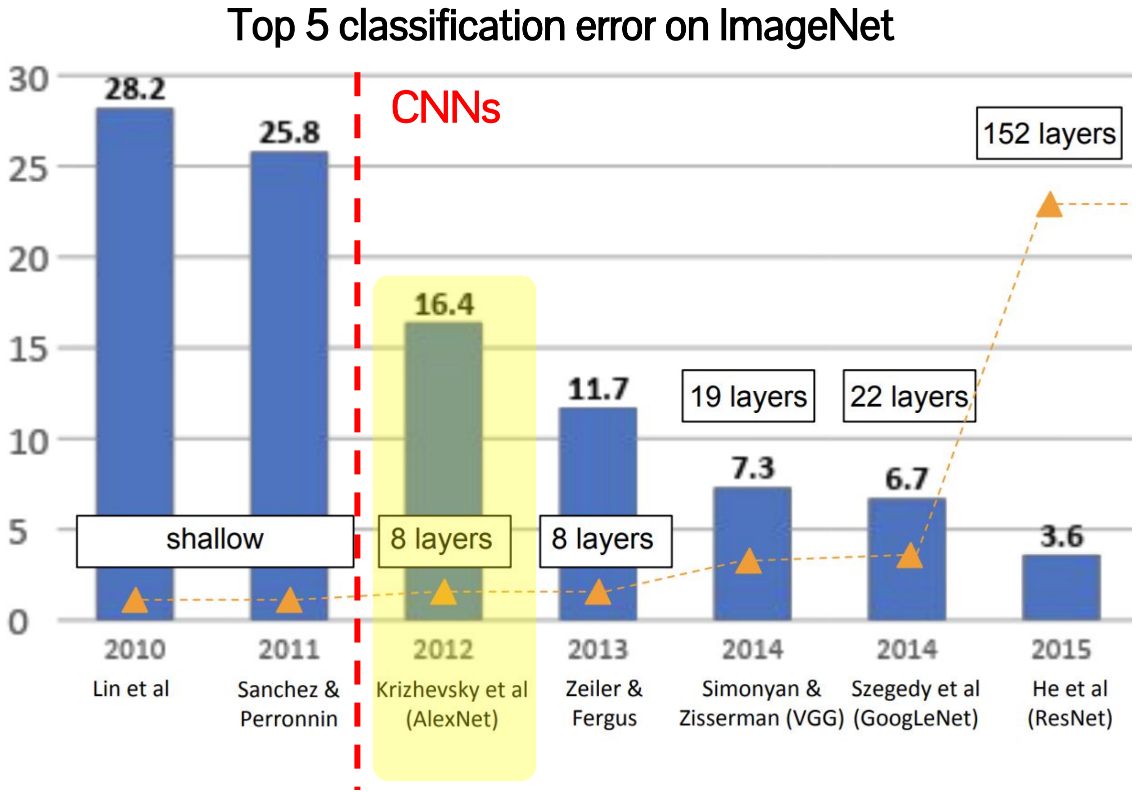
Case study: CNN architectures for image classification



Case study: CNN architectures for image classification



Case study: CNN architectures for image classification

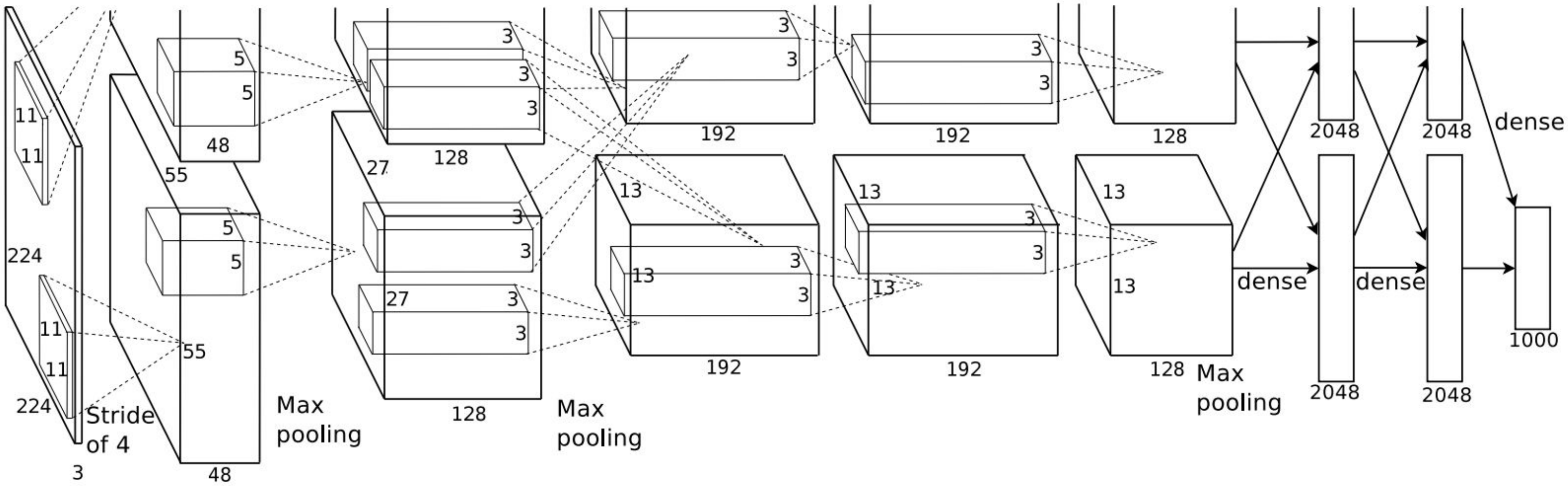


AlexNet [Krizhevsky et al. 2012]:

The first CNN that accelerates the deep learning era in vision

AlexNet

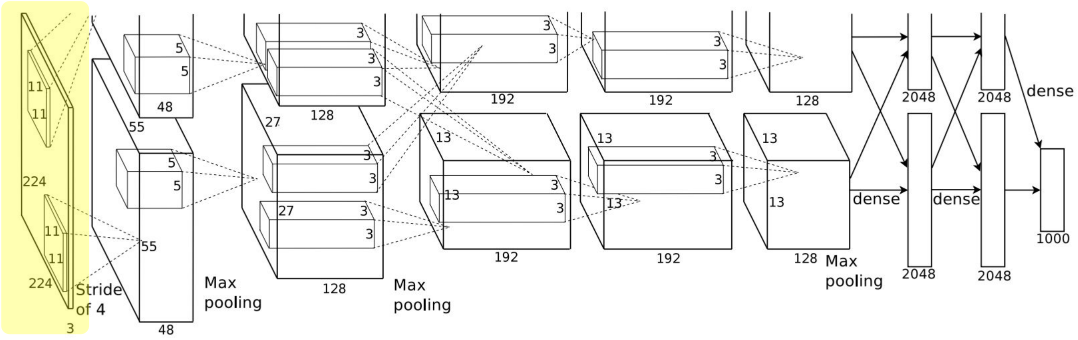
- Architecture
  - 8 layer CNN = 5 convolution layers + 3 fully-connected layers



AlexNet

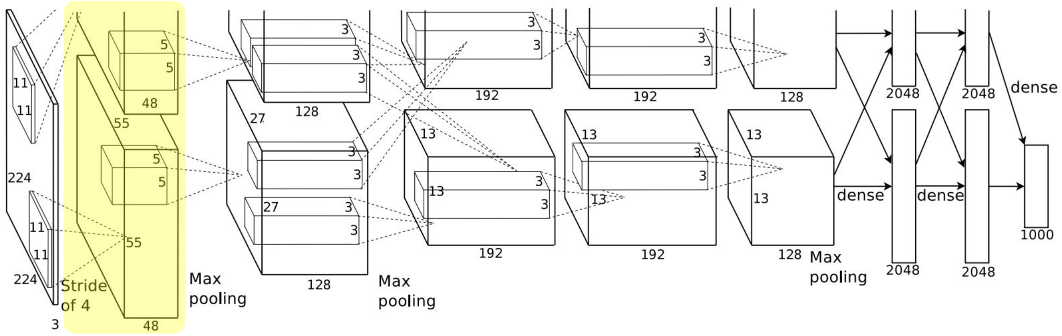
- Architecture

[227x227x3] Input



AlexNet

- Architecture



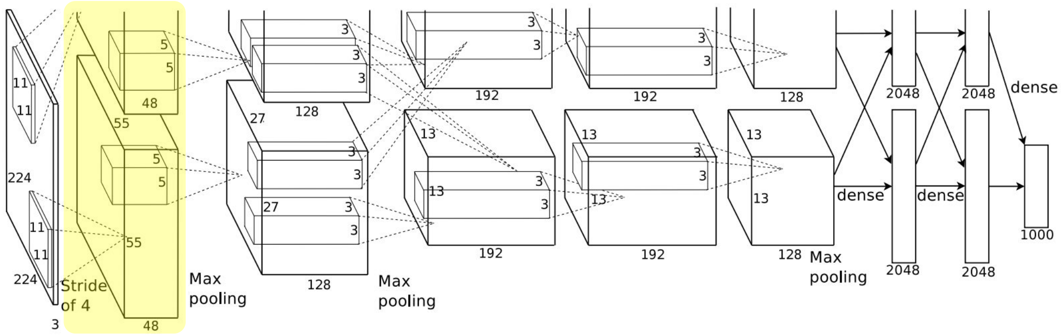
[227x227x3] Input

[? x ? x ?] Conv1 (96, kernel=11x11, stride=4, pad=0) + **ReLU**

(Feature size - kernel size)/stride + 1

AlexNet

- Architecture

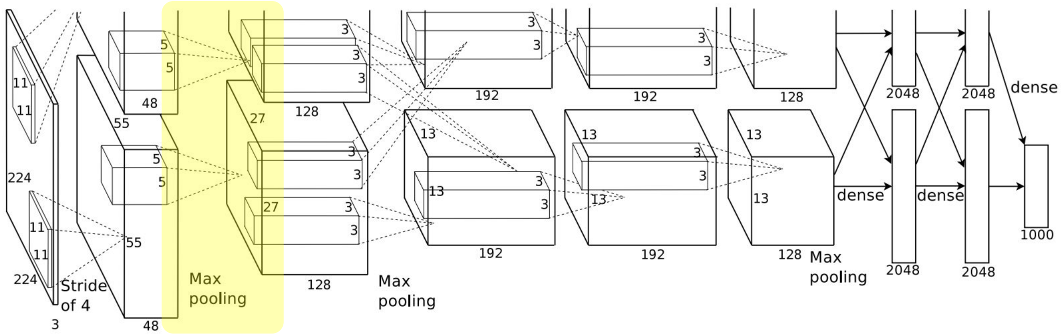


[227x227x3] Input

[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + **ReLU**

AlexNet

- Architecture



[227x227x3] Input

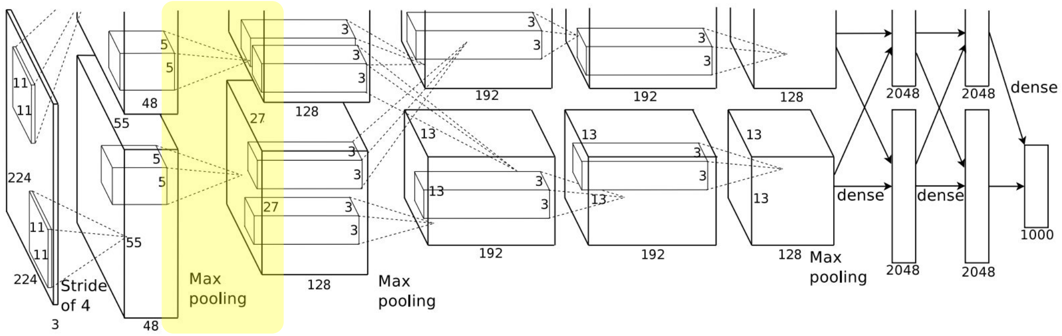
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + **ReLU**

[? x ? x ?] MaxPool1 (kernel=3x3, stride=2)

(Feature size - kernel size)/stride + 1

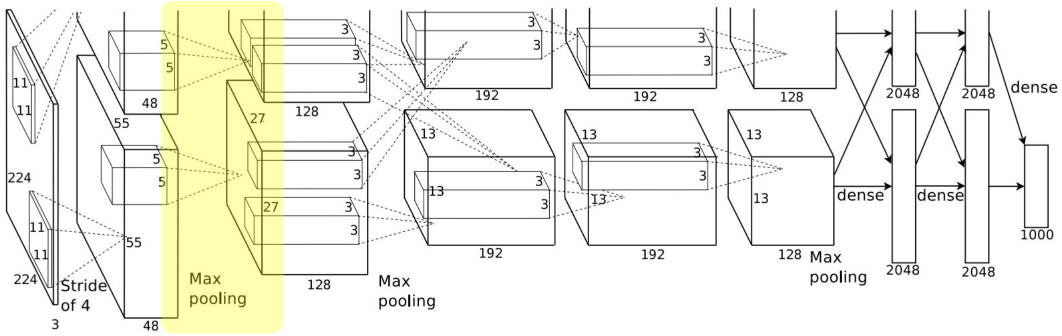
AlexNet

- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)

AlexNet



- Architecture

[227x227x3] Input

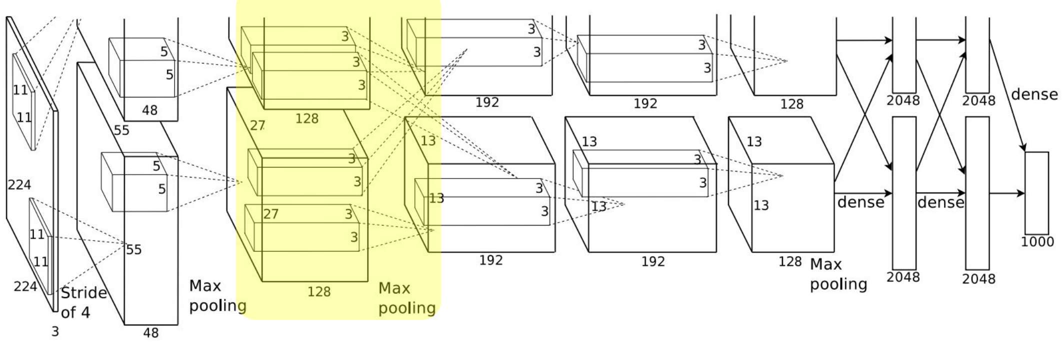
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU

[27x27x96] MaxPool1 (kernel=3x3, stride=2)

[27x27x96] Norm1

AlexNet

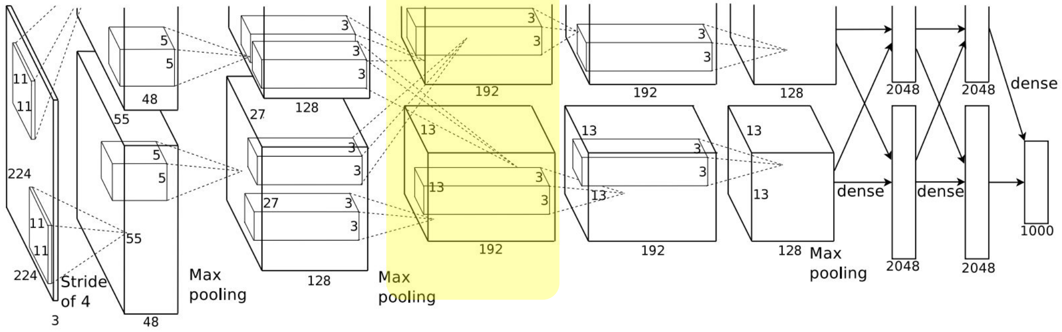
- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2

AlexNet

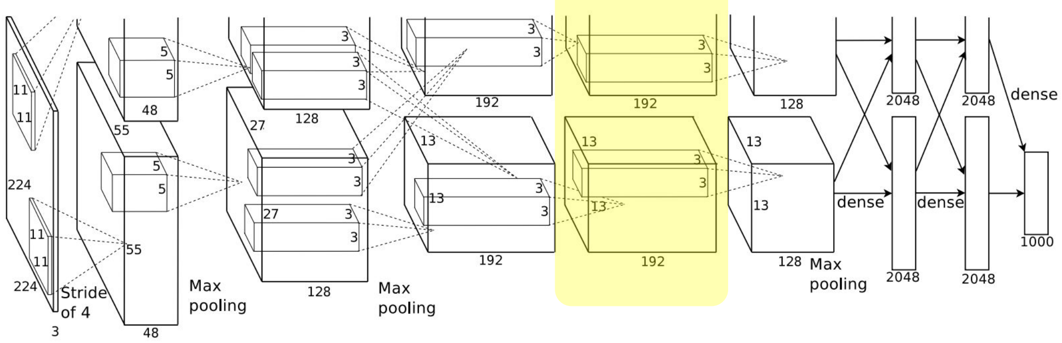
- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU

AlexNet

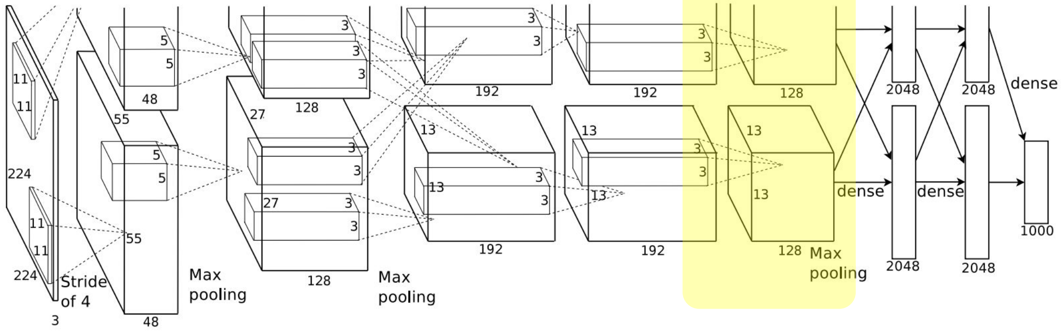
- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU

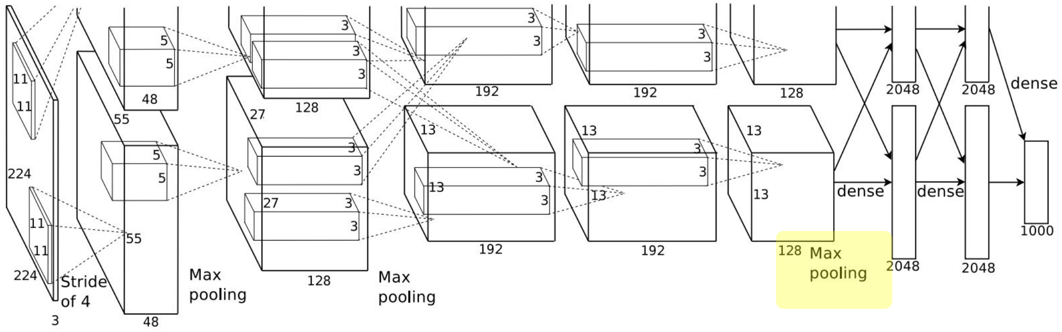
AlexNet

- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU

AlexNet

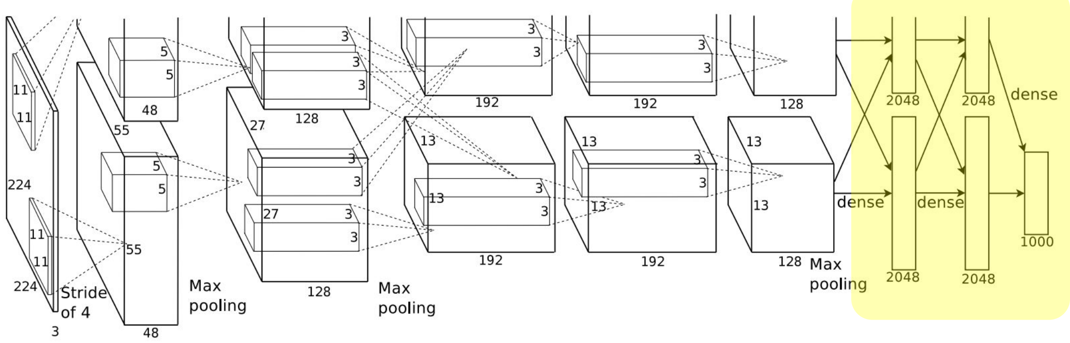


- Architecture

[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU
[6x6x256] MaxPool3 (kernel=3x3, stride=2)

AlexNet

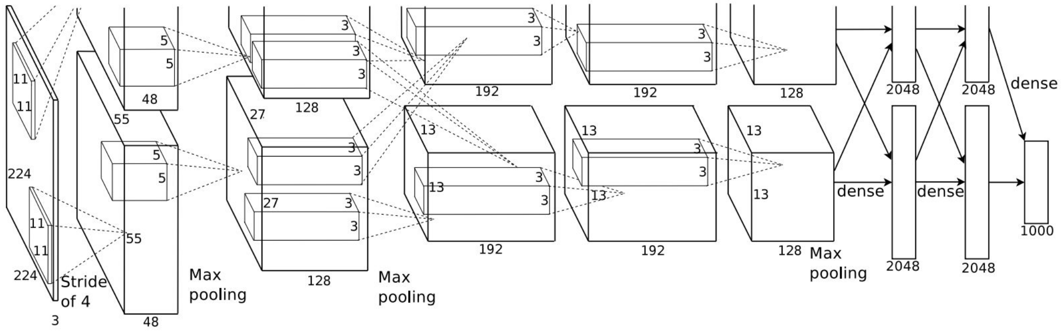
- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096)
[4096] FC7 (4096x4096)
[1000] FC8 (4096x1000)

AlexNet

- Architecture

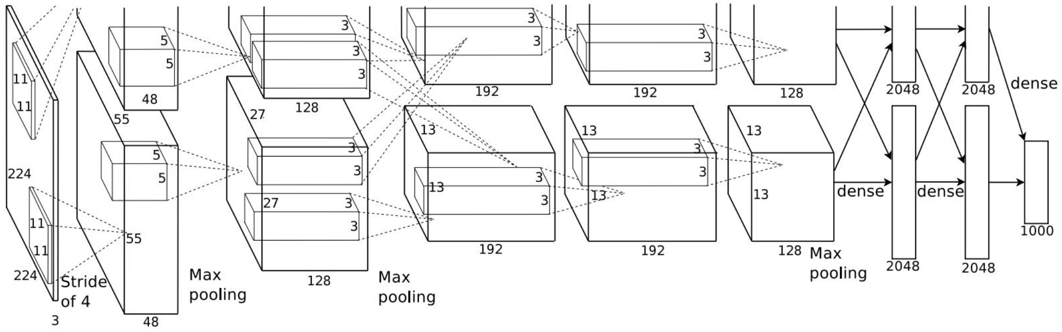


[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096)
[4096] FC7 (4096x4096)
[1000] FC8 (4096x1000)

First CNN that applied ReLU nonlinearity
(before: sigmoid or tanh → **problems**?)

AlexNet

- Architecture

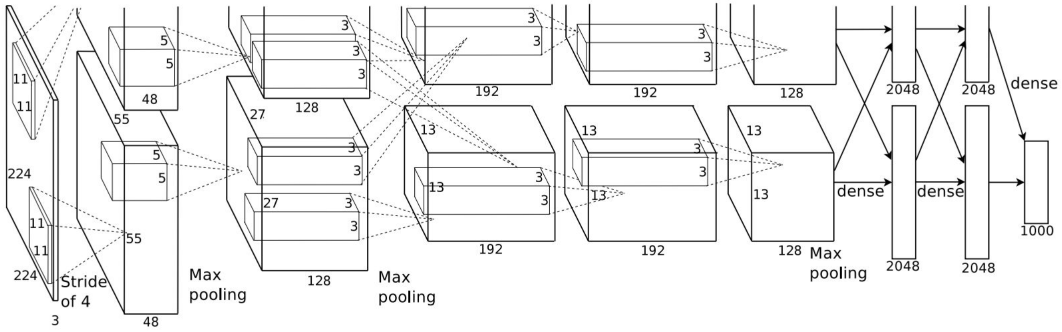


[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096)
[4096] FC7 (4096x4096)
[1000] FC8 (4096x1000)

First CNN that applied ReLU nonlinearity (before: sigmoid or tanh → Saturating gradient → slow learning)

AlexNet

- Architecture

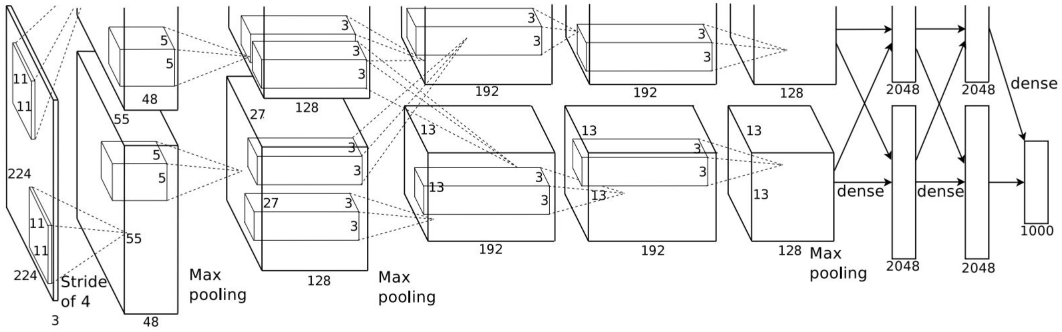


[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096)
[4096] FC7 (4096x4096)
[1000] FC8 (4096x1000)

First CNN that applied ReLU nonlinearity

Large stride at the first layer → **to reduce feature size and save computation**
(the model is trained with ~3G GPU memory)

AlexNet



- Architecture

[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + **ReLU**
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + **ReLU**
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + **ReLU**
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + **ReLU**
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + **ReLU**
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096)
[4096] FC7 (4096x4096)
[1000] FC8 (4096x1000)

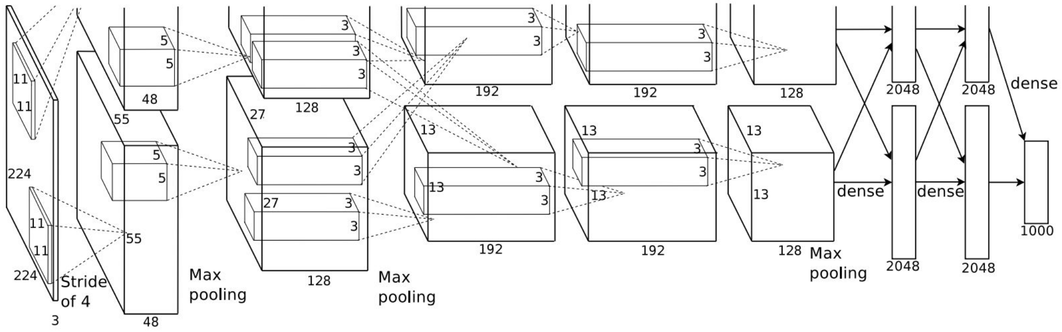
First CNN that applied ReLU nonlinearity

Large stride at the first layer

Normalization layer
(**no more used in recent CNNs**)

AlexNet

- Architecture



[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096)
[4096] FC7 (4096x4096)
[1000] FC8 (4096x1000)

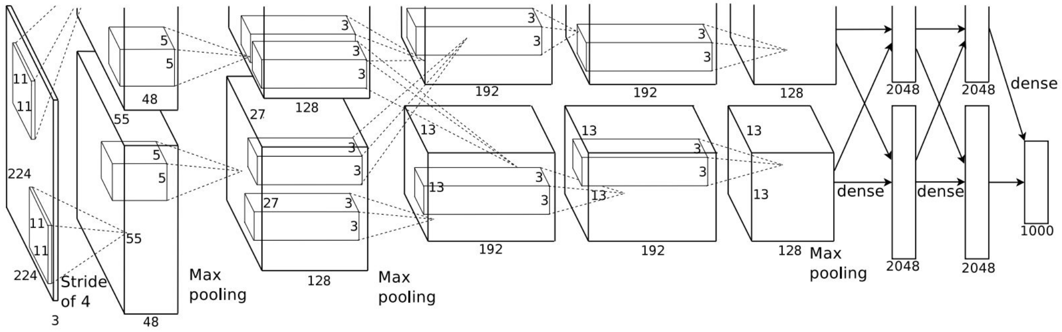
First CNN that applied ReLU nonlinearity

Large stride at the first layer

Normalization layer

Dropout=0.5 is applied in FC layers
to prevent overfitting

AlexNet

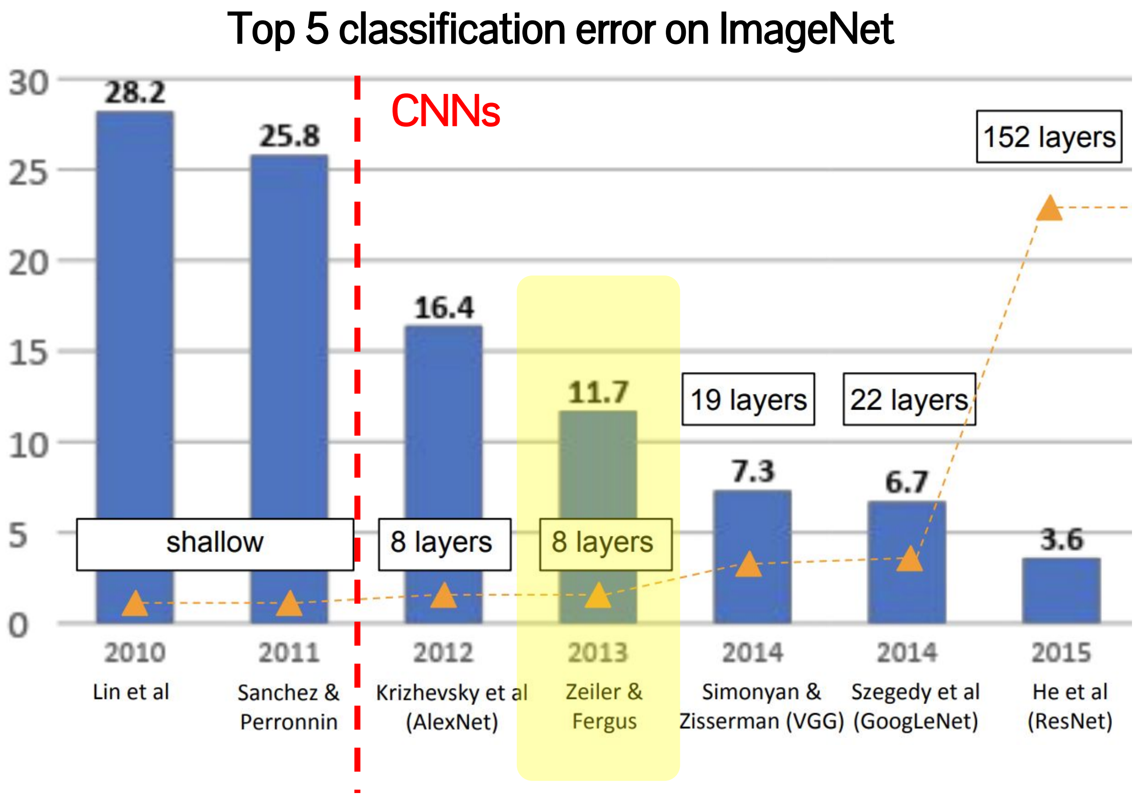


- Number of parameters

[227x227x3] Input
[55x55x96] Conv1 (96, kernel=11x11, stride=4, pad=0) + ReLU → 11.6K
[27x27x96] MaxPool1 (kernel=3x3, stride=2)
[27x27x96] Norm1
[27x27x256] Conv2 (256, kernel=5x5, stride=1, pad=2) + ReLU → 6.4K
[13x13x256] MaxPool2 (kernel=3x3, stride=2)
[13x13x256] Norm2
[13x13x384] Conv3 (384, kernel=3x3, stride=1, pad=1) + ReLU → 3.5K
[13x13x384] Conv4 (384, kernel=3x3, stride=1, pad=1) + ReLU → 3.5K
[13x13x256] Conv5 (256, kernel=3x3, stride=1, pad=1) + ReLU → 2.3K
[6x6x256] MaxPool3 (kernel=3x3, stride=2)
[4096] FC6 (9216x4096) → 37749K
[4096] FC7 (4096x4096) → 16777K
[1000] FC8 (4096x1000) → 4096K

FC layers are extremely expensive!

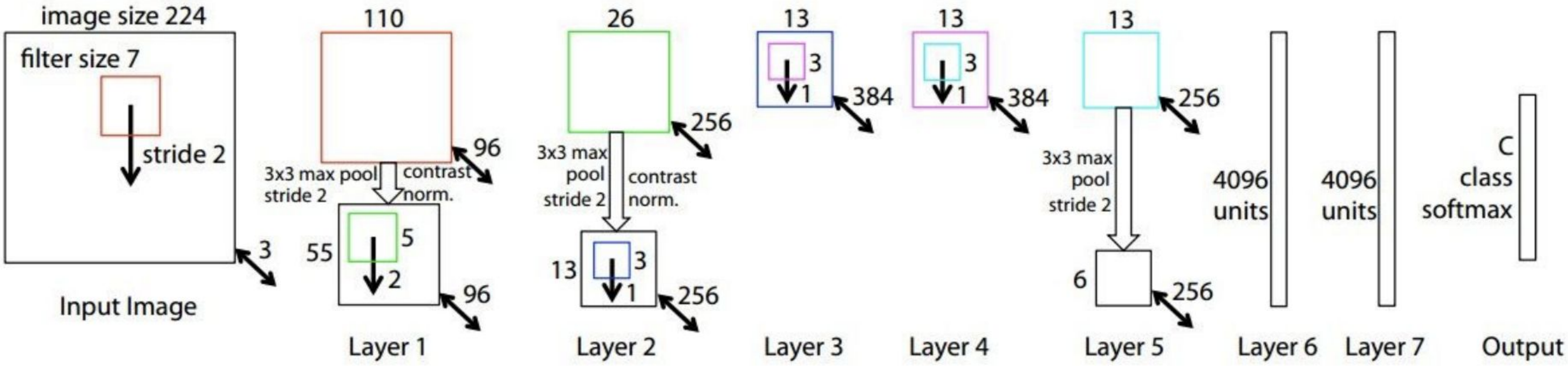
Case study: CNN architectures for image classification



ZFNet [Zeiler et al. 2013]:

Better tuning of network parameters over AlexNet

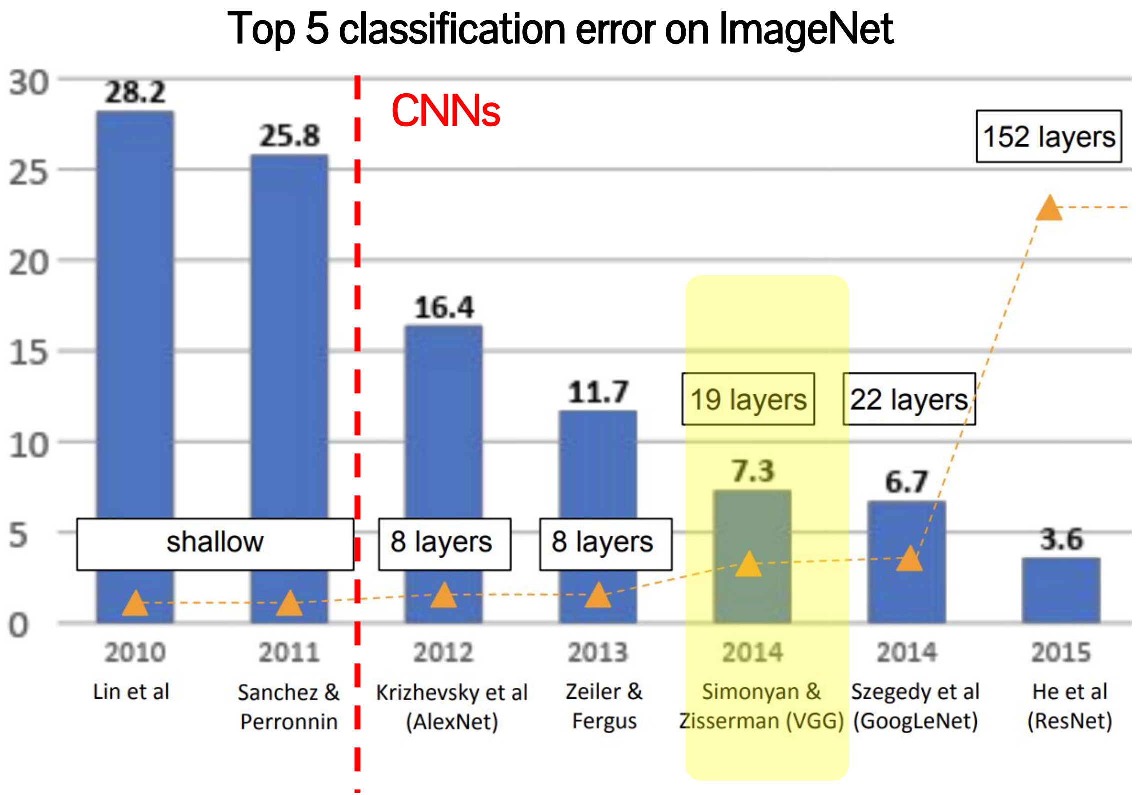
ZFNet



AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% -> 11.7%

Case study: CNN architectures for image classification



VGGNet [Simonyan et al. 2013]:

Building deeper network with smaller convolutional filters

VGGNet

- Use only small conv filters (3x3 conv)
  (c.f. Early conv layers in AlexNet)
- Stack much deeper layers
  (8 -> 16 or 19 layers)
- Perf. improvement:
  11.7% -> 7.3% top-5 error

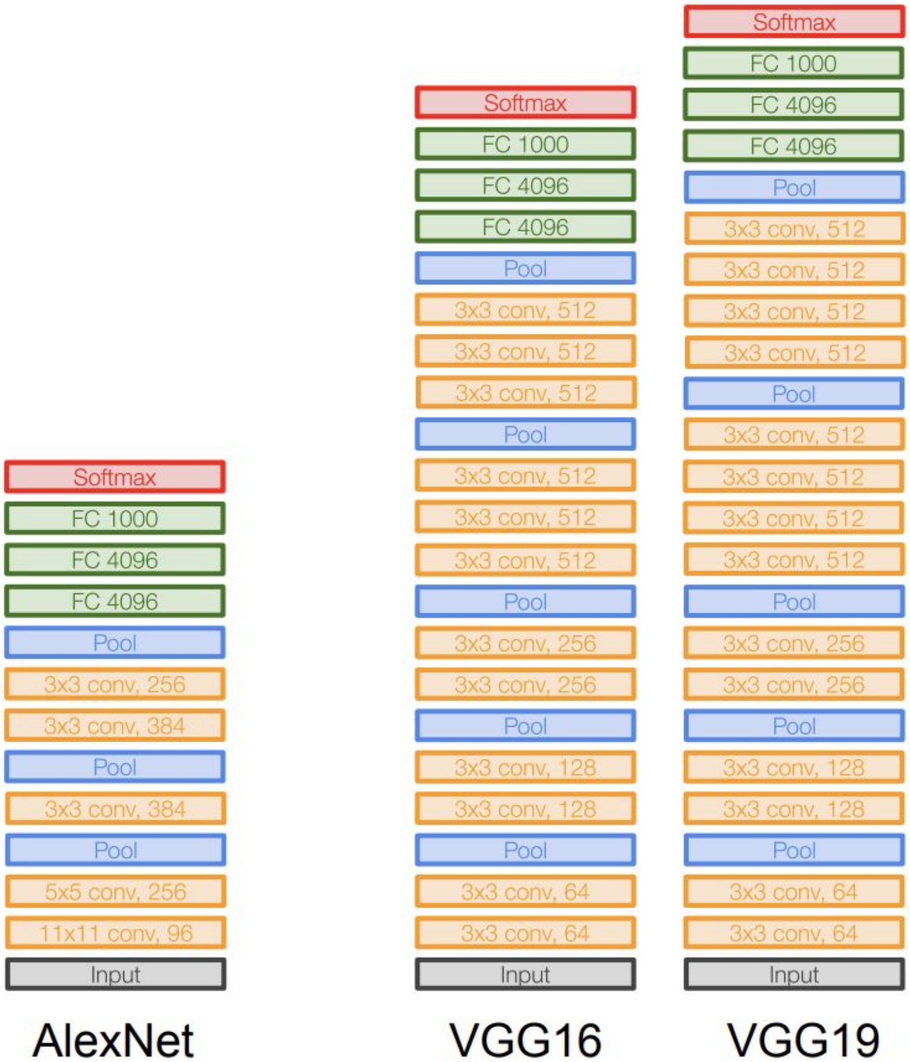


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

VGGNet

- Why stacking small filters is better than a shallow but large filters?
  → More non-linearity!
- It also reduces number of parameters

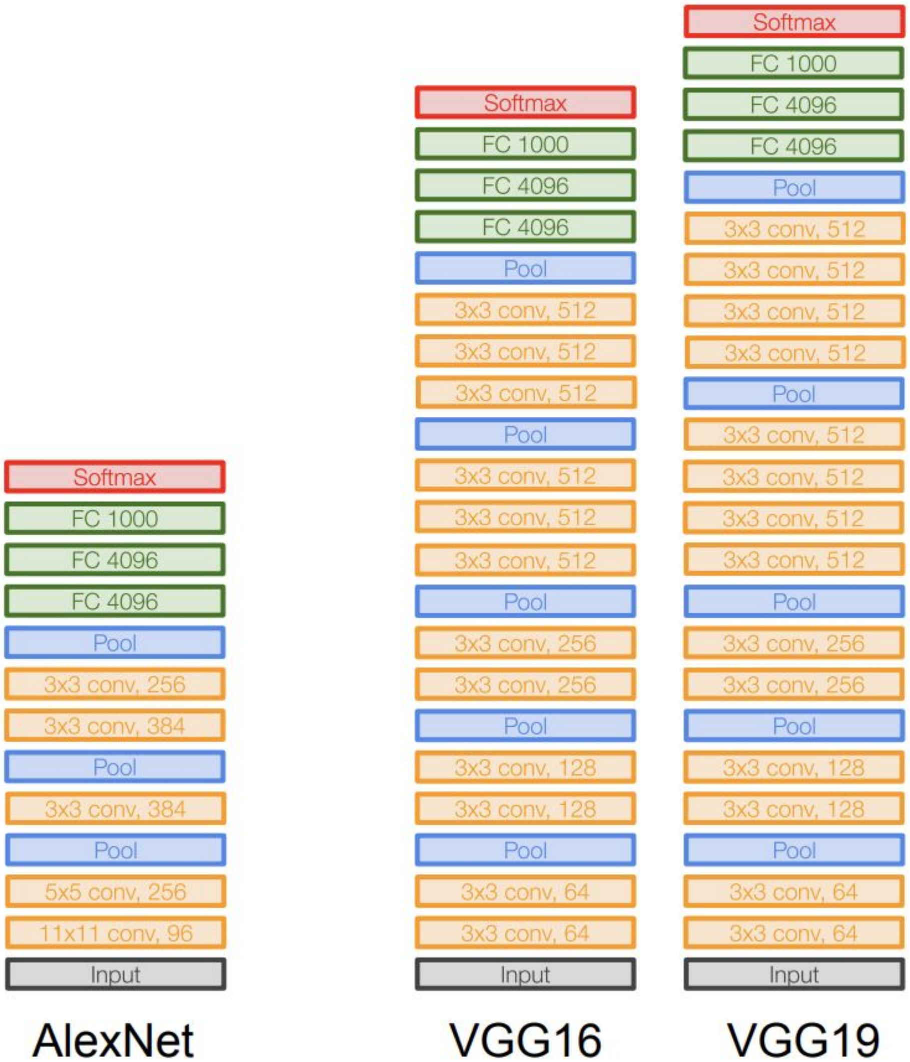
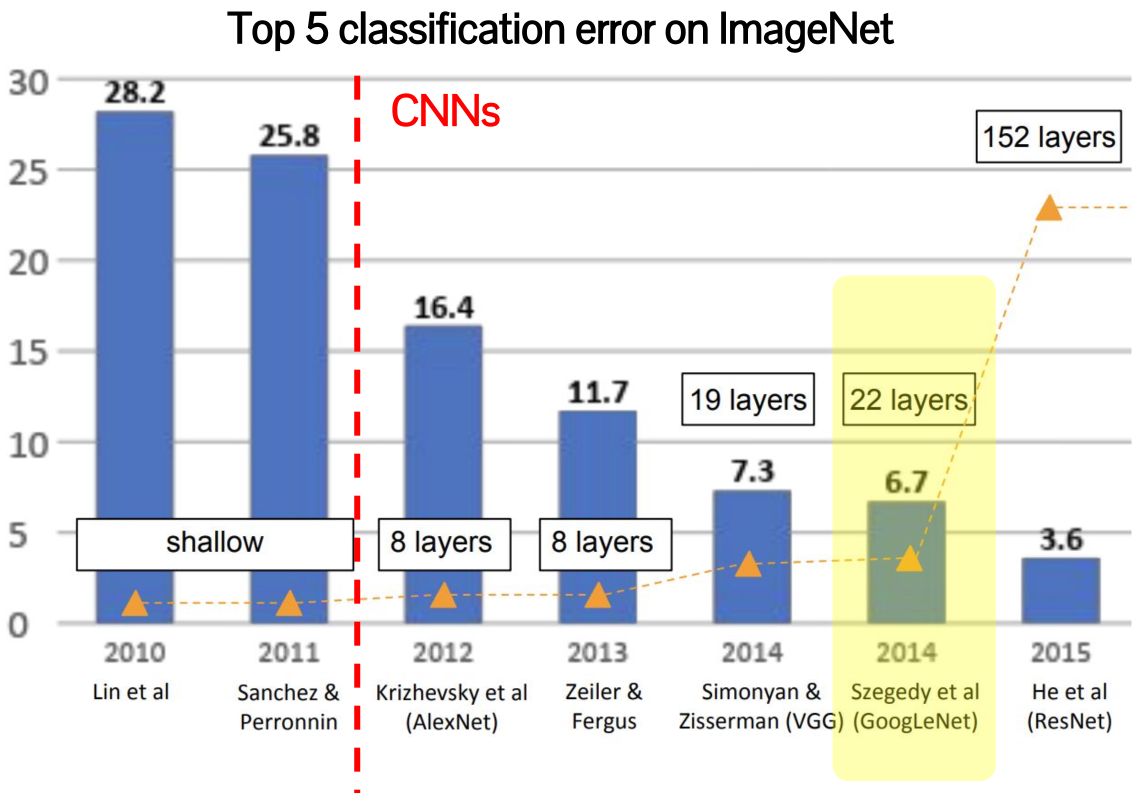


Figure credit: © Stanford CS231n: Convolutional Neural Networks for Visual Recognition

Case study: CNN architectures for image classification

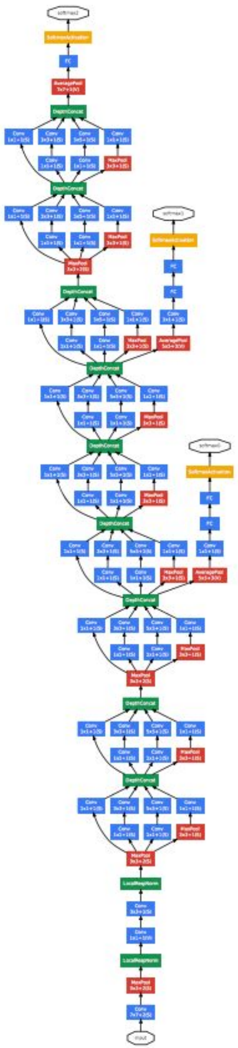


GoogleNet [Simonyan et al. 2013]:

Deeper, much efficient and accurate

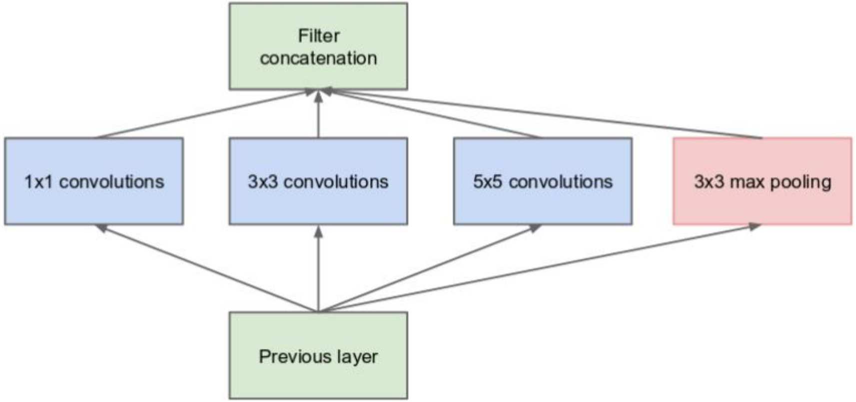
GoogleNet

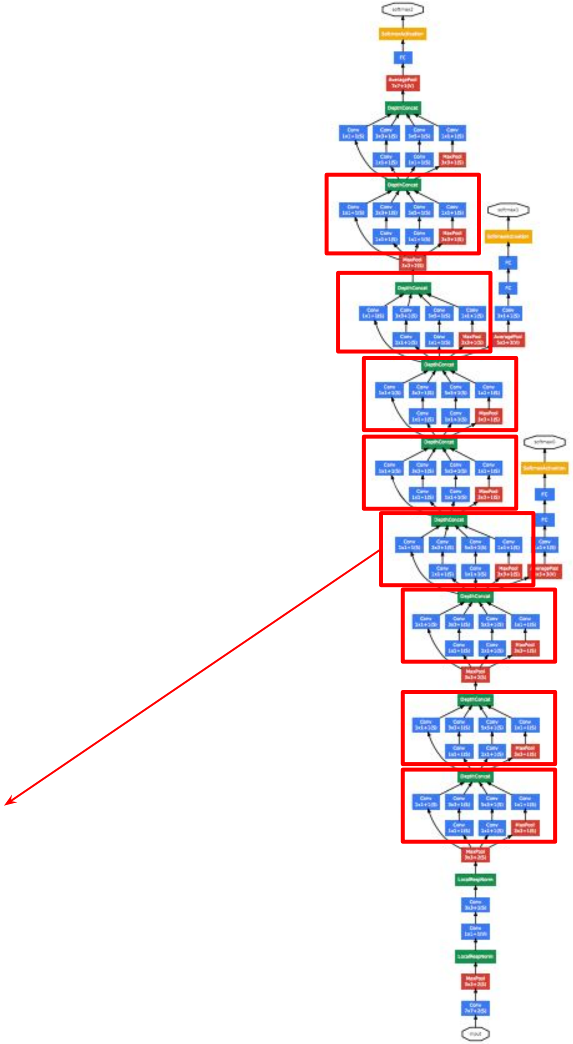
- Deeper network
  - 22 layers (vs. 16 layers in VGG)
  - Each layer is composed of
    a small network (inception module)
- Efficient parametrization
  - No fully-connected layers
  - 12x fewer parameters than AlexNet
- Improved performance
  - 7.3 (VGGNet) → 6.7% top-5 error



GoogleNet

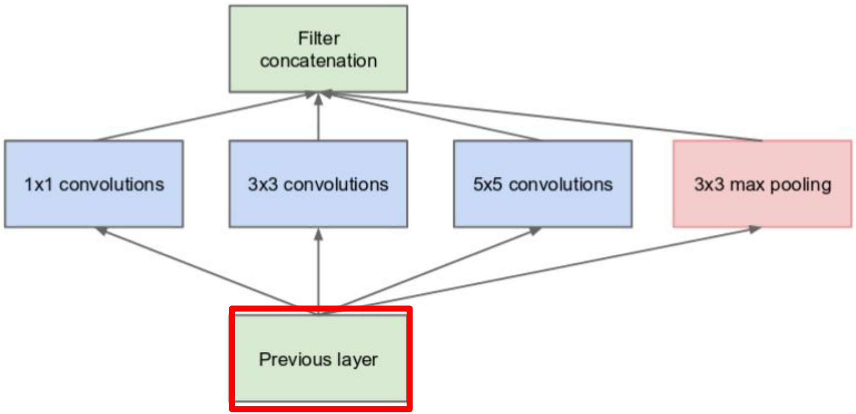
- Inception module
  - Combination of filters in different size
    (1x1, 3x3, 5x5, maxpool with stride=1)
  - Aggregating information in multiple receptive fields
  - Combine all filter responses by depth-wise concatenation



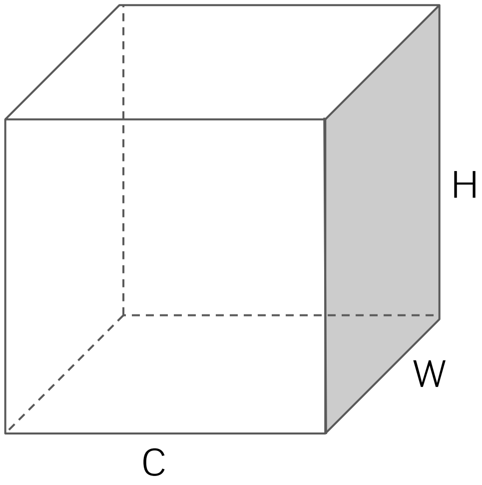


Inception module

- Closer inspection
  - 1x1 convolution

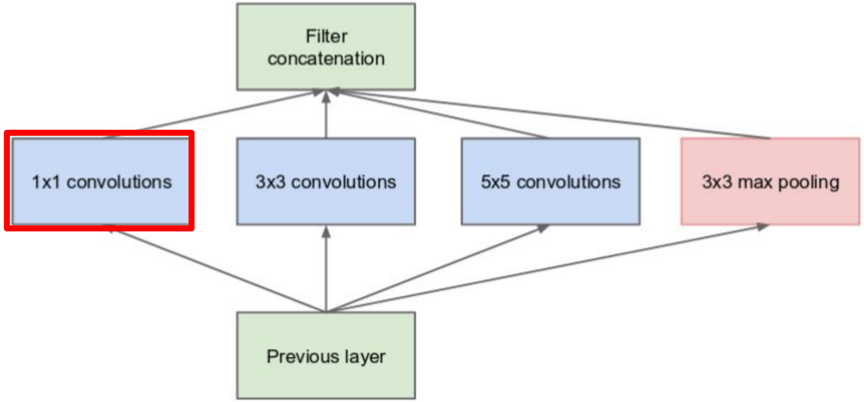


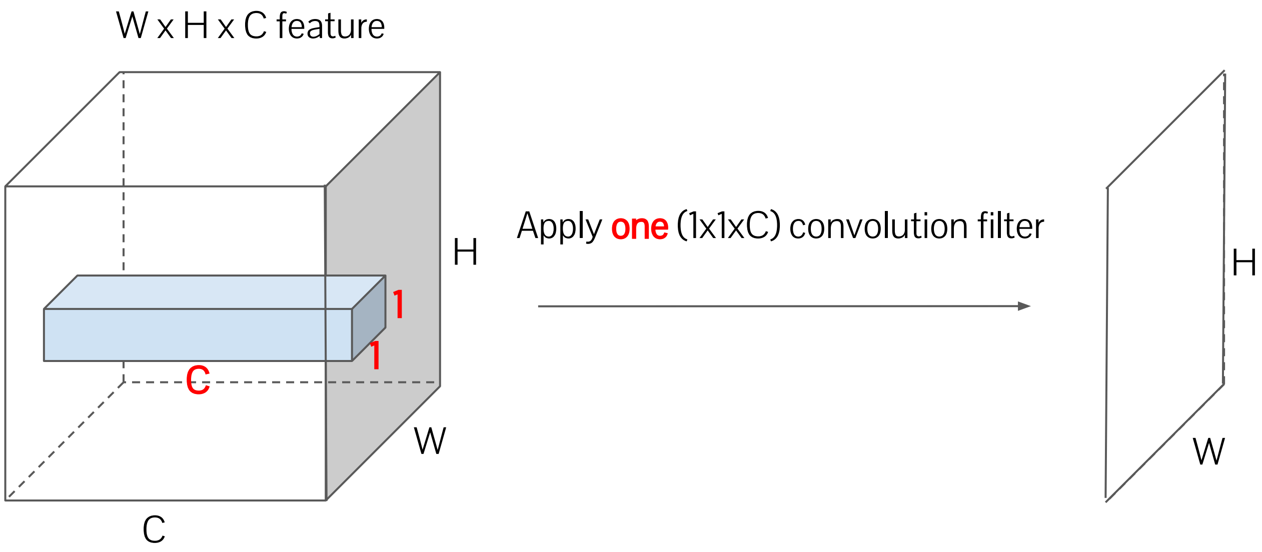
W x H x C feature



Inception module

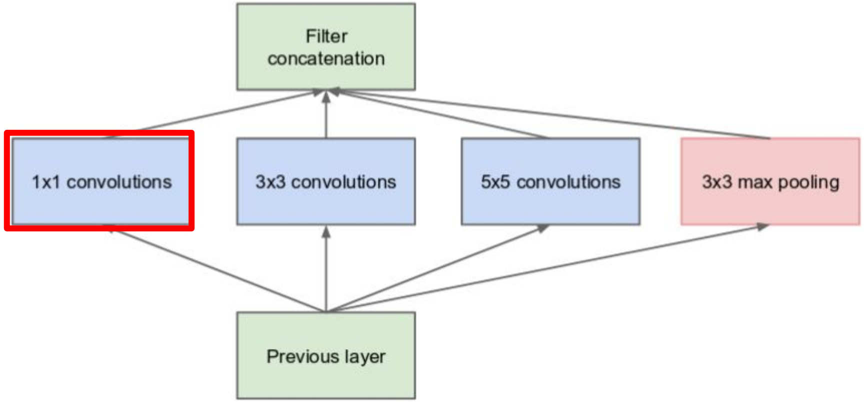
- Closer inspection
  - 1x1 convolution





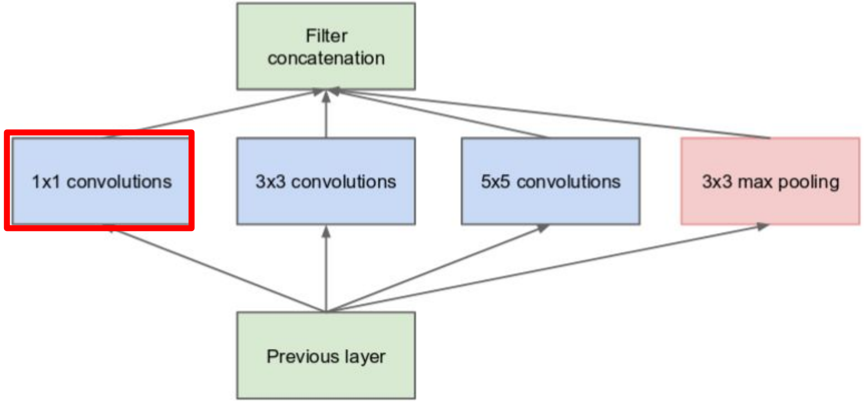
Inception module

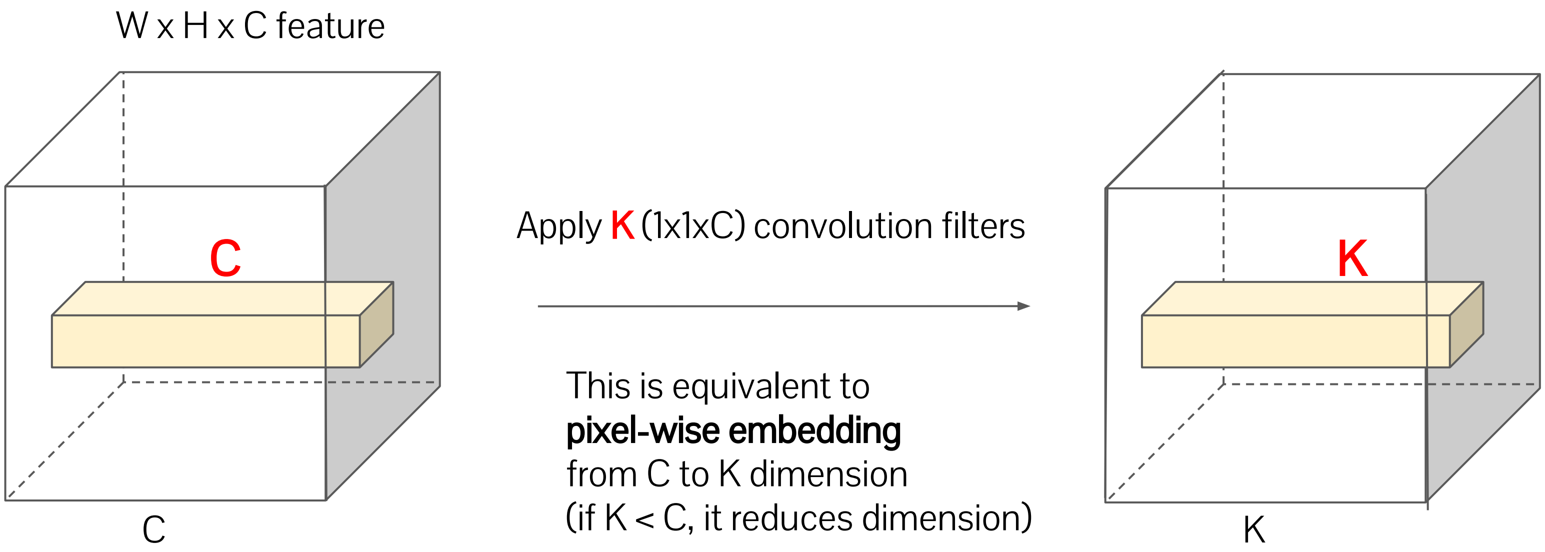
- Closer inspection
  - 1x1 convolution



Inception module

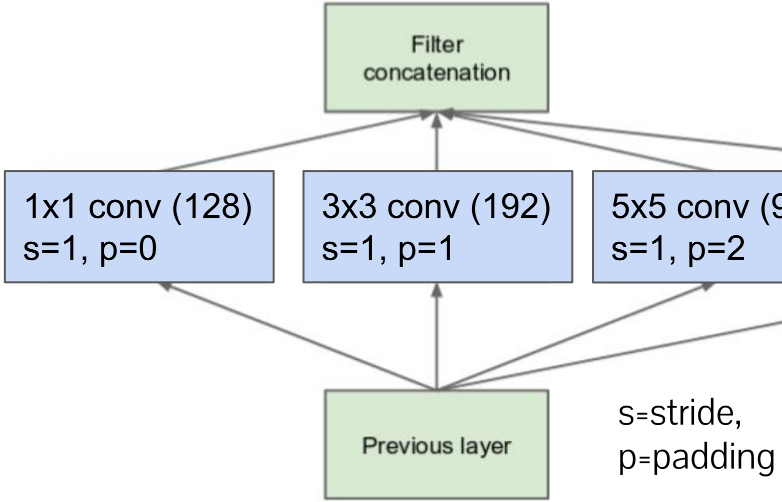
- Closer inspection
    - 1x1 convolution





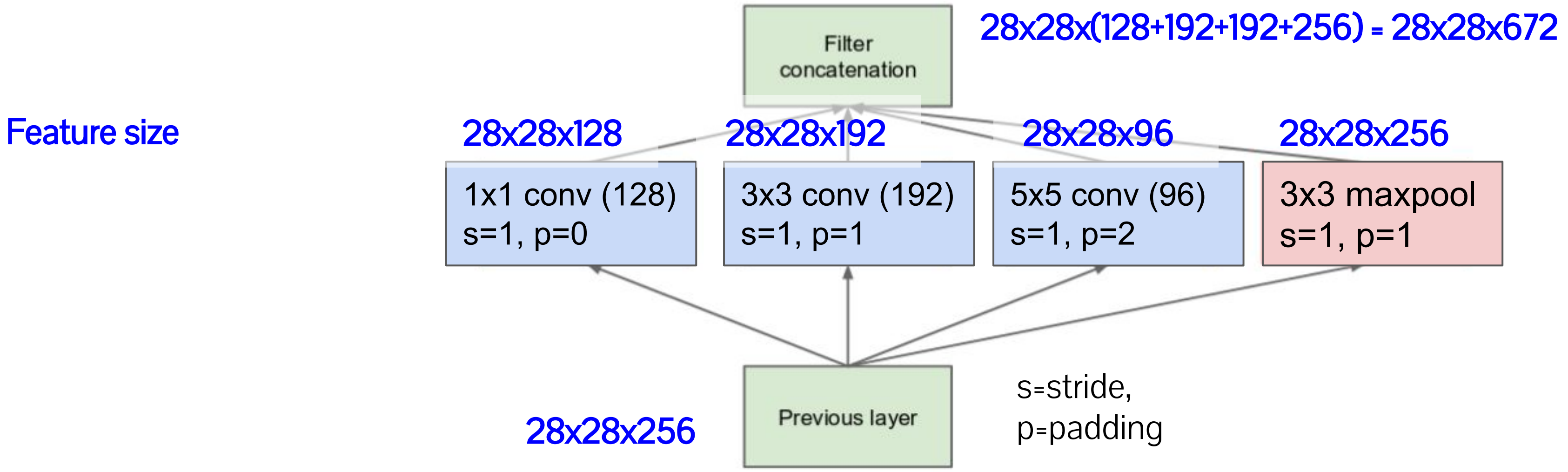
Inception module

- Closer inspection
  - 1x1 convolution, 3x3 convolutions, 5x5 convolutions
    - Convolutions with different receptive fields.
    - Set to have the same spatial feature size
  - Max pooling for additional spatial abstraction



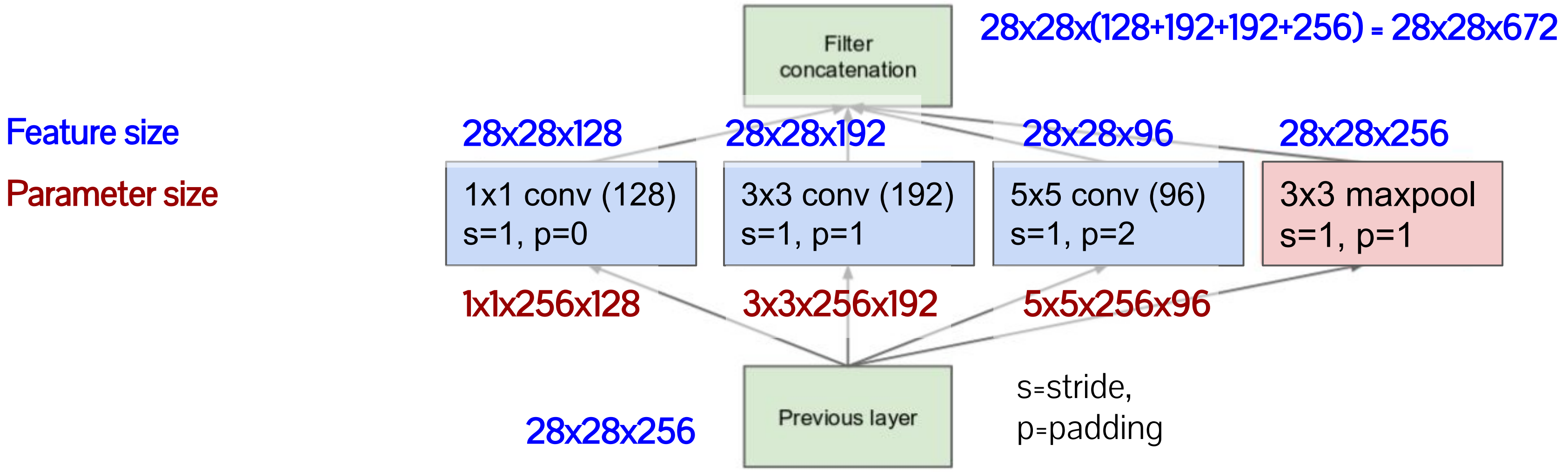
Inception module

- Closer inspection
  - 1x1 convolution, 3x3 convolutions, 5x5 convolutions
    - Convolutions with different receptive fields.
    - Set to have the same spatial feature size
  - Max pooling for additional spatial abstraction



Inception module

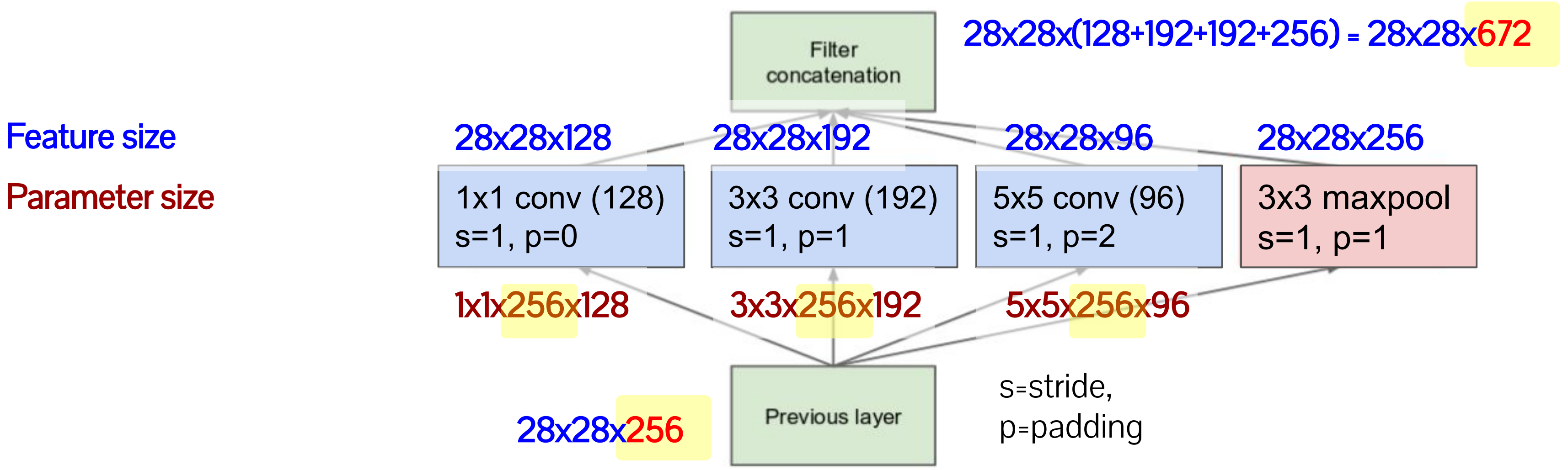
- Closer inspection
  - 1x1 convolution, 3x3 convolutions, 5x5 convolutions
    - Convolutions with different receptive fields.
    - Set to have the same spatial feature size
  - Max pooling for additional spatial abstraction



Inception module

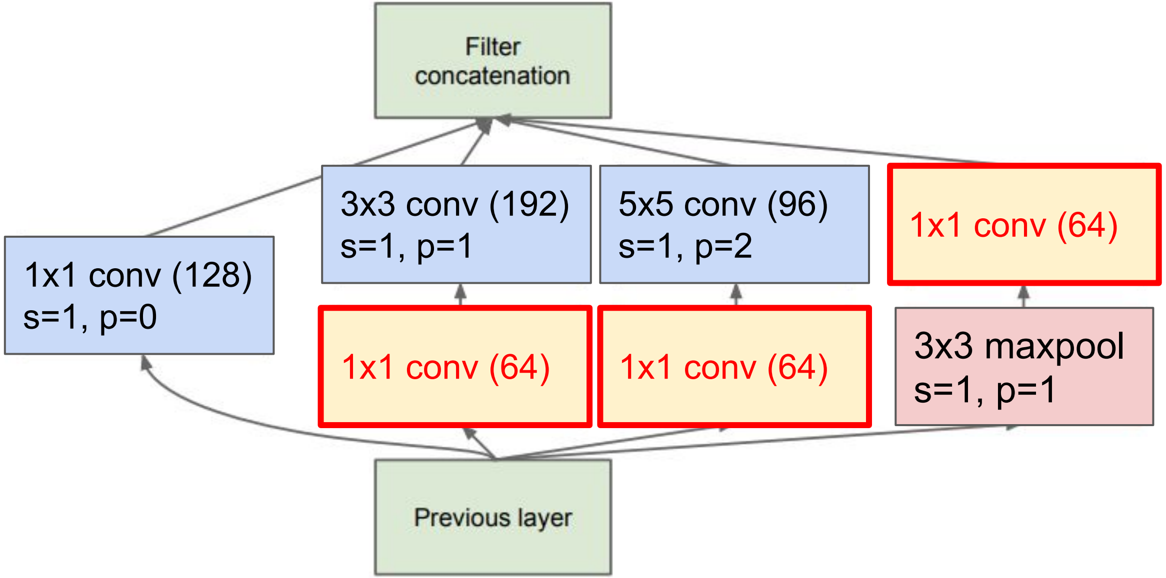
- Closer inspection
  - 1x1 convolution, 3x3 convolutions, 5x5 convolutions
    - Convolutions with different receptive fields.
    - Set to have the same spatial feature size
  - Max pooling for additional spatial abstraction

Problem: too large feature dimension
→ increase the parameter size in upper layers
→ increase the memory requirement



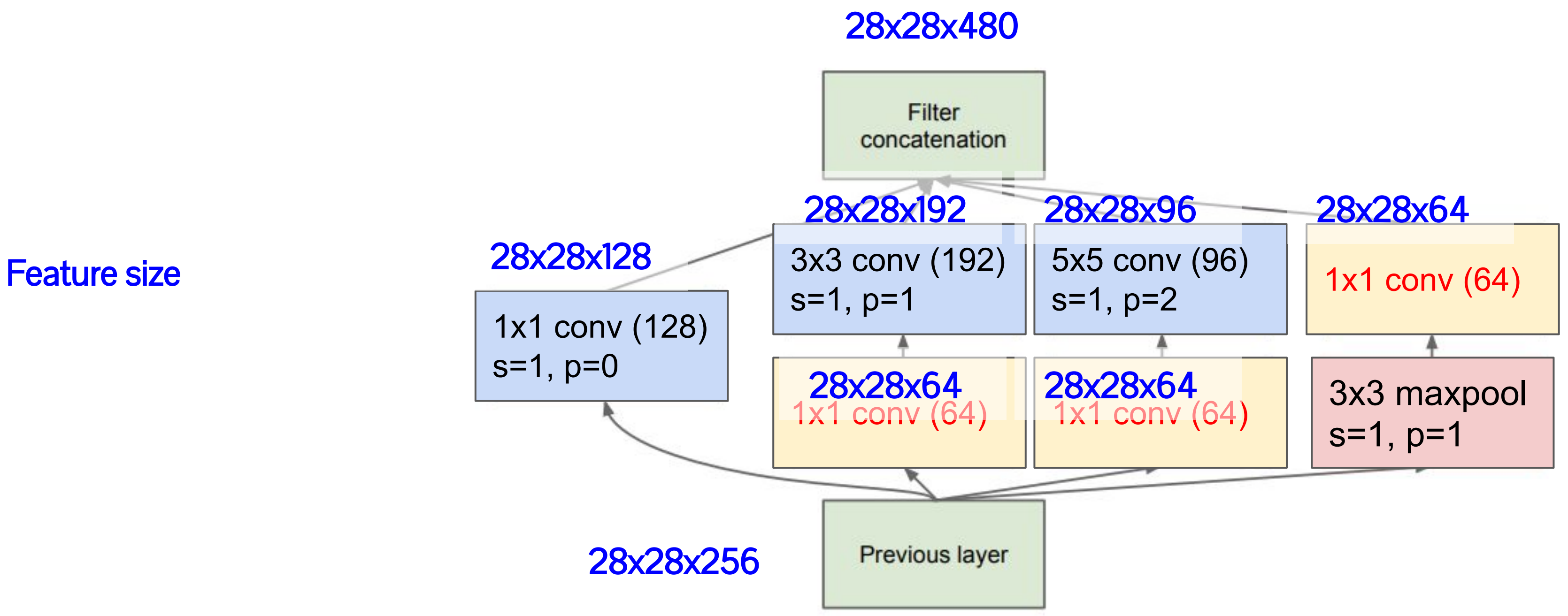
Inception module

- 1x1 convolution for dimensionality reduction
  - Insert 1x1 convs for every conv and pooling



Inception module

- 1x1 convolution for dimensionality reduction
  - Insert 1x1 convs for every conv and pooling

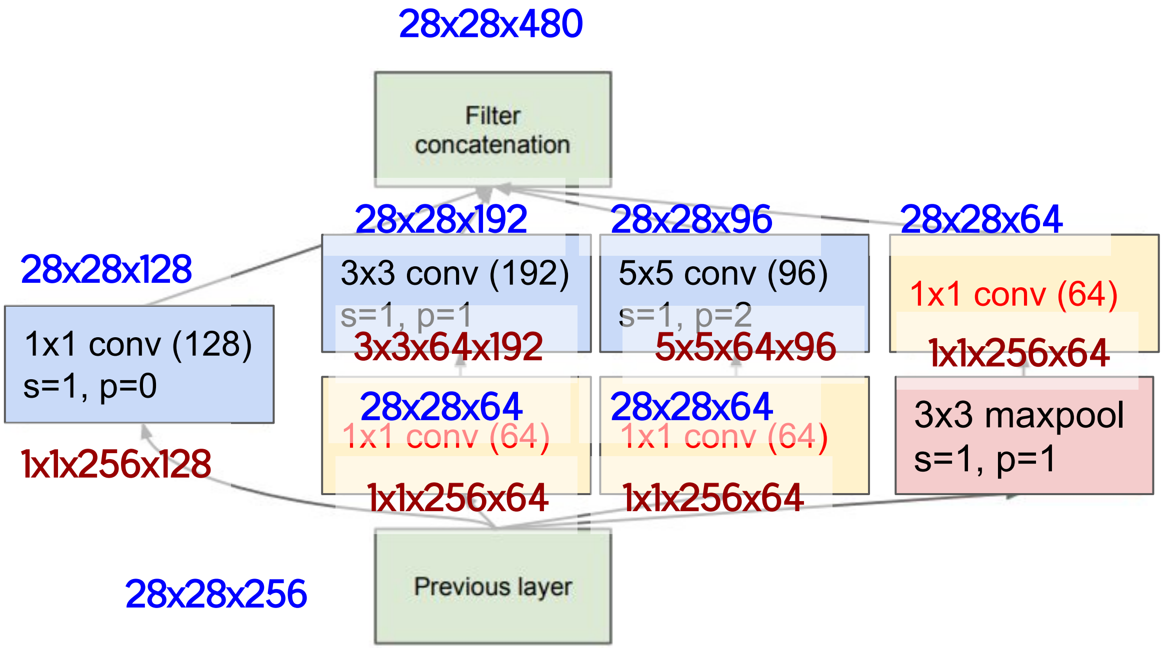


Inception module

- 1x1 convolution for dimensionality reduction
  - Insert 1x1 convs for every conv and pooling

Feature size

Parameter size



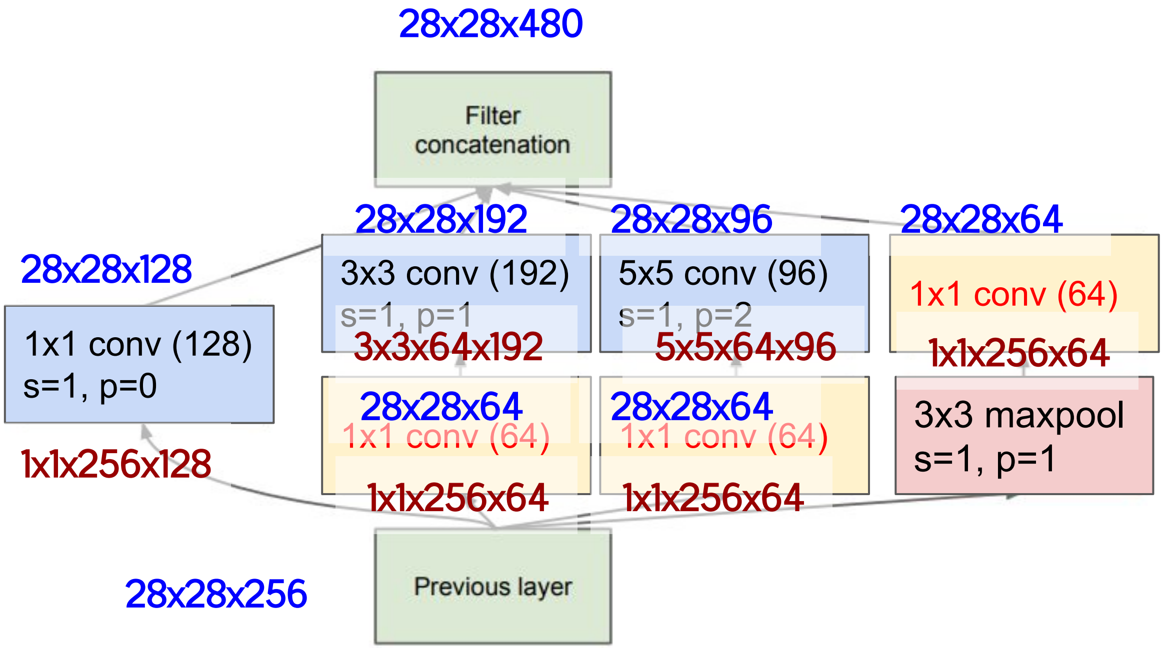
Inception module

- 1x1 convolution for dimensionality reduction
  - Insert 1x1 convs for every conv and pooling

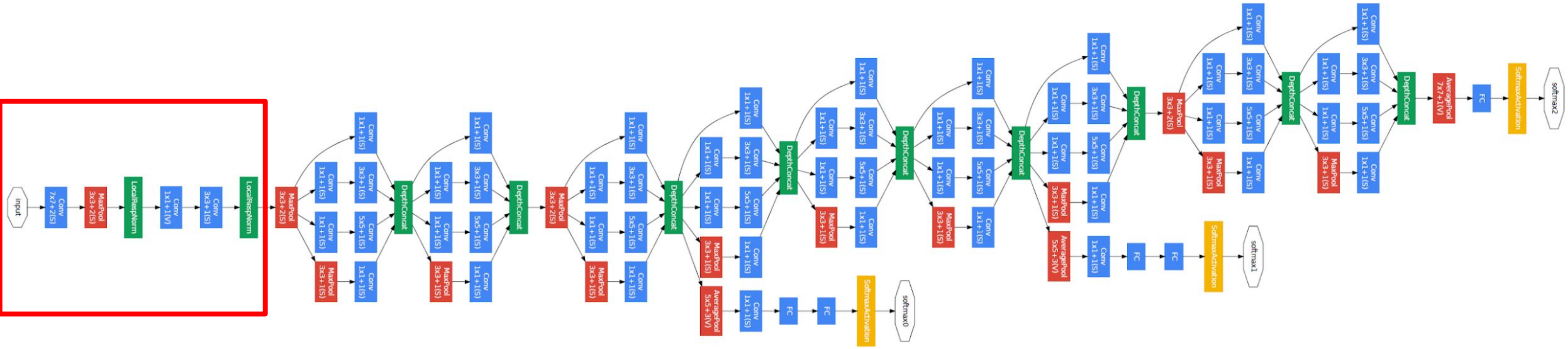
Feature size

Parameter size

Naive inception:
~ 1090k
Inception with bottleneck:
~ 330k

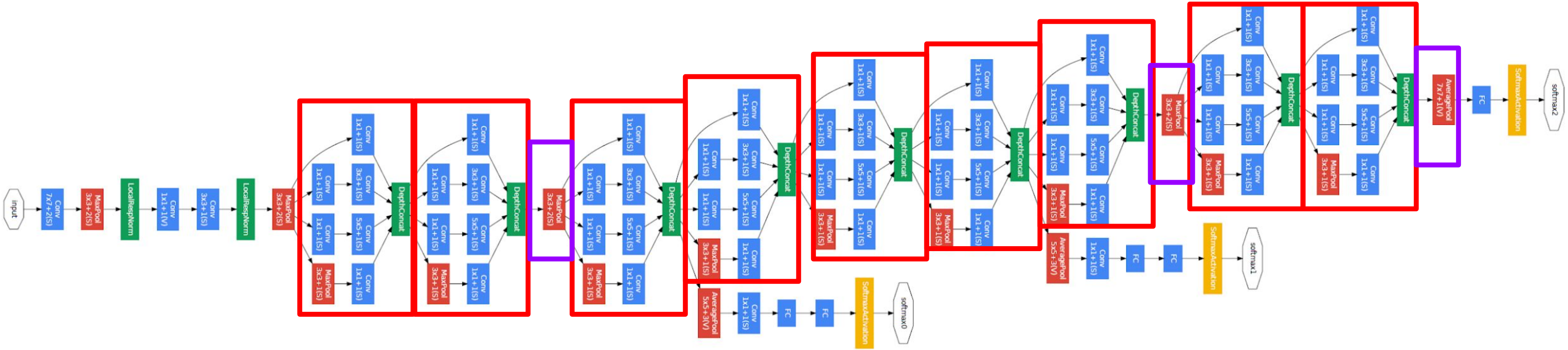


GoogleNet architecture



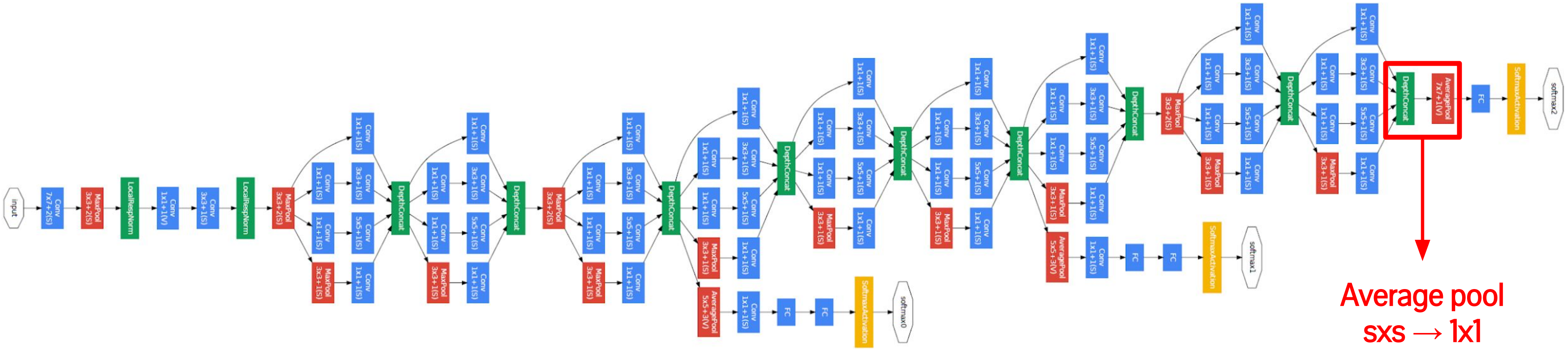
- A stack of simple convolution layers for initial layers

GoogleNet architecture



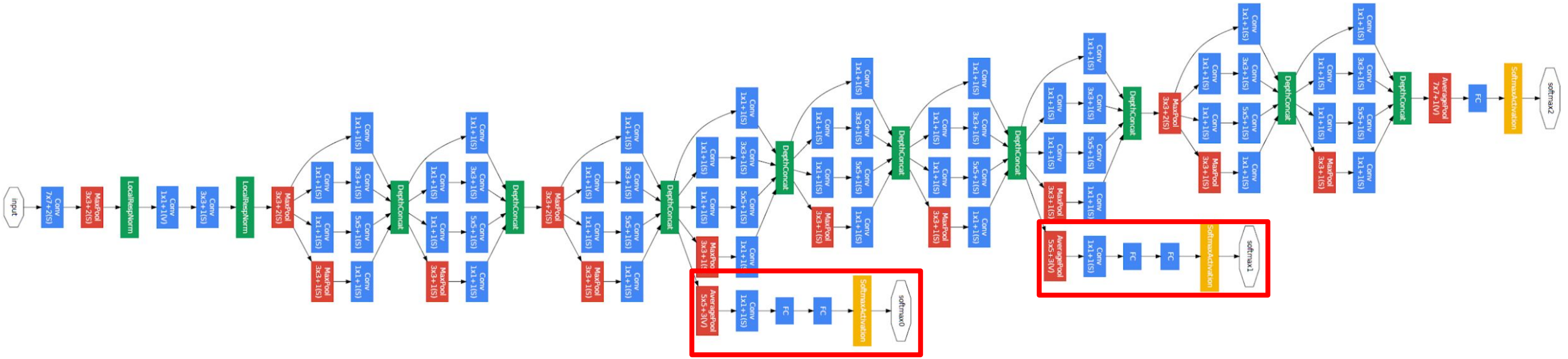
- A stack of simple convolution layers for initial layers
- A stack of inception modules in higher layers with occasional downsamplings

GoogleNet architecture



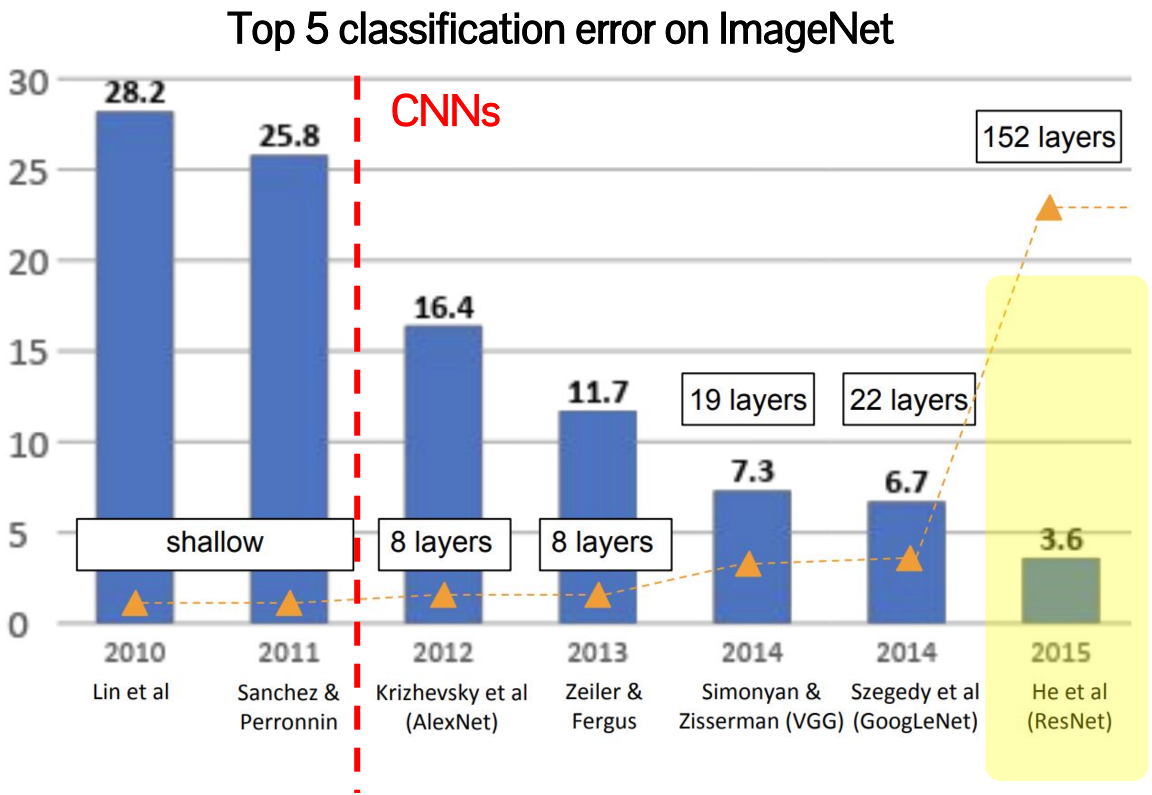
- A stack of simple convolution layers for initial layers
- A stack of inception modules in higher layers
- Average pooling to reduce the spatial feature dimension instead of FC

GoogleNet architecture



- A stack of simple convolution layers for initial layers
- A stack of inception modules in higher layers
- Average pooling to reduce the spatial feature dimension instead of FC
- Auxiliary loss (i.e. shortcuts) for strong gradient signals

Case study: CNN architectures for image classification



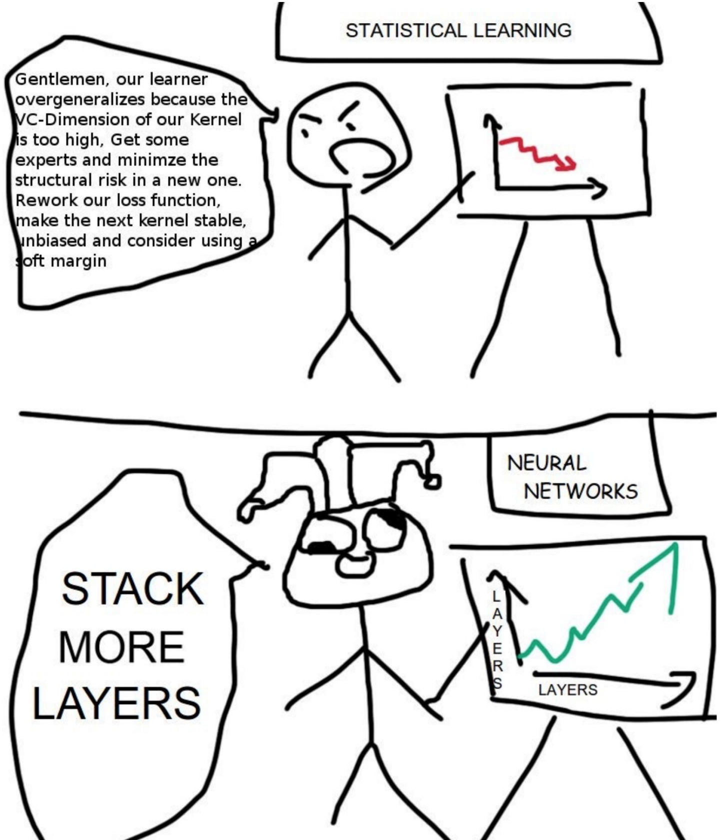
Resnet [Simonyan et al. 2013]:

Revolution in network depth
Substantial performance improvement

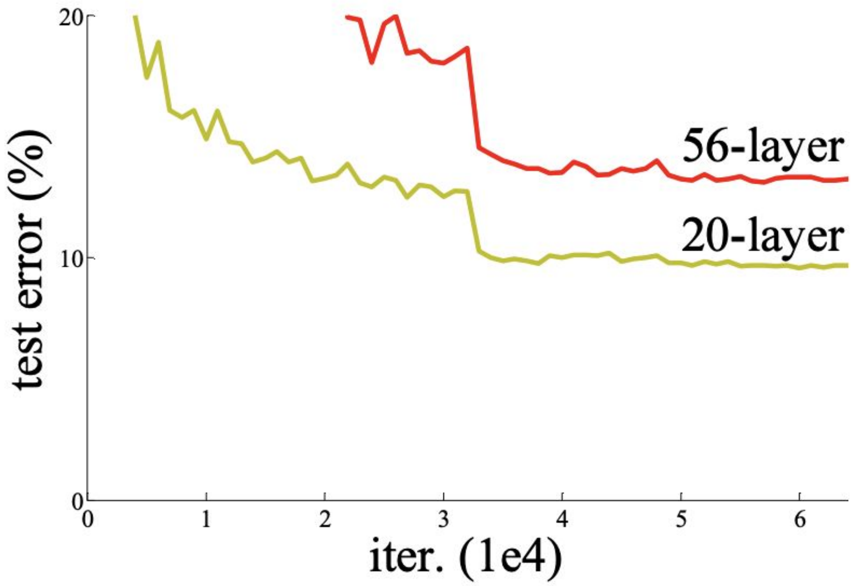
So far, the deeper network seems to be better

- AlexNet → VGGNet → GoogleNet
  (8 layers) (16 layers) (22 layers)
  (16.4%) (7.3%) (6.3%)

- How about deeper network?

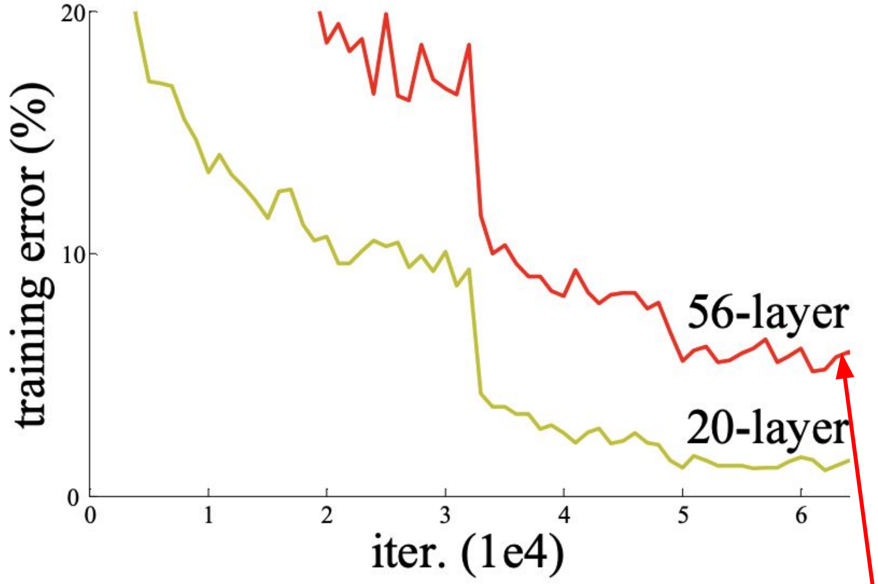


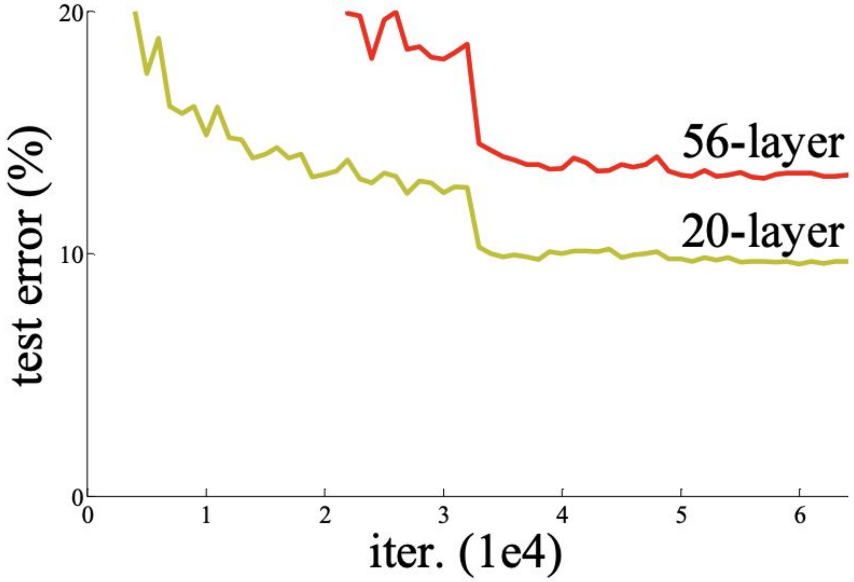
Depth vs. performance



Deeper network performs worse (higher test error).
Maybe an overfitting issue? (due to increased amount of params)

Depth vs. performance





The deeper model fits even worse on training data!
It's not an overfitting problem!

Why deeper network performs worse?

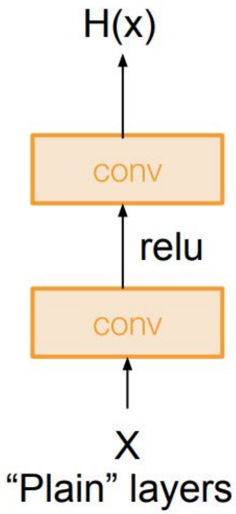
- In theory, deeper network should be at least as good as the shallow one in fitting the training data
- However, larger networks are much **difficult to optimize**
  - Potential problems in deeper networks
    - **Gradient vanishing** (gradients norm approaches near zero)

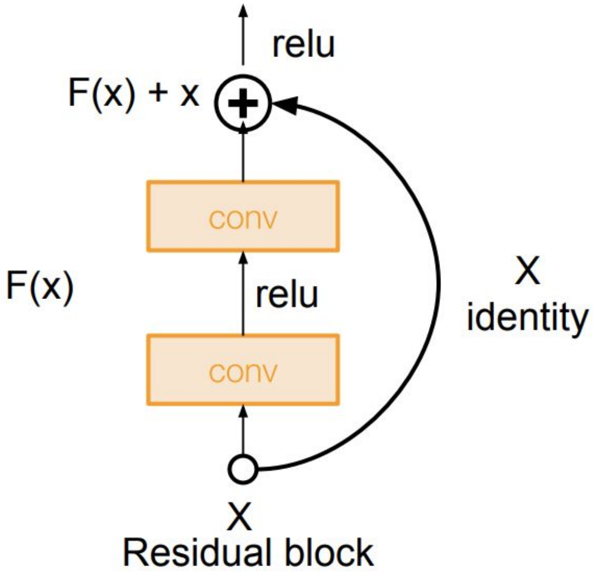
$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{h}^{(l)}} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \mathbf{h}^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{h}^{(L-1)}} \frac{\partial \mathbf{h}^{(L-1)}}{\partial \mathbf{h}^{(L-2)}} \cdots \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}$$

    - **Covariate shift** (small variations in lower layers lead to large variations in deeper layers)

Residual connection

- Main idea: add a shortcut connection that allows learning identity mapping

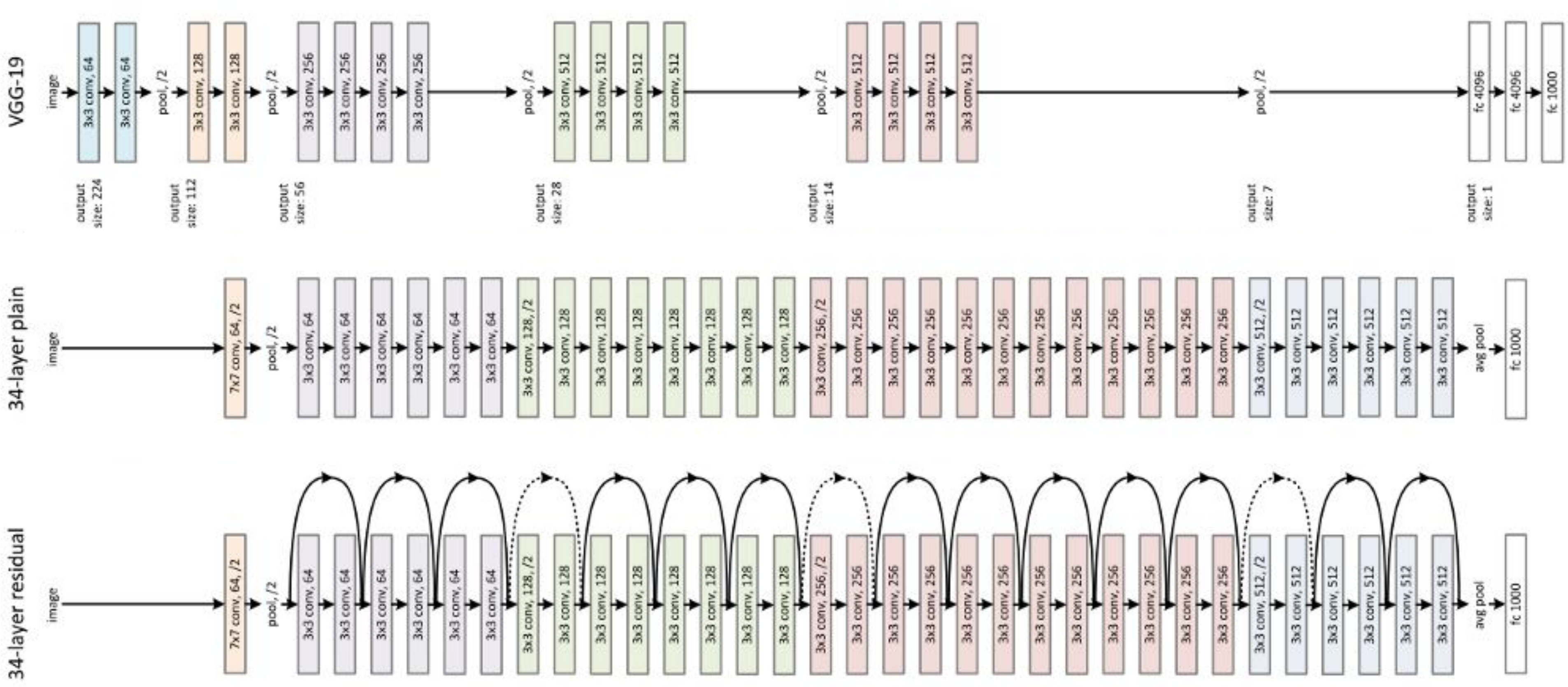




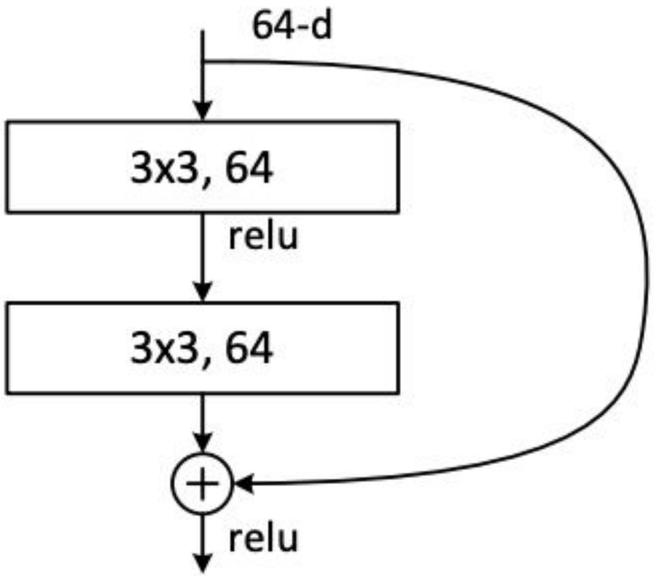
In degenerated case,
We can learn identity mapping by setting F(x)=0

Optimization is much easier by allowing “bypassing” gradients through identity connection

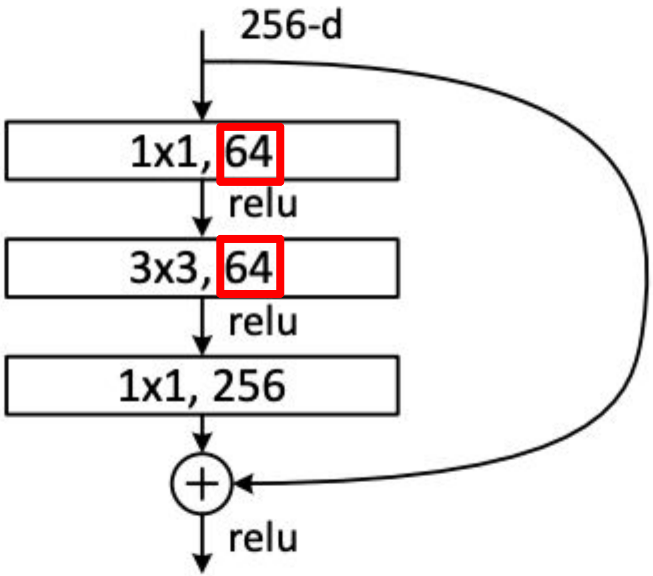
Residual network



Blocks of residual connection



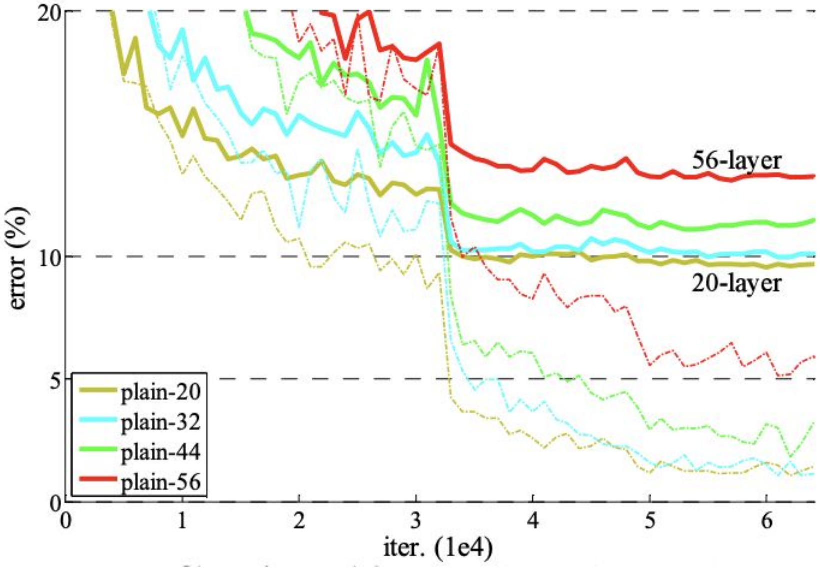
Plain residual block



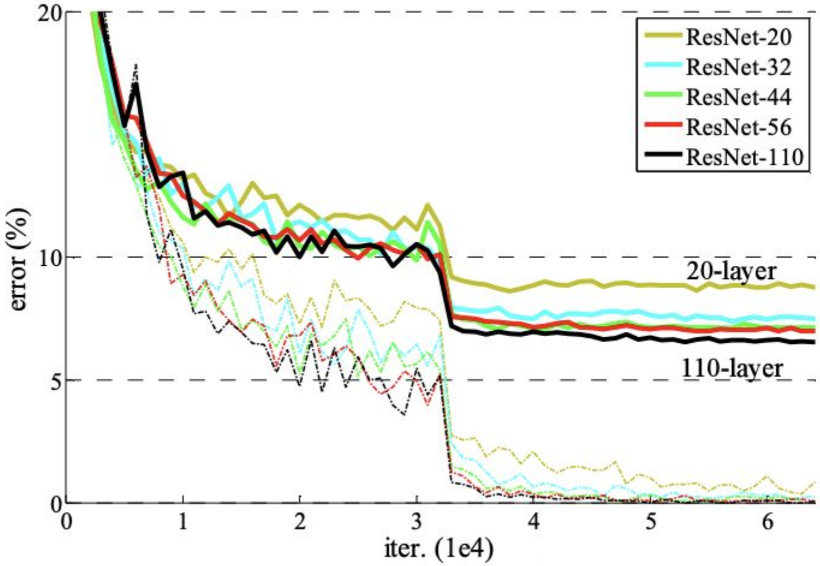
Residual block with “bottleneck”

Classification on Cifar 10 dataset

Plain network



Residual network



Dashed line: training
Solid line: testing

Today's agenda

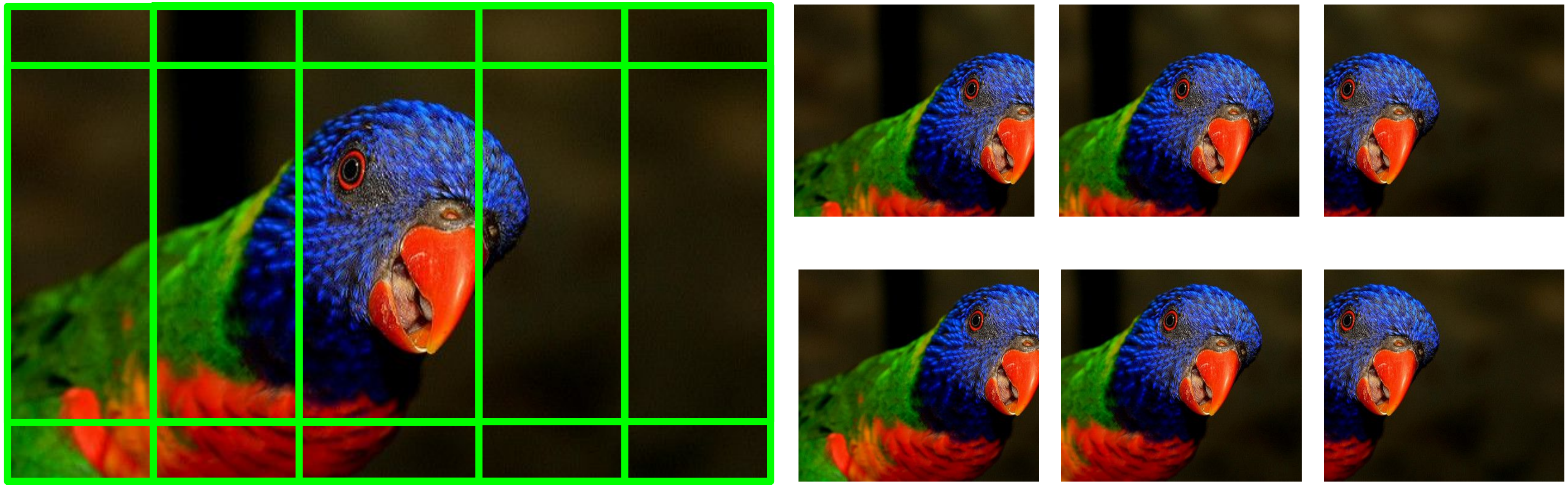
- CNN architectures for image classification
  - AlexNet, ZFNet, VGGNet, Resnet, DenseNet
- **Training tips for CNN**
  - data augmentation, fine-tuning

Data augmentation

- Increases the training data to prevent overfitting
- Approaches: horizontal flip

Data augmentation

- Increases the training data to prevent overfitting
- Approaches: Random crop



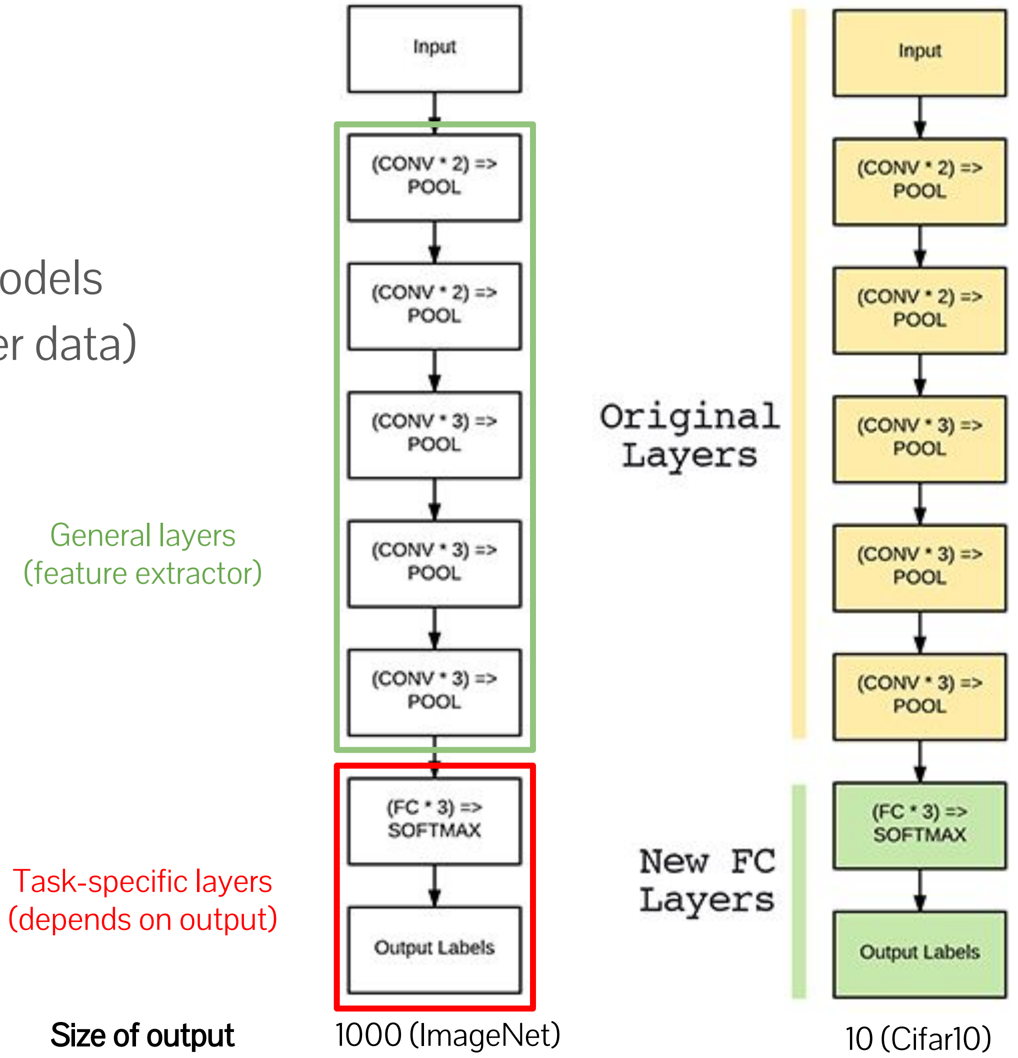
Increase the image size slightly more than input size, and crop the images at random locations

Data augmentation

- Increases the training data to prevent overfitting
- Approaches: random color jittering

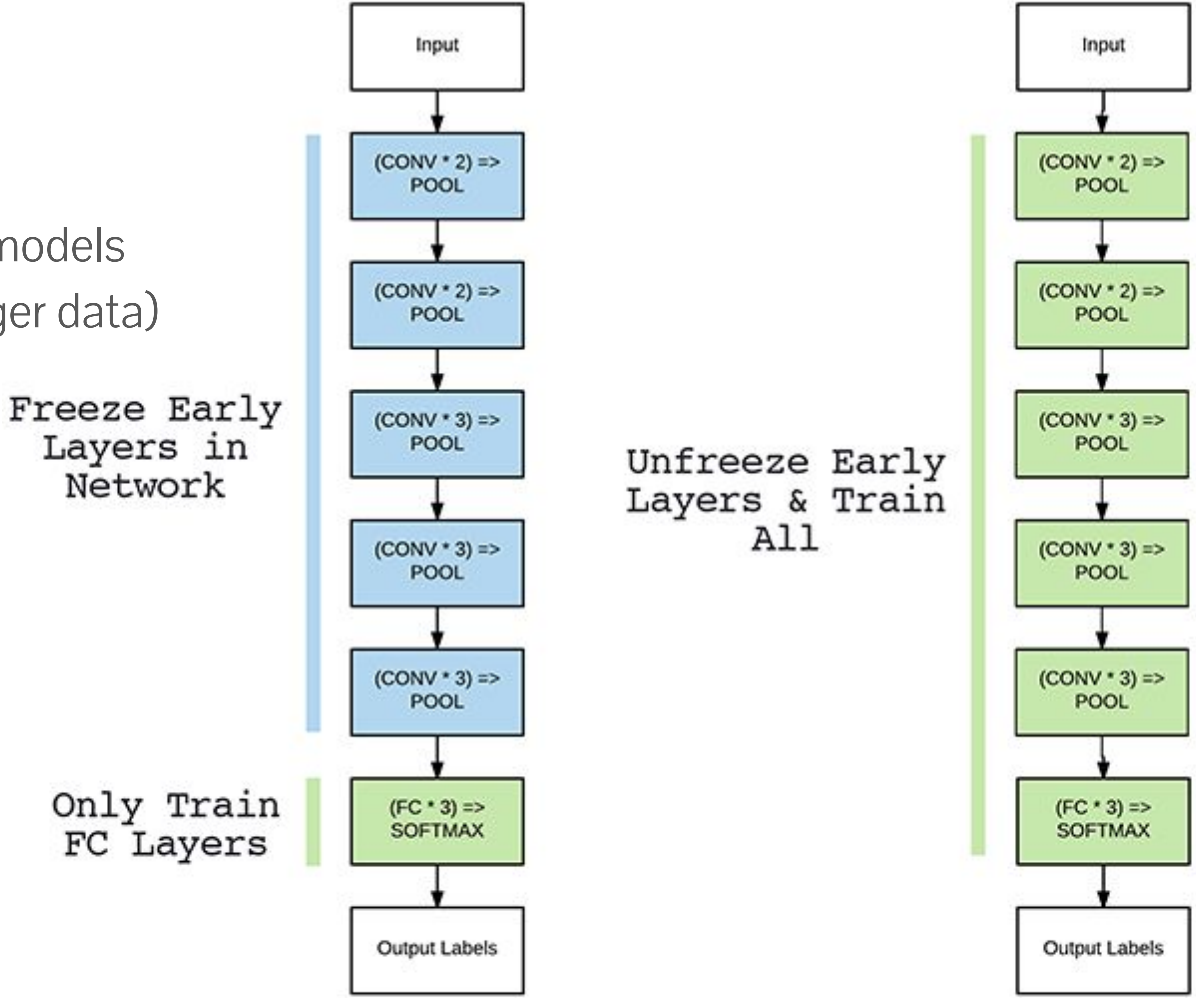
Fine-tuning

Transfer the weights from the models trained on other tasks (with larger data)



Fine-tuning

Transfer the weights from the models trained on other tasks (with larger data)



Fine-tuning

- Transfer the weights from the models trained on other tasks (with larger data)
  - In terms of optimization: initialize the parameters near the good local optima
  - Also related to transfer learning (i.e. transferring the knowledge from one experience to another)