DEEP LEARNING FOR VISUAL RECOGNITION

Lecture 7 – ConvNet Architectures

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

Today's agenda

- You will learn about convolutional neural network (CNN) architectures and transfer learning.
- Topics
  - Transfer learning
  - AlexNet
  - VGG Net
  - GoogLeNet (Inception)
  - ResNet
  - MobileNet
  - U-net
  - Siamese nets (and one-shot learning)

Last time – training neural networks

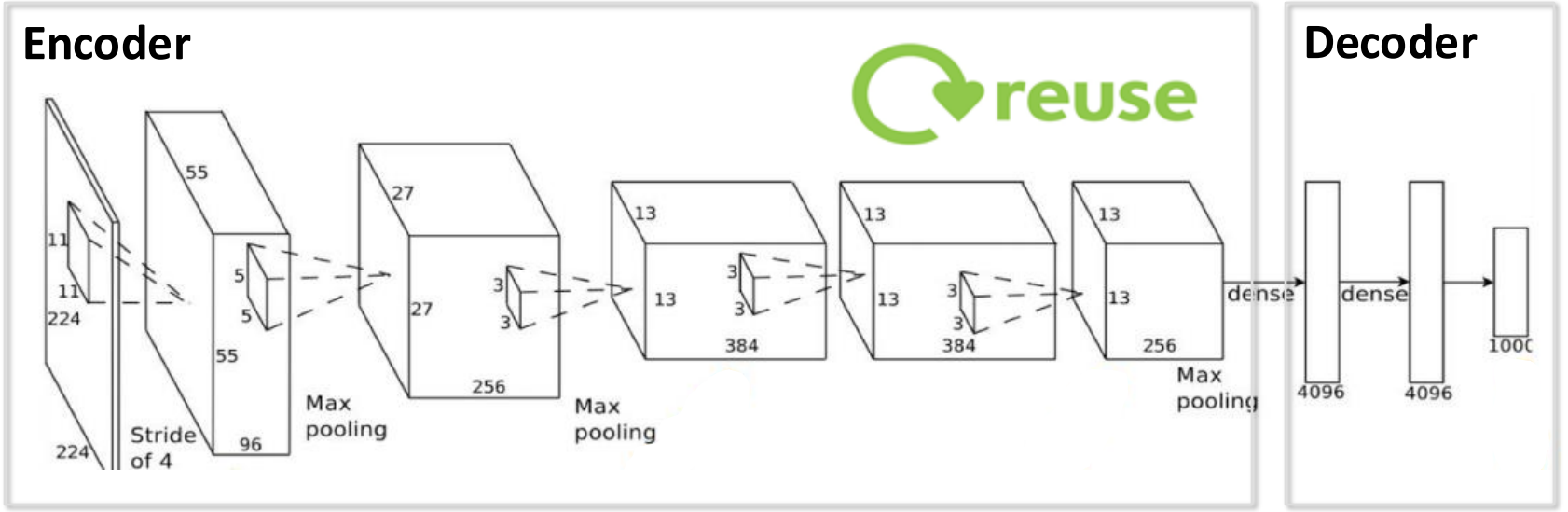
- One time setup:
  - Activation functions
  - Data preprocessing
  - Weight initialization
  - Batch normalization
  - Stochastic Gradient Descent (Momentum, AdaGrad, RMSProp, Adam)
  - Learning rate decay and cycling
  - Regularization: Early stopping, weight decay, dropout, data augmentation
- Training dynamics (self-study):
  - Babysitting the learning process
  - Hyperparameter search
  - Inspecting loss curves
  - Train/validation accuracy
  - Gradient/activation distributions per layer
  - First layer visualizations
- Evaluation:
  - Test on unseen data (test set)

One more thing: Transfer learning

- Myth: “You need a lot of data to train a ConvNet”
- **Not true**
- A **generic convolutional encoder** trained on ImageNet has **sufficient representational power** and generalization ability to perform visual discrimination tasks using simple linear classifiers [Donahue et al.].

Transfer learning

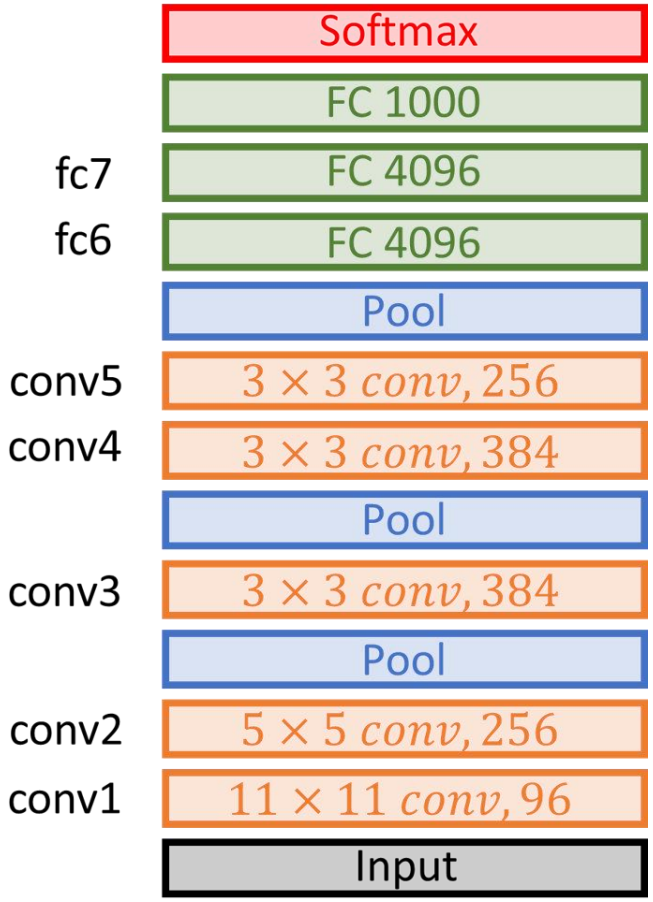
Why transfer learning?



- Training a ConvNet from scratch can take days
- Use pre-trained encoder trained on ImageNet
- Replace decoder and train it
- Training time: Typically less than 1 hour
- More details: How transferable are features in deep neural networks?

Transfer learning: Pretraining

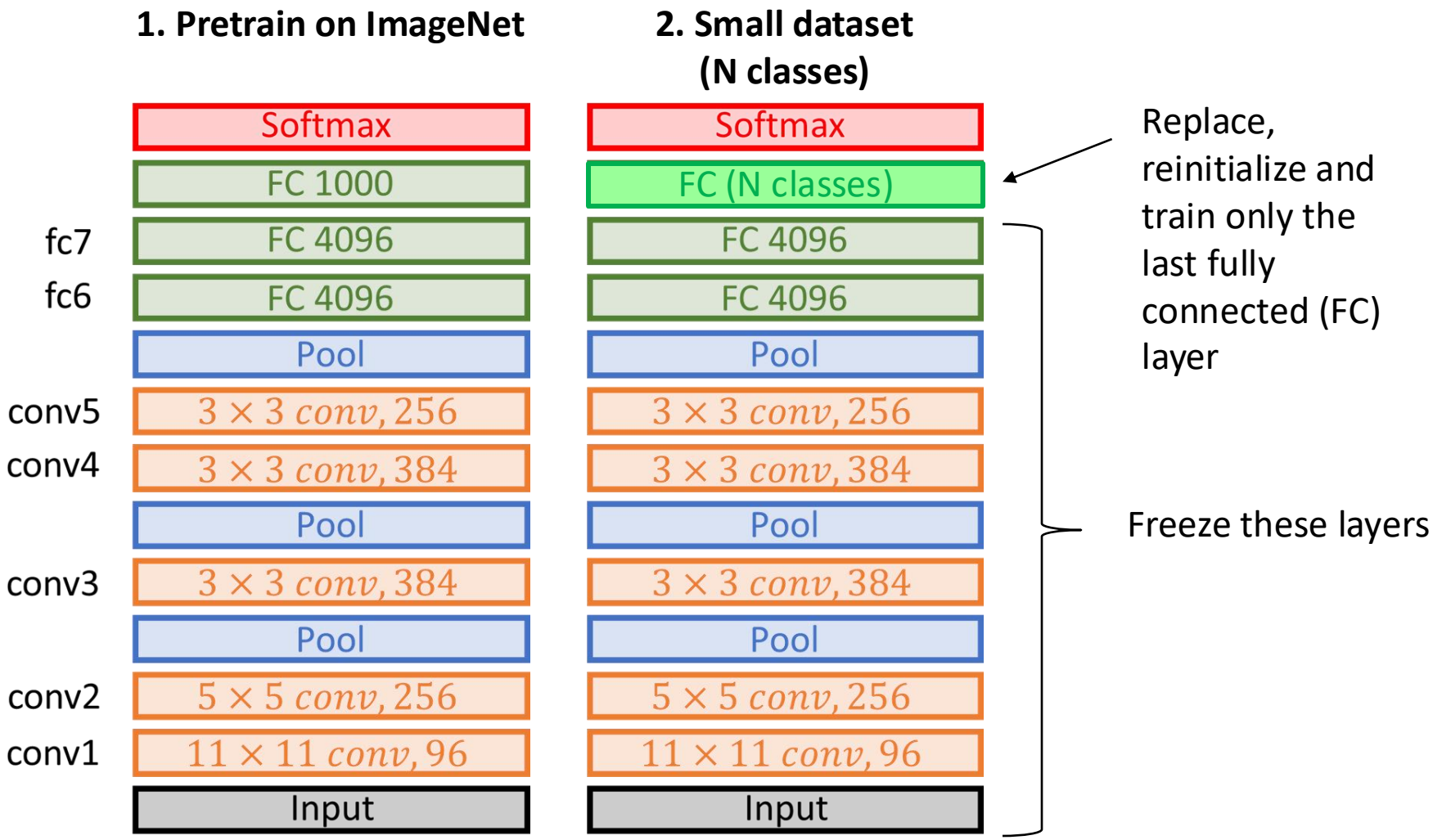
1. Pretrain on ImageNet



AlexNet

- Train from scratch on ImageNet
- Finishing ImageNet training with AlexNet takes several days on a single GPU.
- These guys finished ImageNet training with AlexNet in 11 minutes on 1024 CPUs.

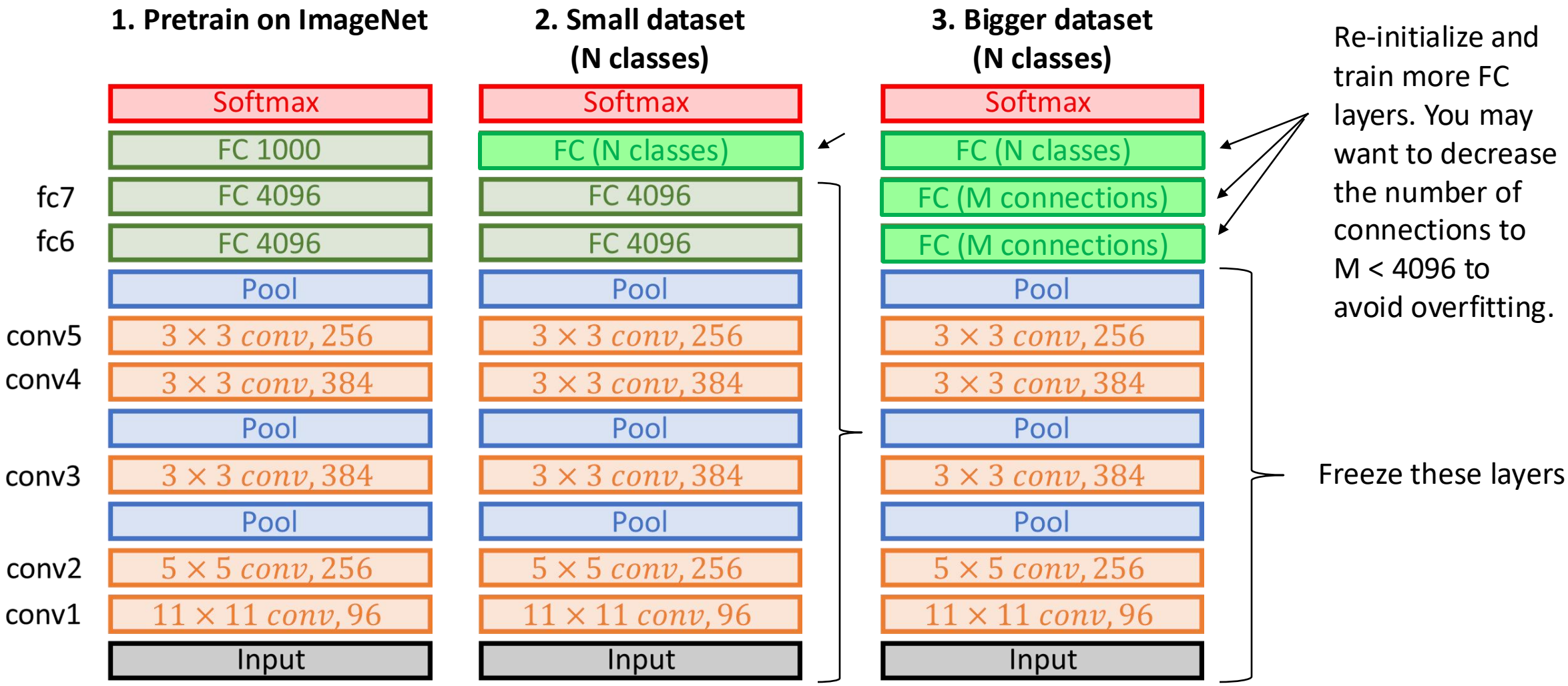
Transfer learning: Scenario 1



Replace, reinitialize and train only the last fully connected (FC) layer

Freeze these layers

Transfer learning: Scenario 2

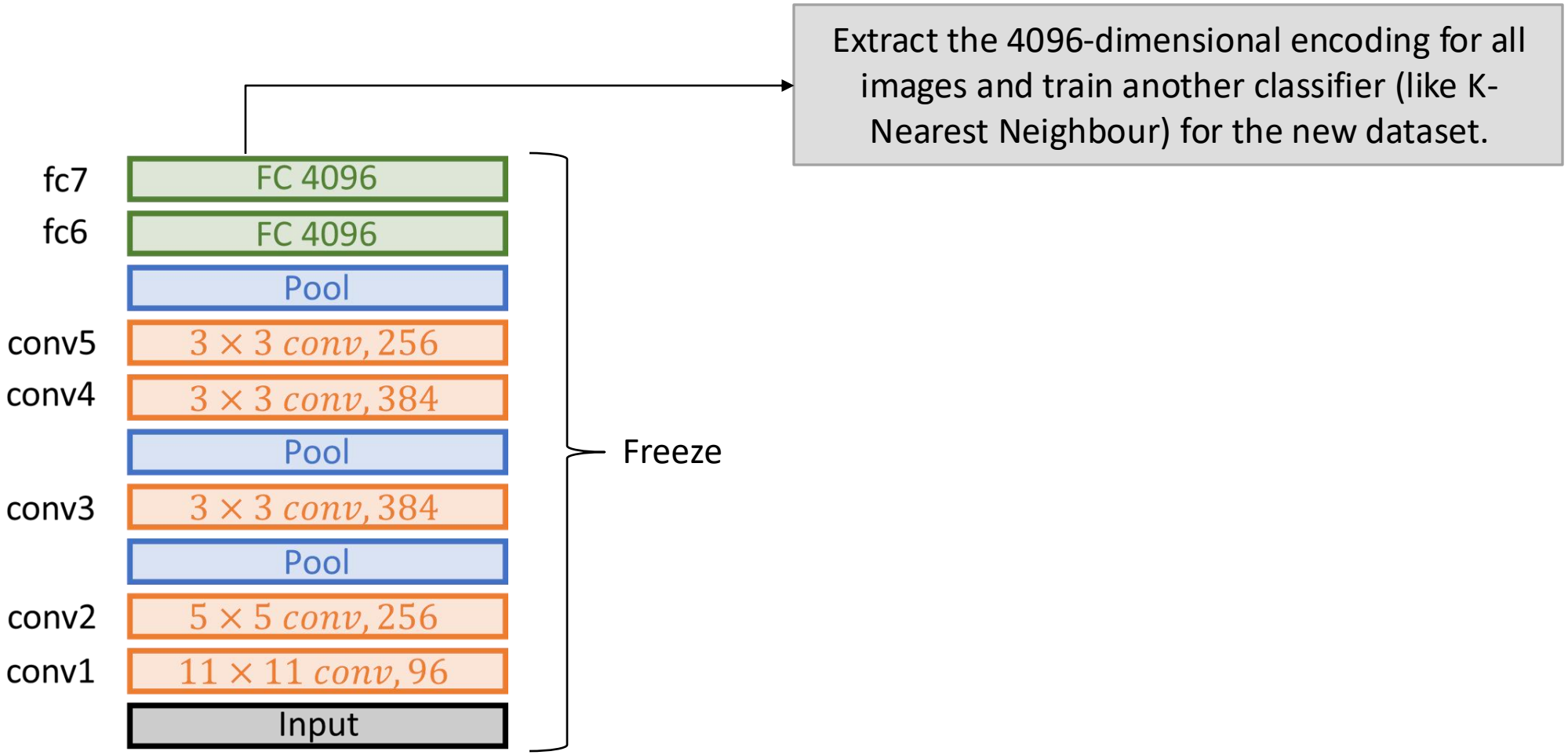


Scenario 1 and 2 explained

- CNNs that are trained on ImageNet will output vector with N = 1000 class probabilities.
- To train your own model with transfer learning, you always must replace the last FC layer such that it has **N = "number of classes in your dataset"**.
- **Scenario 1:** For small dataset, only replace the last layer (to avoid overfitting).
- **Scenario 2:** For bigger data sets, replace more FC layers (to increase model capacity).
- In both cases, the new FC layers are initialized with random weights.
- To begin with, train only the FC layers. In other words, freeze the weights of the pre-trained convolutional layers.
- After training the FC layers, you can often increase model performance by **fine-tuning** all layers, including the convolutional layers (this is explained in another slide below).
- See Task 9 in Lab 2: https://github.com/klaverhenrik/Deep-Learning-for-Visual-Recognition-2025/blob/main/Lab2_FeatureExtractionAndTransferLearning.ipynb

Transfer learning: Scenario 3

Fixed feature extractor

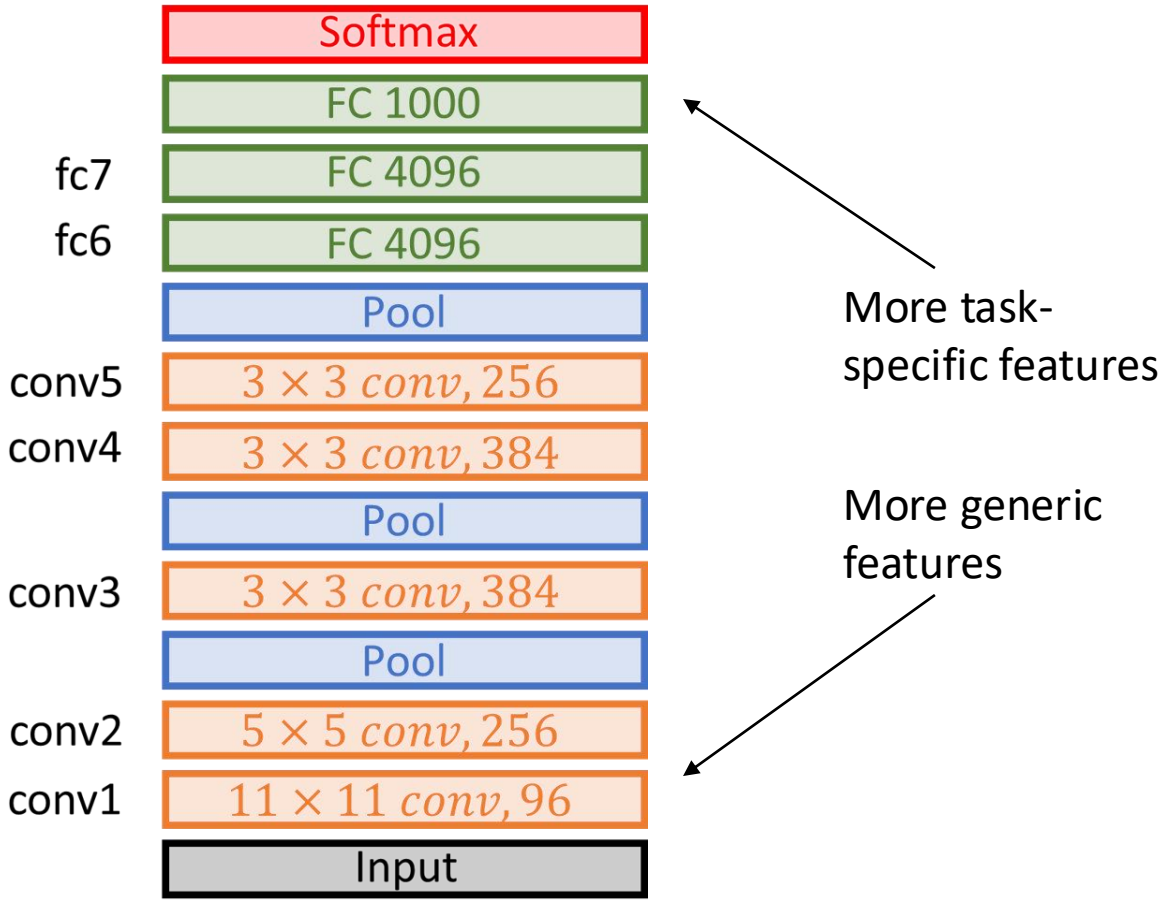


AlexNet

Scenario 3 explained

- Take a CNN pretrained on ImageNet.
- Remove one or more fully connected (FC) layers.
- Then treat the rest as a fixed feature extractor (**encoder**) for the new dataset.
- For AlexNet, removing the last FC layer would compute a 4096-dimensional vector that contains the activations of the hidden layer immediately before the classifier.
- Once you extract the 4096-dimensional codes for all images, you can train another classifier (like K-Nearest Neighbour) for the new dataset.
- **Disadvantages:** Features are fixed (and possibly sub-optimal), and end-to-end training not possible.
- See Task 8 of Lab 2: https://github.com/klaverhenrik/Deep-Learning-for-Visual-Recognition-2025/blob/main/Lab2_FeatureExtractionAndTransferLearning.ipynb

Transfer learning and fine-tuning

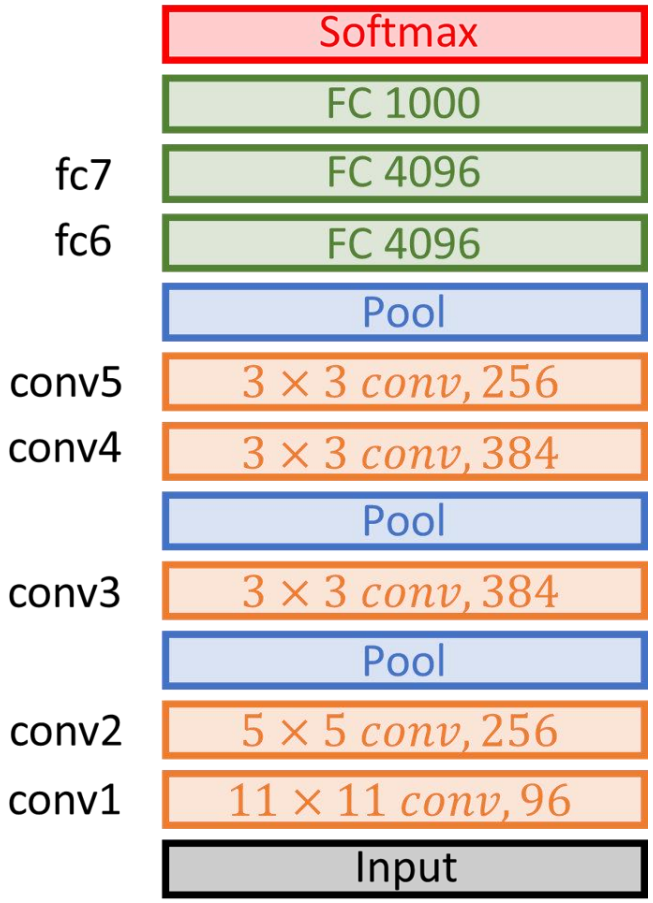


AlexNet

Observation:

Features in the early layers of a ConvNet contain more generic features (e.g., edge detectors or color blob detectors) that should be useful to many tasks, whereas later layers become progressively more specific to the details of the classes contained in the original dataset.

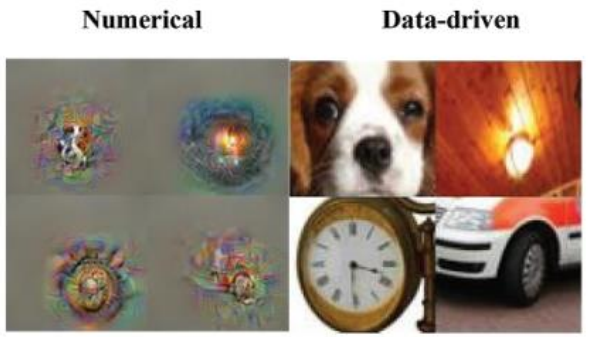
Transfer learning and fine-tuning



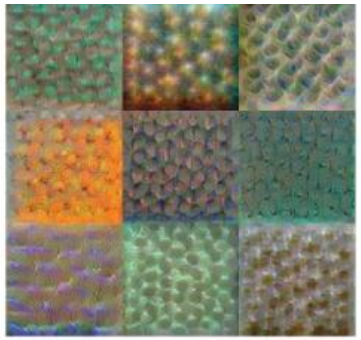
AlexNet

More task-specific features

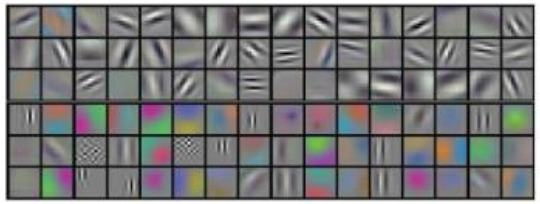
More generic features



Conv 5: Object Parts

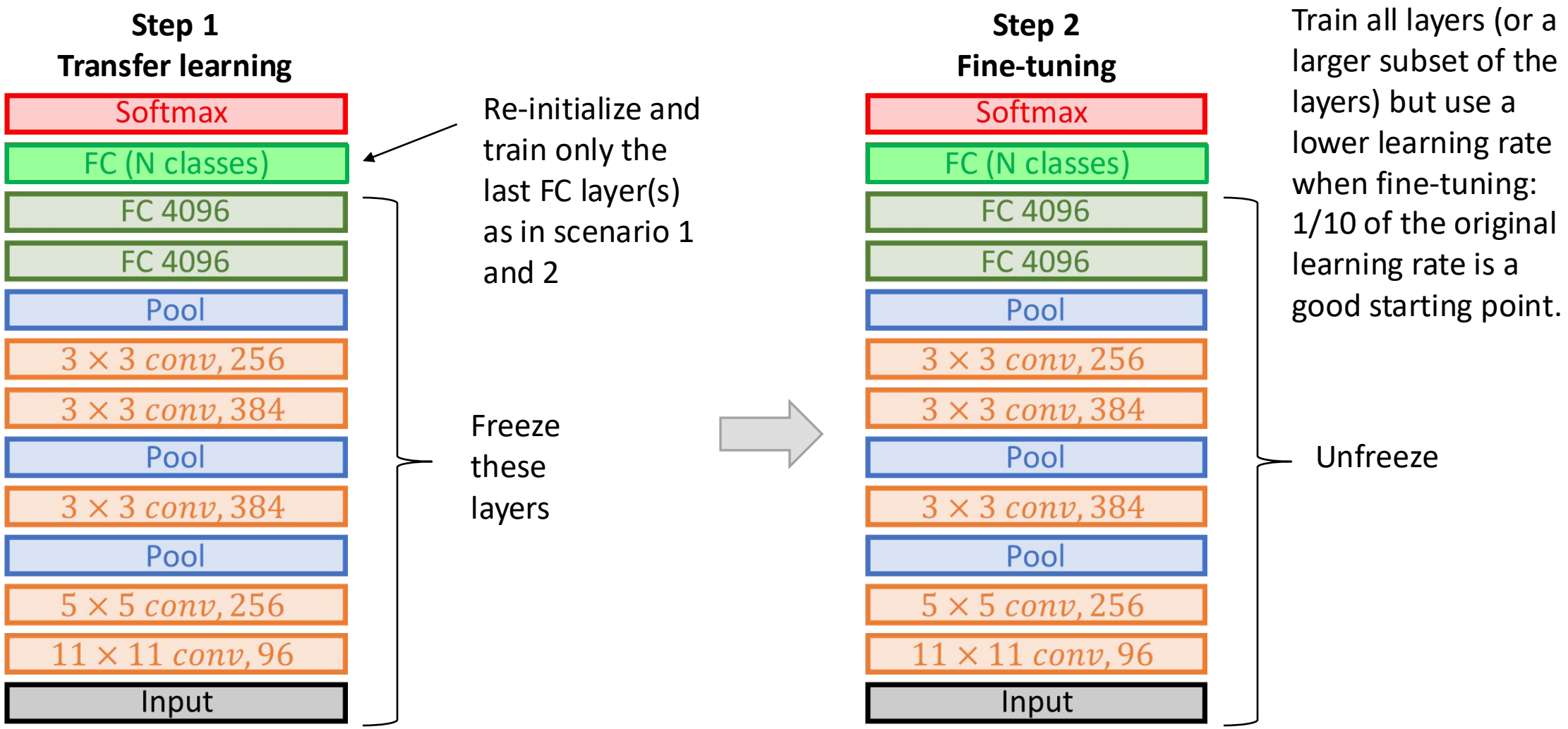


Conv 3: Texture

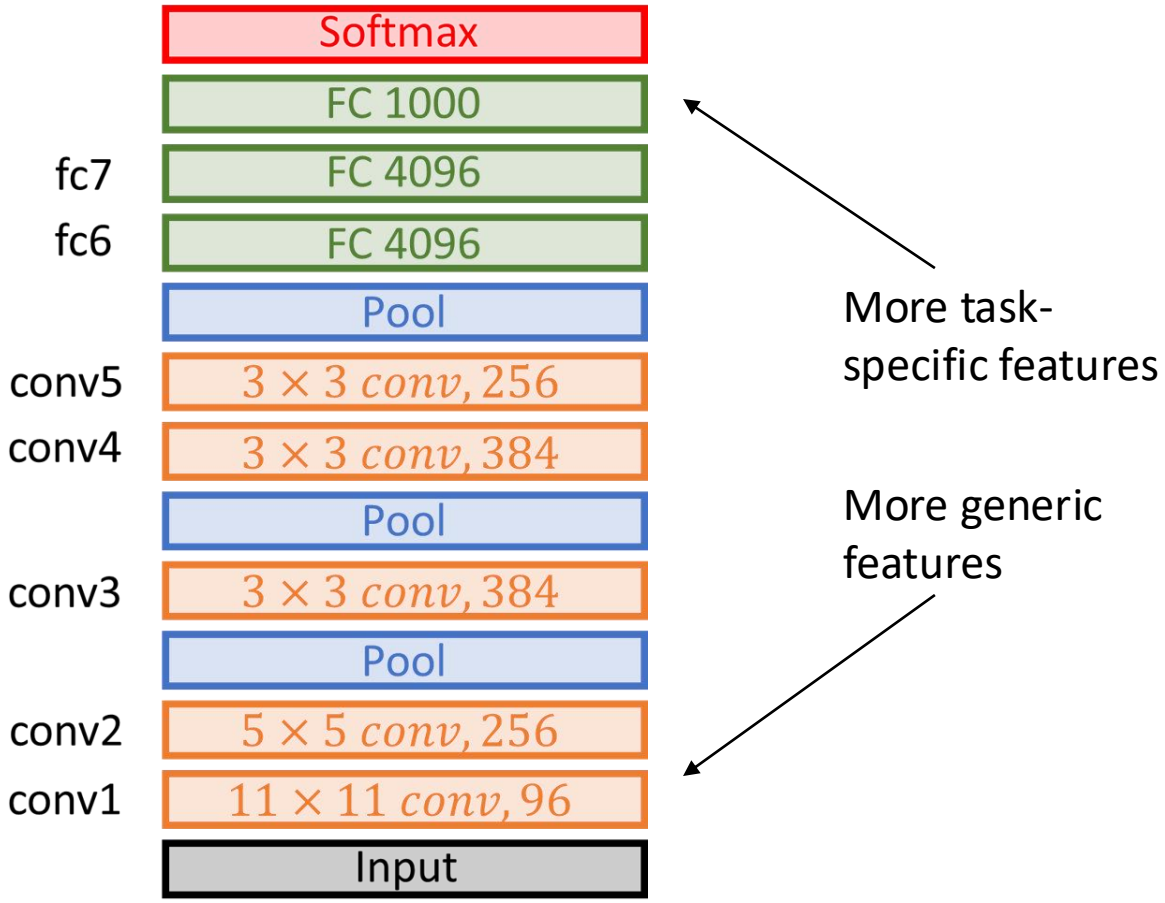


Conv 1: Edge+Blob

Transfer learning and fine-tuning



Transfer learning and fine-tuning



AlexNet

- **Step 1: Transfer learning**
  - Replace one or more FC layers (initialized with random values) to solve the new task.
  - Train only those layers for a few epochs, while freezing the remaining layers.
  - Sometimes transfer learning is good enough (i.e., if the generic features of the frozen layers have enough representational power to solve the new task).
- **Step 2: Fine-tuning**
  - What if the generic features of the frozen layers do not have enough representational power to solve the new task?
  - Unfreeze some or all the frozen layers and continue training.
  - Assumption: Features of previously frozen layers can be improved to better solve new task.

Fast.ai approach

- Jeremy Howards's approach:
  1. Download a pretrained convolutional encoder and add your own decoder (classifier)
  2. Train only the decoder for a few cycles (with cyclic learning rate). This is *transfer learning*.
  3. "Unfreeze" the encoder and run learning rate finder.
  4. Plot loss vs. learning rate and pick suitable learning rate range
  5. Train your model for another few cycles (train all layers, including convolutional base). This is called *fine-tuning*.

See Jeremy's guide:
https://youtu.be/Egp4Zajhzog

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

Then run `fit_one_cycle` 4 times and see how we go. And we have a 2% error rate. So that's pretty good. Sometimes it's easy for me to recognize a black bear from a grizzly bear, but sometimes it's a bit tricky. This one seems to be doing pretty well.

```
learn.fit_one_cycle(4)
```

Transfer learning

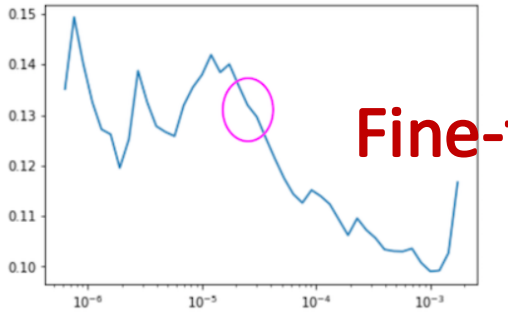
```
Total time: 00:54
epoch  train_loss  valid_loss  error_rate
1      0.710584    0.087024    0.021277    (00:14)
2      0.414239    0.045413    0.014184    (00:13)
3      0.306174    0.035602    0.014184    (00:13)
4      0.239355    0.035230    0.021277    (00:13)
```

```
learn.unfreeze()
```

Then we run the learning rate finder and plot it (it tells you exactly what to type). And we take a look.

```
learn.lr_find()
```

```
learn.recorder.plot()
```

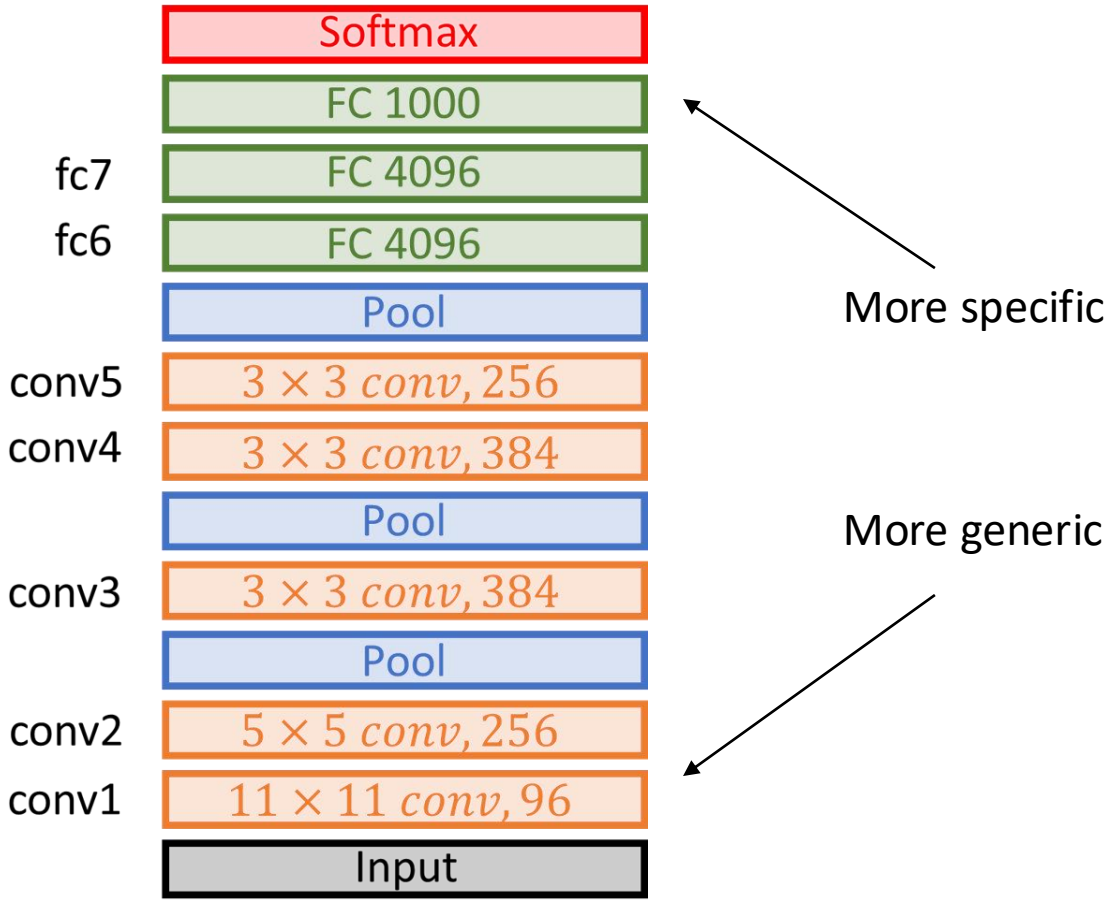


Fine-tuning

```
learn.fit_one_cycle(2, max_lr=slice(3e-5,3e-4))
```

```
Total time: 00:28
epoch  train_loss  valid_loss  error_rate
1      0.107059    0.056375    0.028369    (00:14)
2      0.070725    0.041957    0.014184    (00:13)
```

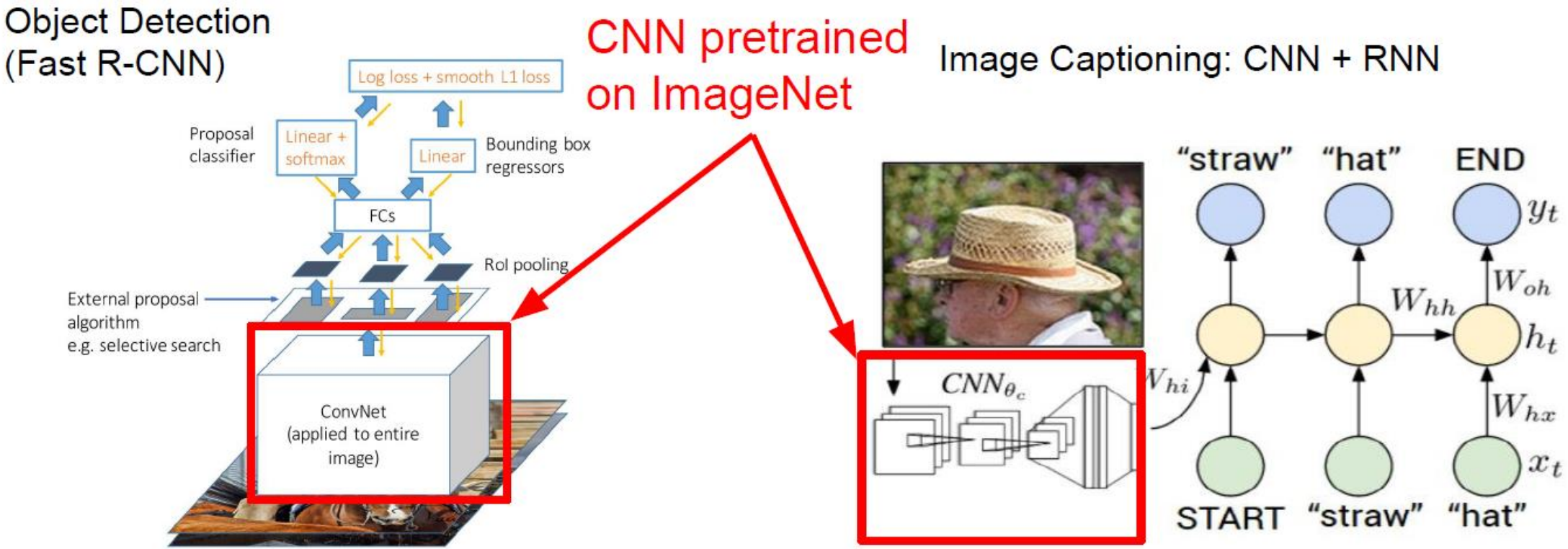
Transfer learning and fine-tuning



AlexNet

	Very similar dataset	**Very different dataset**
Very little data	Replace top-layer (last FC)	Replace more FC layers
A lot of data	Replace top-layer (last FC) + fine-tune a few layers	Replace more FC layers + fine-tune a large number of layers

Transfer learning is the norm



Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Model Zoo

- Deep learning frameworks provide a “Model Zoo” of pretrained models, so you don’t need to train your own.
- Keras: https://keras.io/applications/
- PyTorch: https://docs.pytorch.org/vision/main/models.html
- TensorFlow: https://github.com/tensorflow/models

When to design your model from scratch?

- Most pretrained models were trained on huge datasets, like ImageNet, with up to 1000 classes.
- This is obviously an advantage.
- However, these pretrained models also have very high capacity (i.e., many parameters), which does make them prone to overfitting and difficult to train from scratch.
- Many real-world problems are limited to just a few classes and therefore require models with much smaller capacity.
- Also, your dataset could be very different from ImageNet, and you would not benefit from using a pre-trained model.
- Therefore, **it sometimes works better to design and train your networks from scratch**, because you can make them much smaller; so, they will be faster to train and less prone to overfitting.
- Most likely, the best solution for you is something in between: **Use a pretrained encoder and built your own decoder on top of it that is smaller than the original** (because your model does not have to learn to distinguish between 1000 classes).

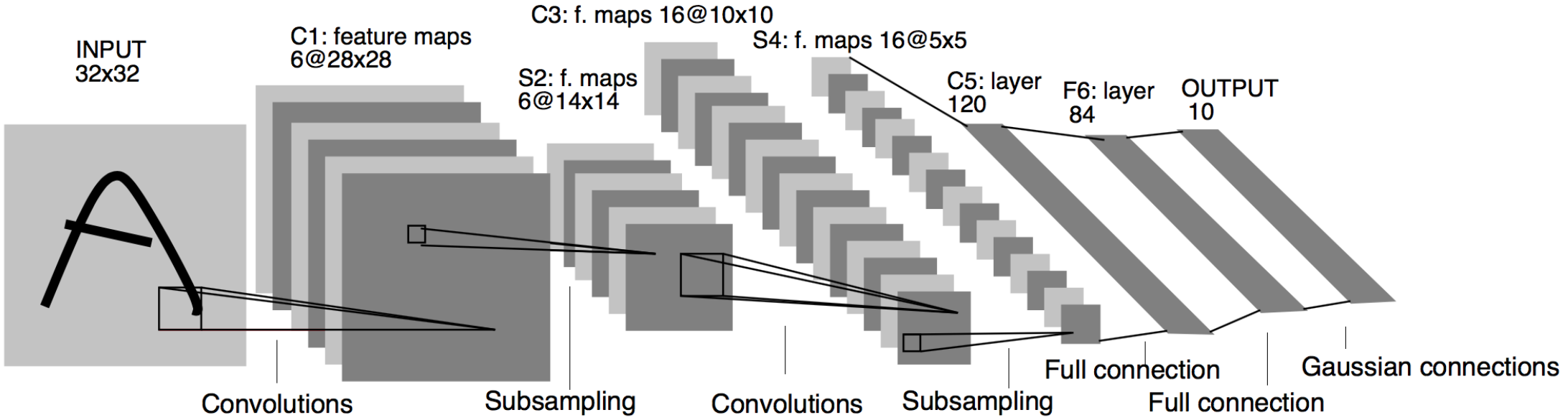
Today: CNN architectures

- AlexNet
- VGG Net
- GoogLeNet (Inception)
- ResNet
- MobileNet
- U-Net
- Siamese nets (and one-shot learning)

AlexNet

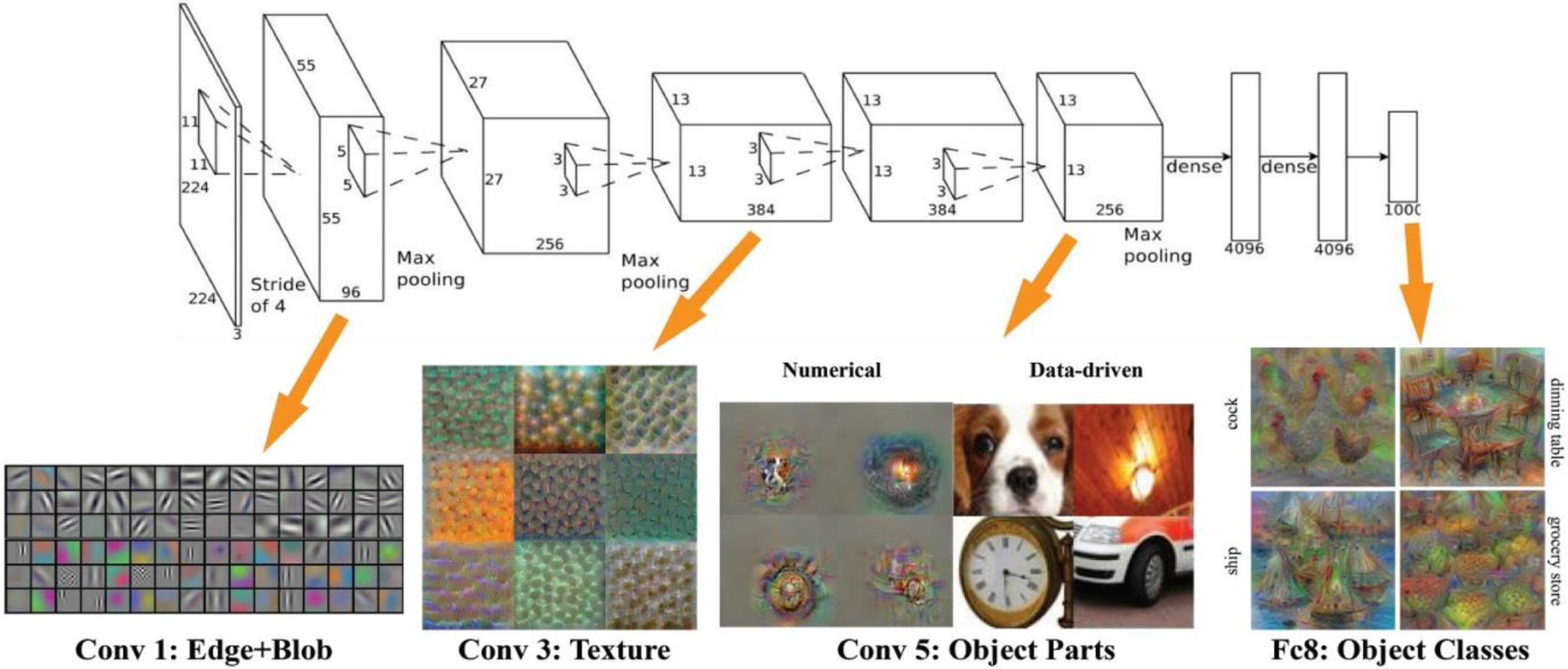
Review: LeNet-5

- First-ever CNN (1998)
- Conv filters were 5x5, applied at stride 1.
- Subsampling (Pooling) layers were 2x2 applied at stride 2
- Architecture is [CONV-POOL-CONV-POOL-FC-FC]



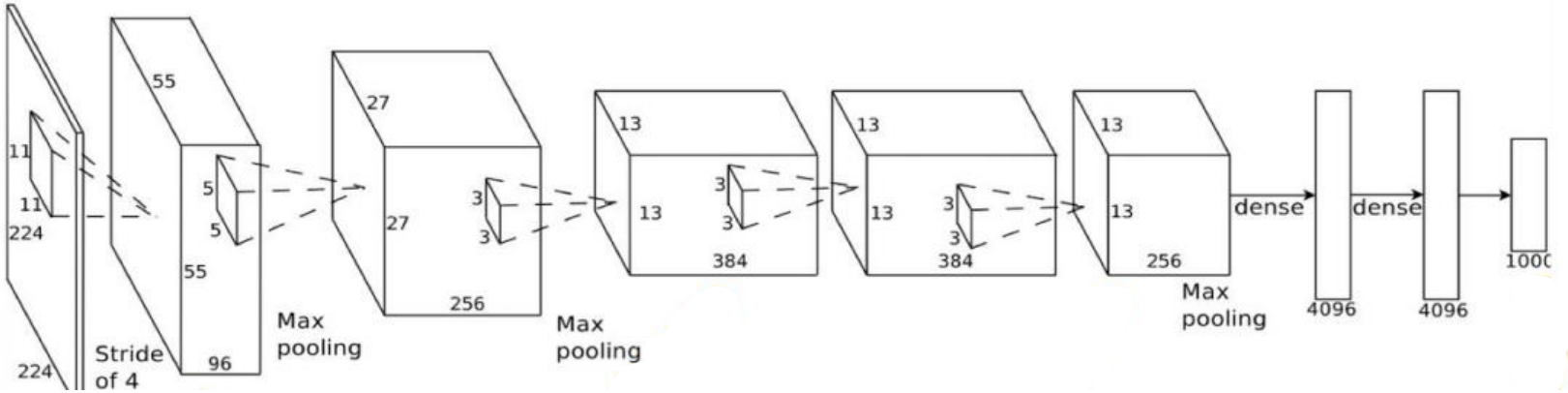
https://medium.com/@pechyonkin/key-deep-learning-architectures-lenet-5-6fc3c59e6f4

AlexNet architecture



https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

AlexNet architecture



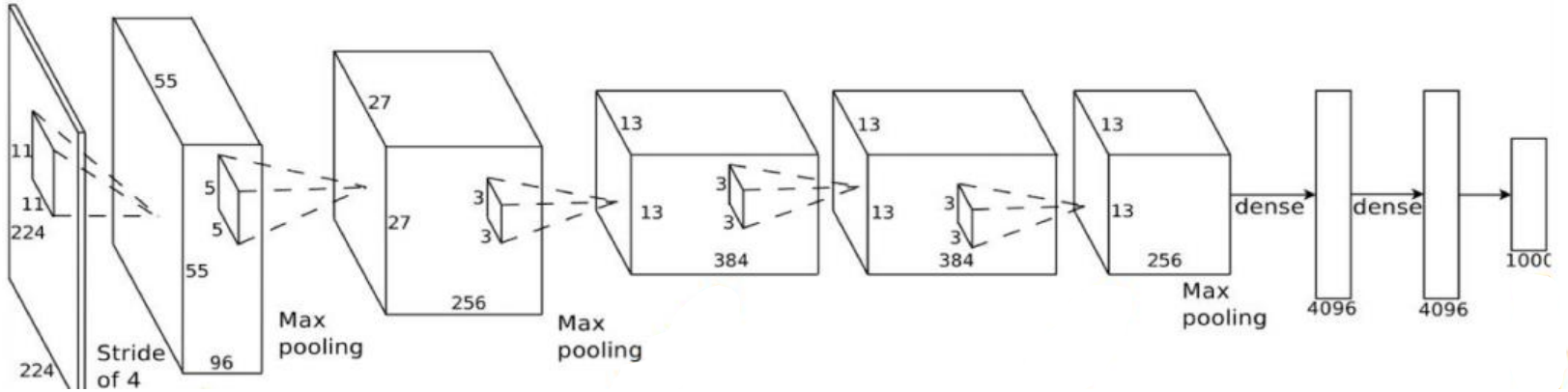
Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

Input: 227 x 227 x 3 image

- Note that the paper mentions the network inputs to be 224×224, but that is a mistake, and the numbers make sense with 227×227 instead.
- Random crops of size 227×227 and their horizontal reflections generated from inside 256×256 image (= data augmentation).
- If grayscale, convert RGB by replicating the single channel 3 times.

AlexNet architecture

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



Input: 227 x 227 x 3 image

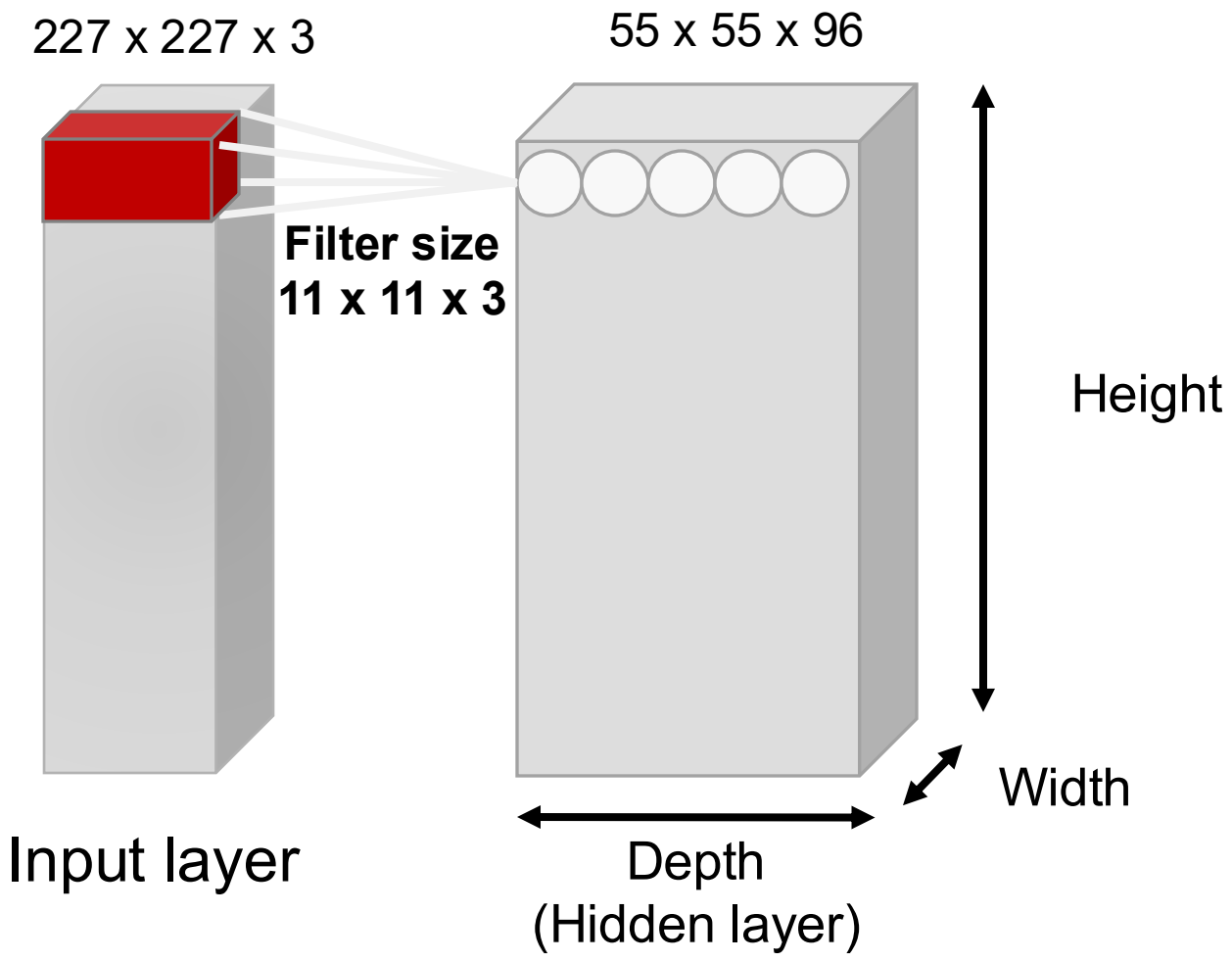
First layer (CONV1): 96 filters of size 11x11x3 applied with stride = 4

Q: What is the output volume size? Hint: (227-11)/4+1 = 55
A: Output volume is 55 x 55 x 96

Recall

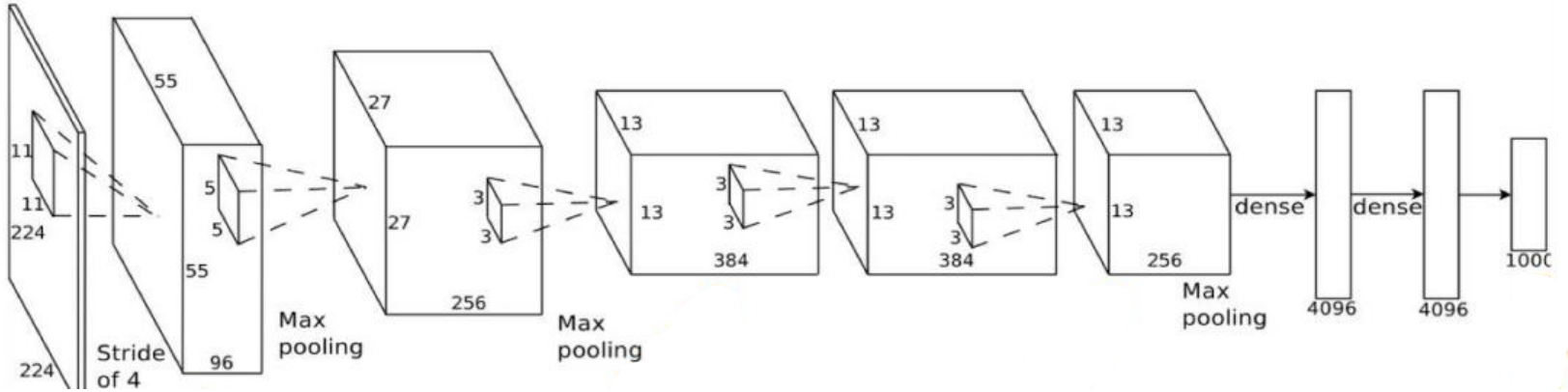
We can think of the output as a multi-dimensional image with N = 96 channels

Each channel corresponds to a **feature map** or activation map and is produced by one filter.



AlexNet architecture

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



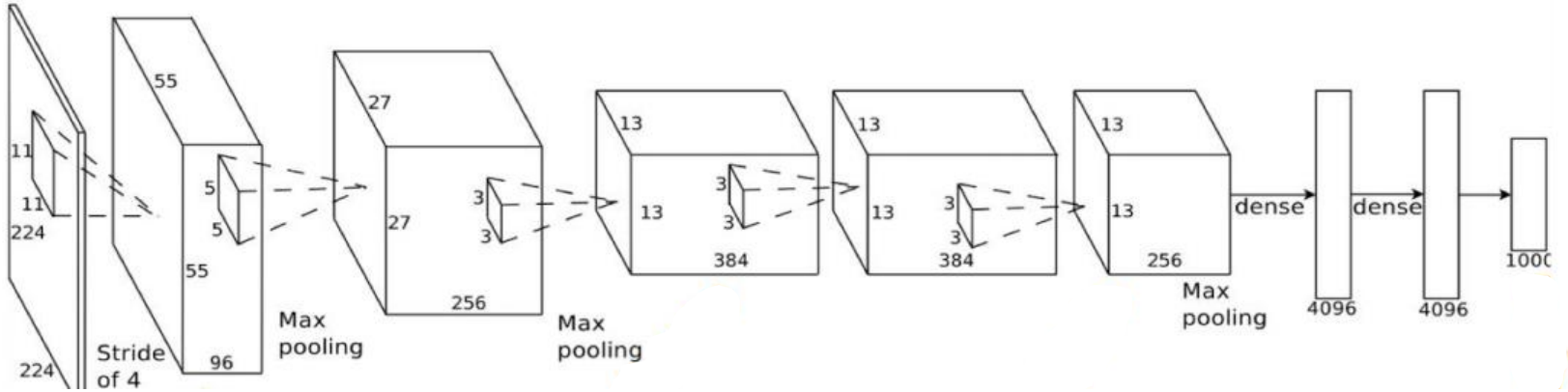
Input: 227 x 227 x 3 image

First layer (CONV1): 96 filters of size 11x11x3 applied with stride = 4

Q: What is the total number of parameters in the first layer?
A: Number of parameters is 11 x 11 x 3 x 96 (filters) + 96 (biases) = 35K

AlexNet architecture

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



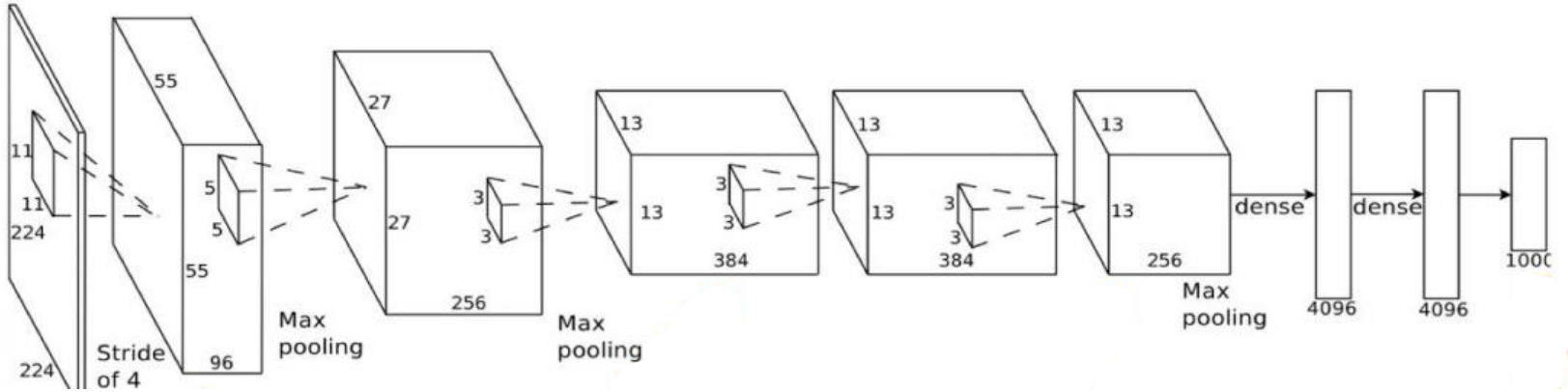
Input: 227 x 227 x 3 image
After CONV1: 55 x 55 x 96

Second layer (POOL1): 3x3 filters applied with stride = 2

Q: What is the output volume size? Hint: (55-3)/2+1 = 27
A: Output volume is 27 x 27 x 96

AlexNet architecture

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

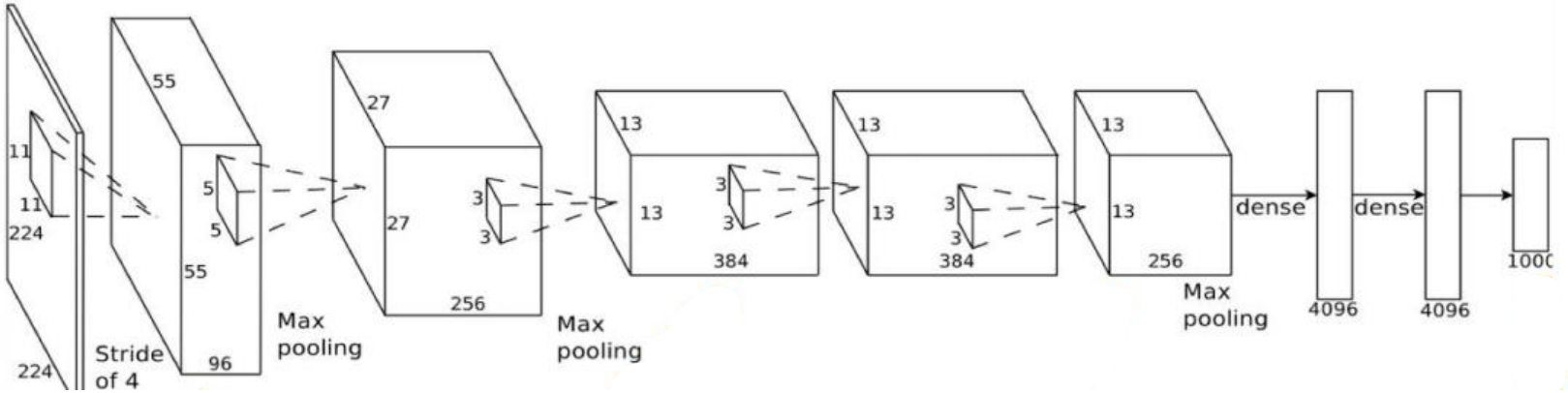


Input: 227 x 227 x 3 image
After CONV1: 55 x 55 x 96

Second layer (POOL1): 3x3 filters applied with stride = 2

Q: What is the total number of parameters in the second layer?
A: Zero!

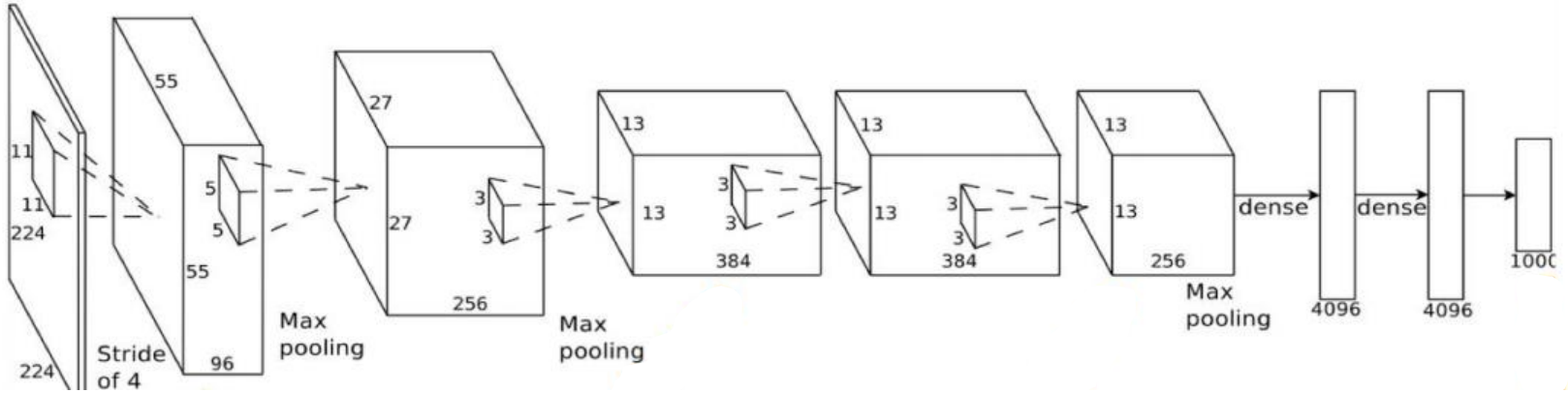
AlexNet architecture



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

Input: 227 x 227 x 3 image
After CONV1: 55 x 55 x 96
After POOL1: 27 x 27 x 96
…

Full architecture

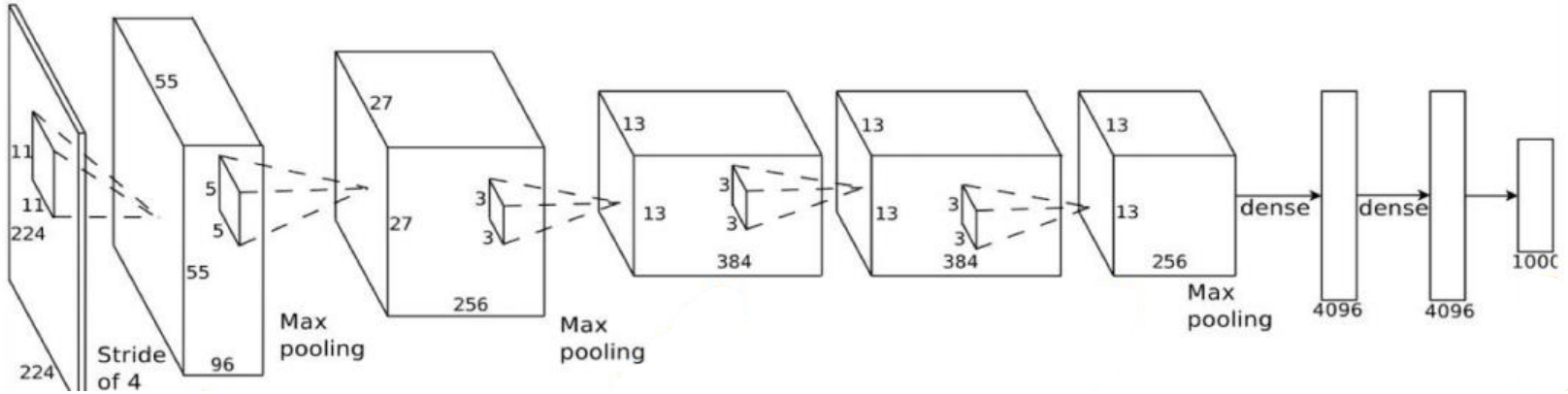


[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] 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)

CONV2:
pad = 2 on each side: 27 + 2*2 = 31
Output size: (31-5)/1+1 = 27

POOL2:
Output size: (27-3)/2+1 = 13

Full architecture

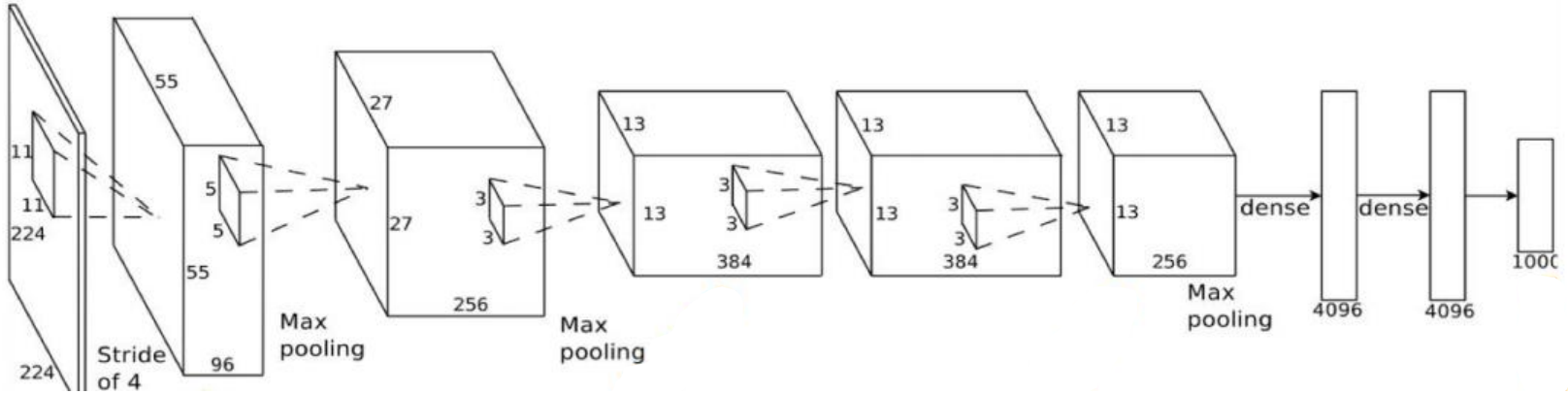


[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] 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)

CONV3/4/6:
pad = 1 on each side: 13 + 2 = 15
Output size: (15-3)/1+1 = 13

POOL3:
Output size: (13-3)/2+1 = 6

Full 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)

Reshape before FC6:

- Recall input to fully connected (FC) layer must be a vector
- Output of POOL3 is 6 x 6 x 256 = 9,216
- Reshaped into 1 x 9,216 vector

Number of parameters

Layer Name	Tensor Size	Weights	Biases	Parameters
Input Image	227x227x3	0	0	0
Conv-1	55x55x96	34,848	96	34,944
MaxPool-1	27x27x96	0	0	0
Conv-2	27x27x256	614,400	256	614,656
MaxPool-2	13x13x256	0	0	0
Conv-3	13x13x384	884,736	384	885,120
Conv-4	13x13x384	1,327,104	384	1,327,488
Conv-5	13x13x256	884,736	256	884,992
MaxPool-3	6x6x256	0	0	0
FC-6	4096×1	37,748,736	4,096	37,752,832
FC-7	4096×1	16,777,216	4,096	16,781,312
FC-8	1000×1	4,096,000	1,000	4,097,000
Output	1000×1	0	0	0
Total				**62,378,344**

FC6:

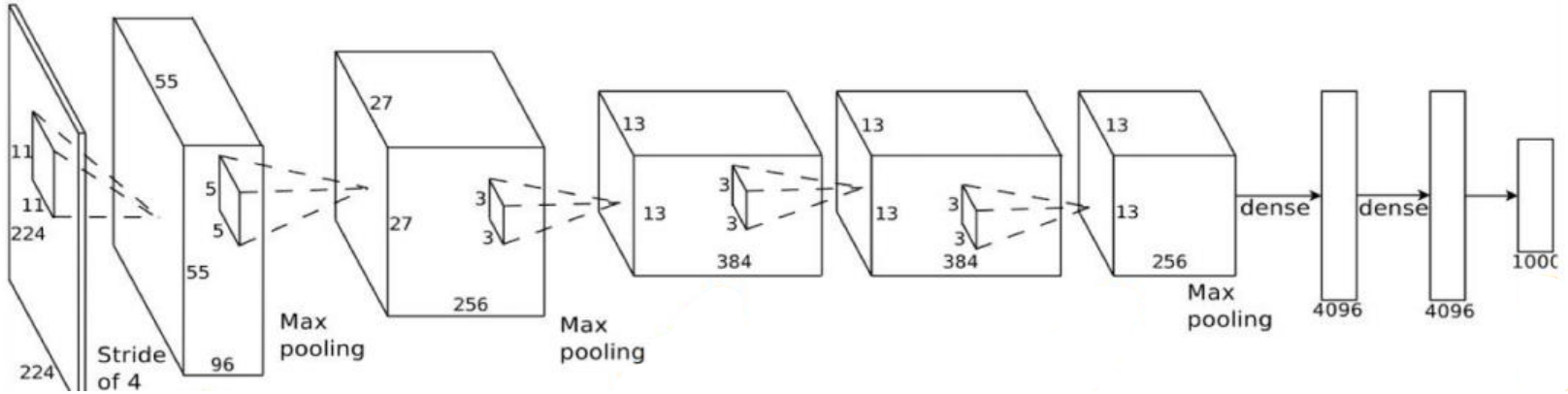
- Input dimension: 9,216
- Output dimensions: 4,096
- Total: 37,748,736 weights

Memory consumption:

- 4 bytes per parameter
- 249 MB

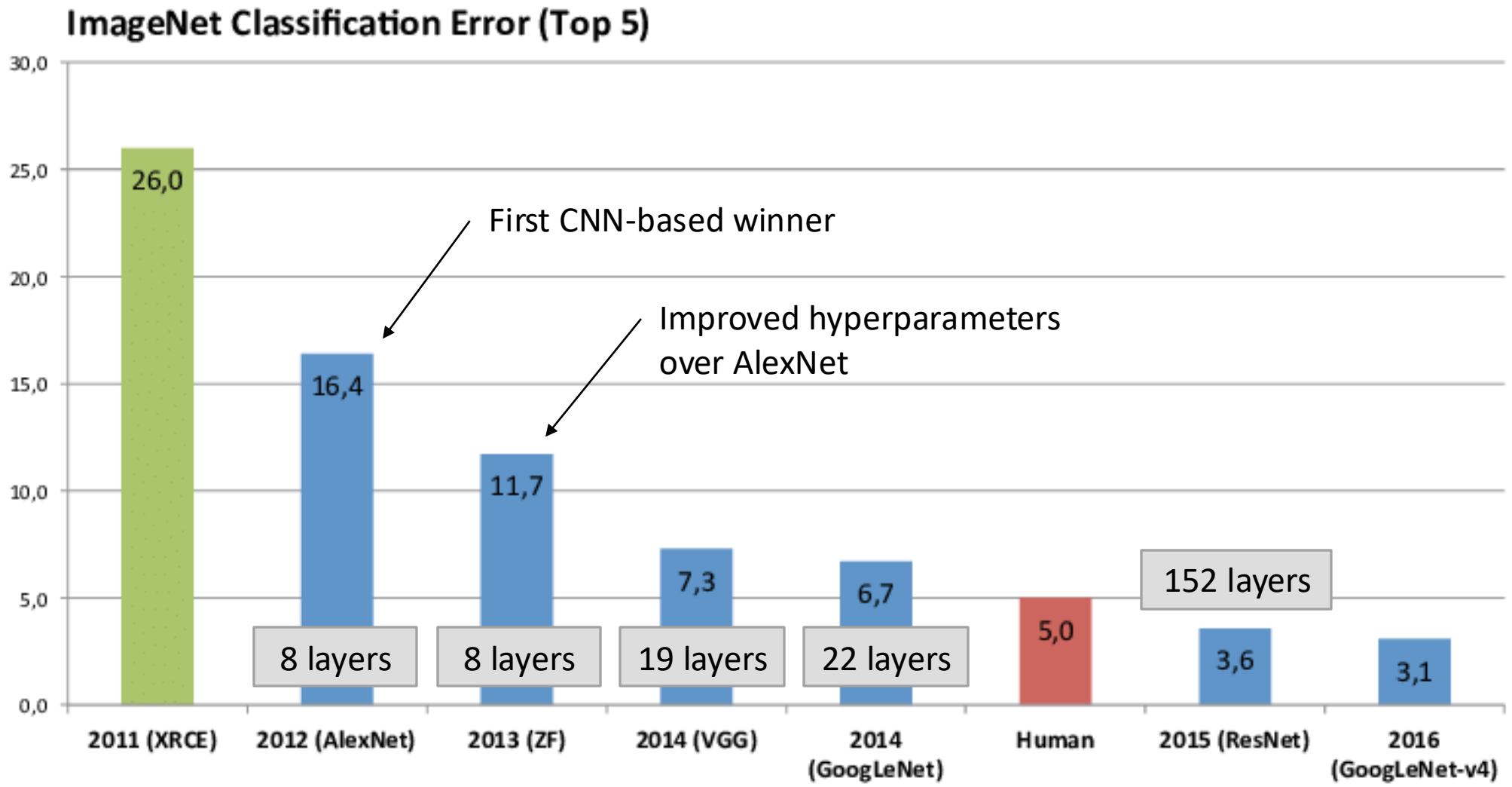
Details/retrospectives

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



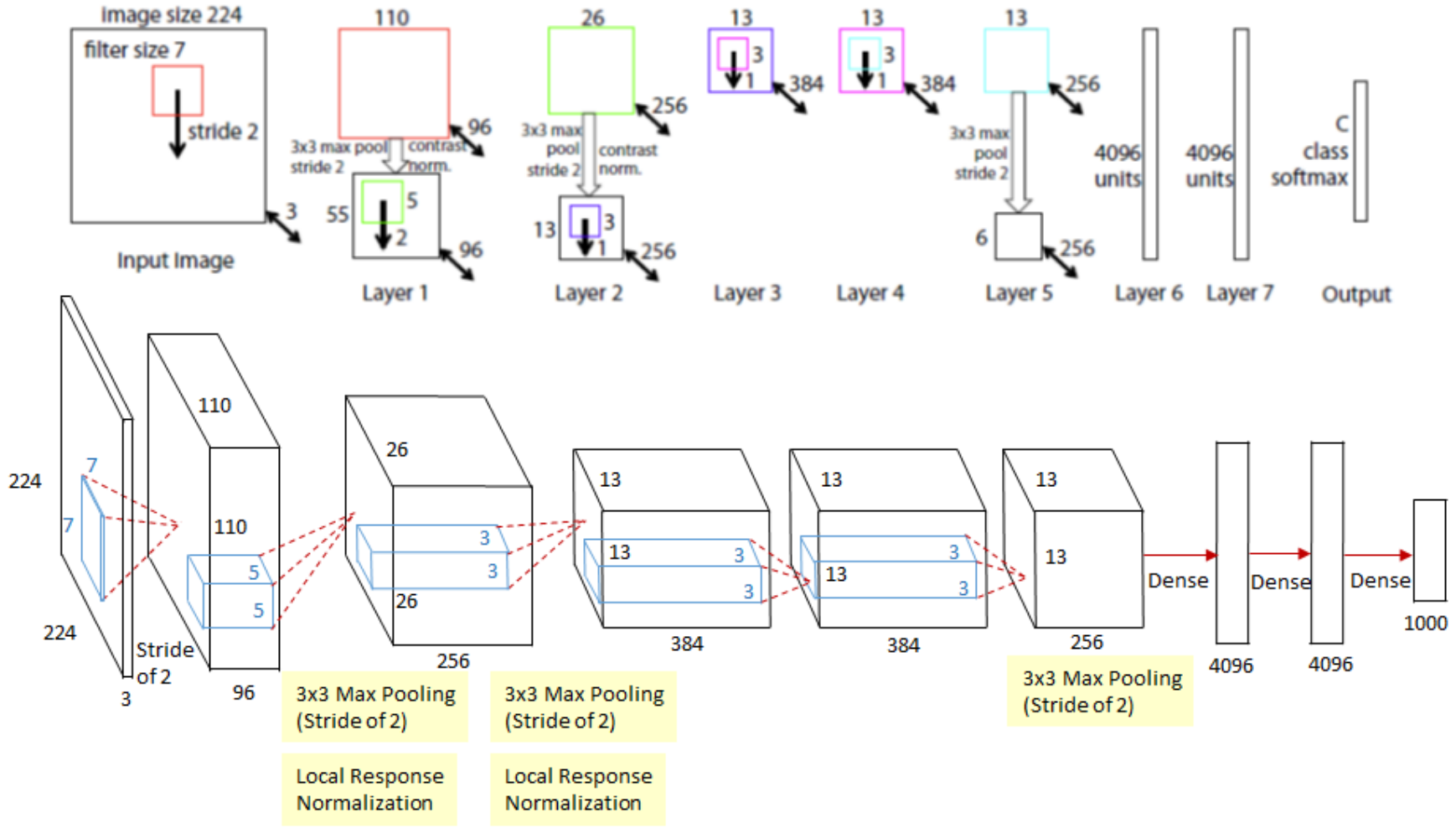
- First use of ReLU
- Used Local Response Normalization layers (not common anymore – replaced with batch normalization)
- Heavy data augmentation
- Dropout 0.5 in first two FC layers
- Batch size 128, SGD Momentum 0.9, learning rate 1e-2 (reduced by 10 manually when validation accuracy plateaus)
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



https://arxiv.org/abs/1311.2901

ZFNet architecture



AlexNet but:

CONV1: change from (11x11 stride 4) to (7x7 stride 2)

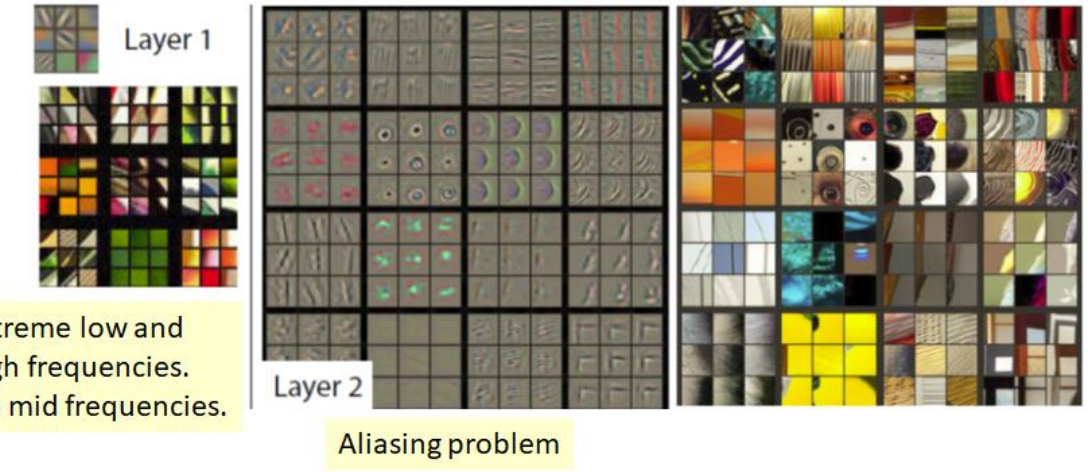
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512.

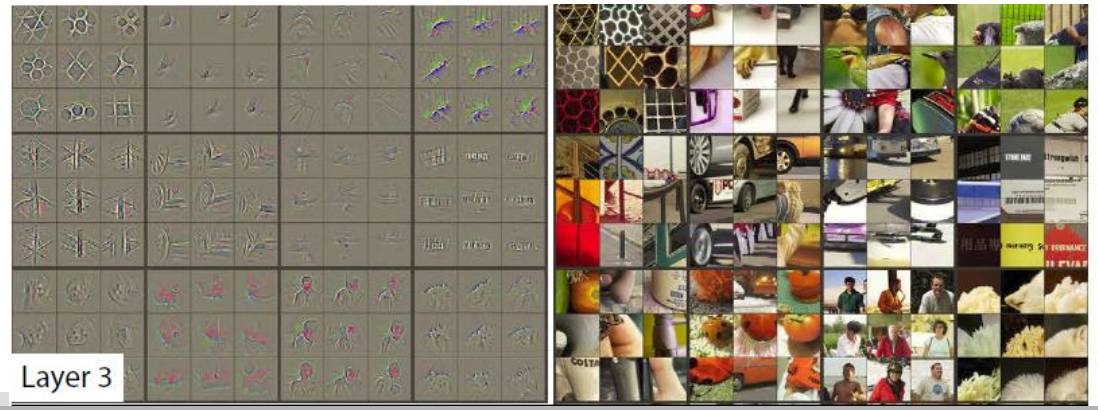
https://arxiv.org/abs/1311.2901

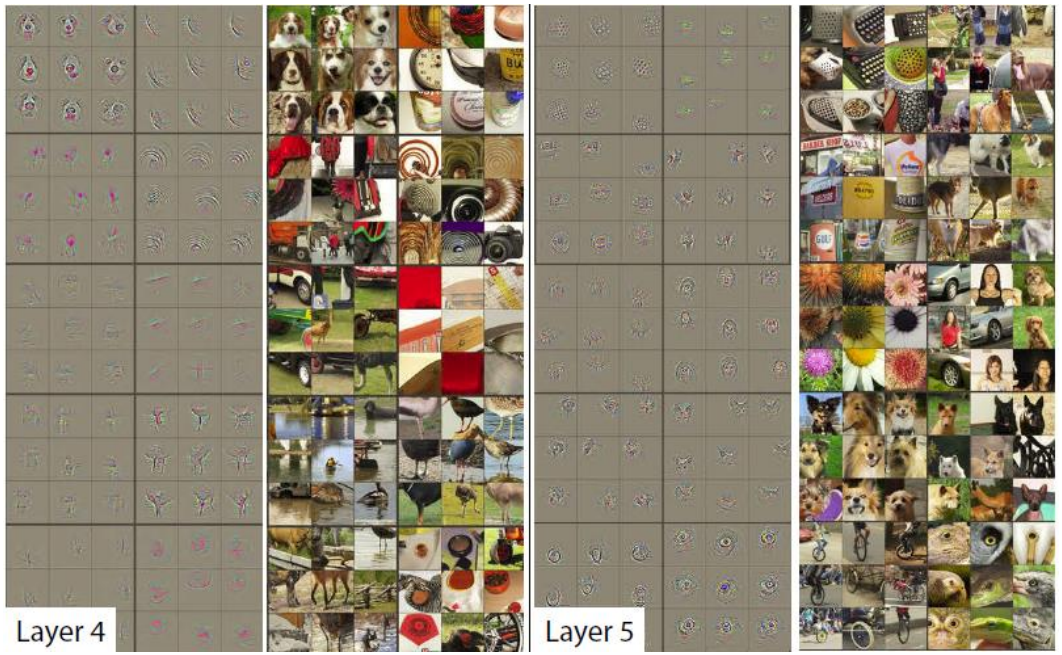
https://medium.com/coinmonks/paper-review-of-zfnet-the-winner-of-ilsvlc-2013-image-classification-d1a5a0c45103

Visualizing and Understanding Convolutional Networks

- Side-note on ZFNet (more in a later lecture): Zeiler and Fergus used different techniques to visualize network activations and improved AlexNet based on their observations.

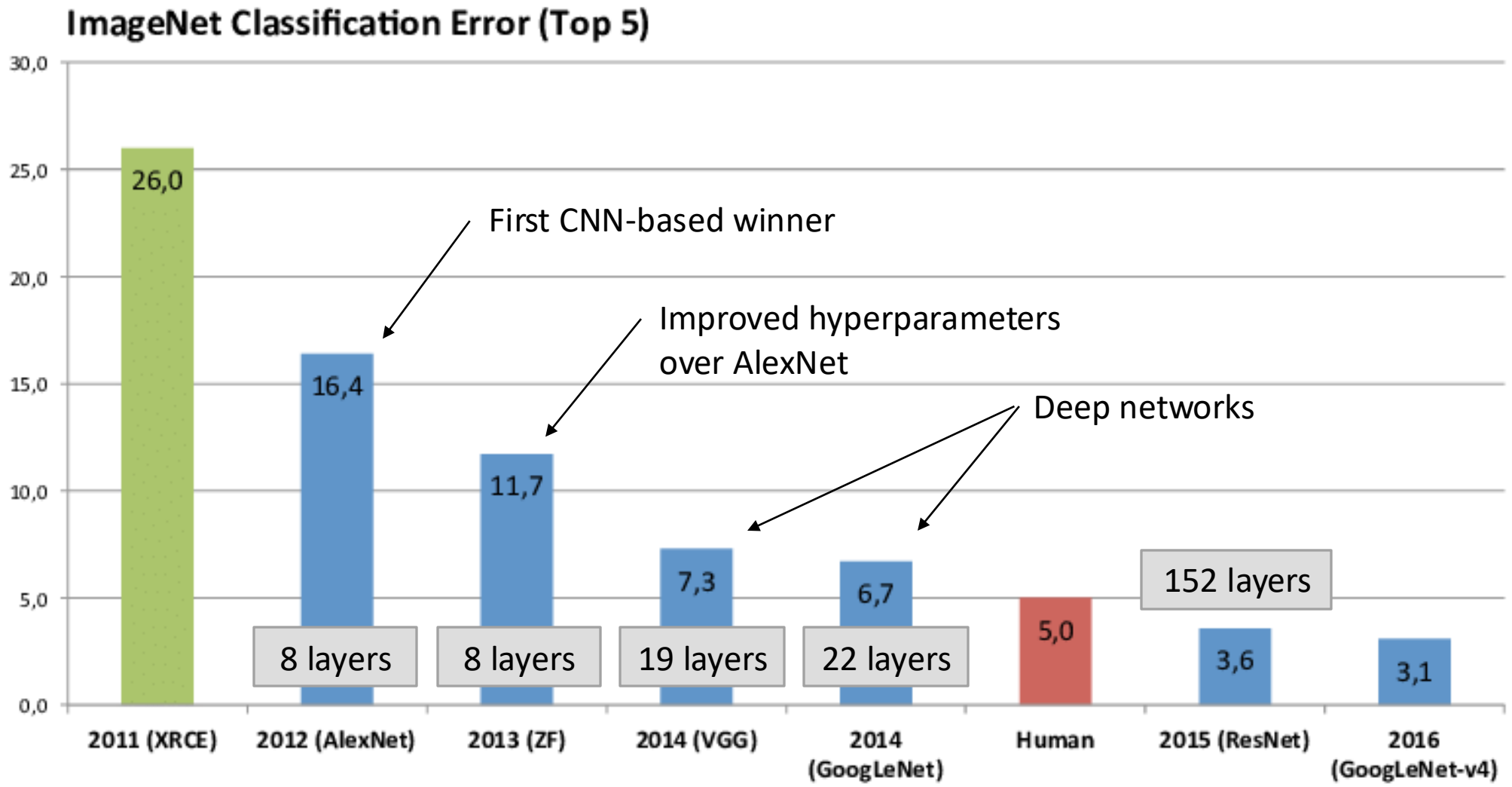






https://medium.com/coinmonks/paper-review-of-zfnet-the-winner-of-ilsvlc-2013-image-classification-d1a5a0c45103

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

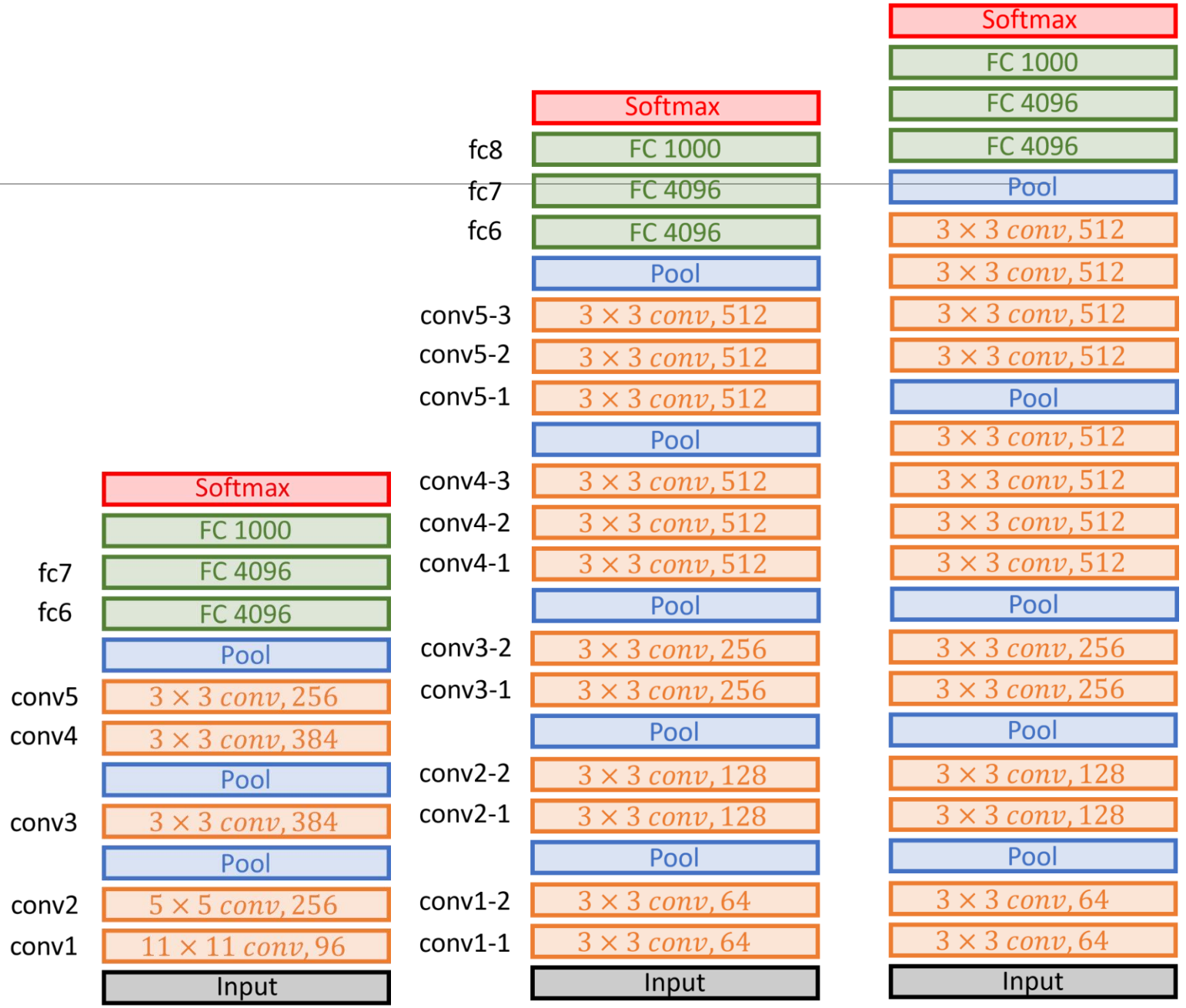


https://arxiv.org/abs/1311.2901

VGGNet

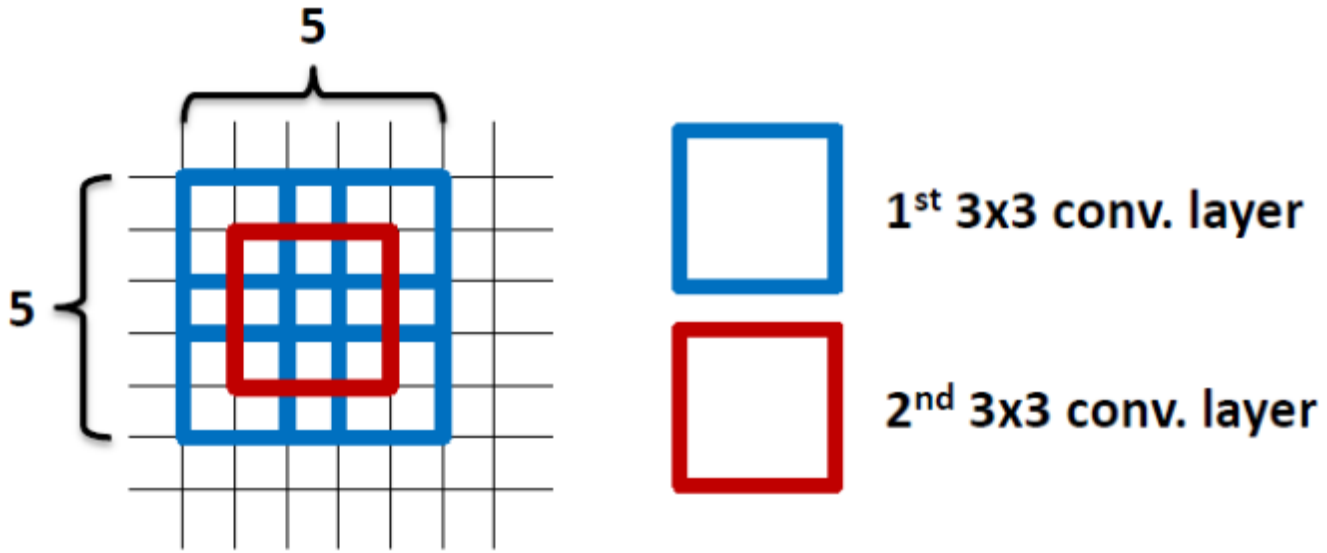
VGGNet

- Small filters, deeper networks
- From 8 layers (AlexNet) to 16-19 layers (VGGNet).
- Only 3x3 CONV stride 1, pad 1 and 2x2 MAX POOL stride 2.
- https://arxiv.org/abs/1409.1556
- Figures
  - http://datahacker.rs/deep-learning-alexnet-architecture/
  - http://datahacker.rs/deep-learning-vgg-16-vs-vgg-19/



VGGNet

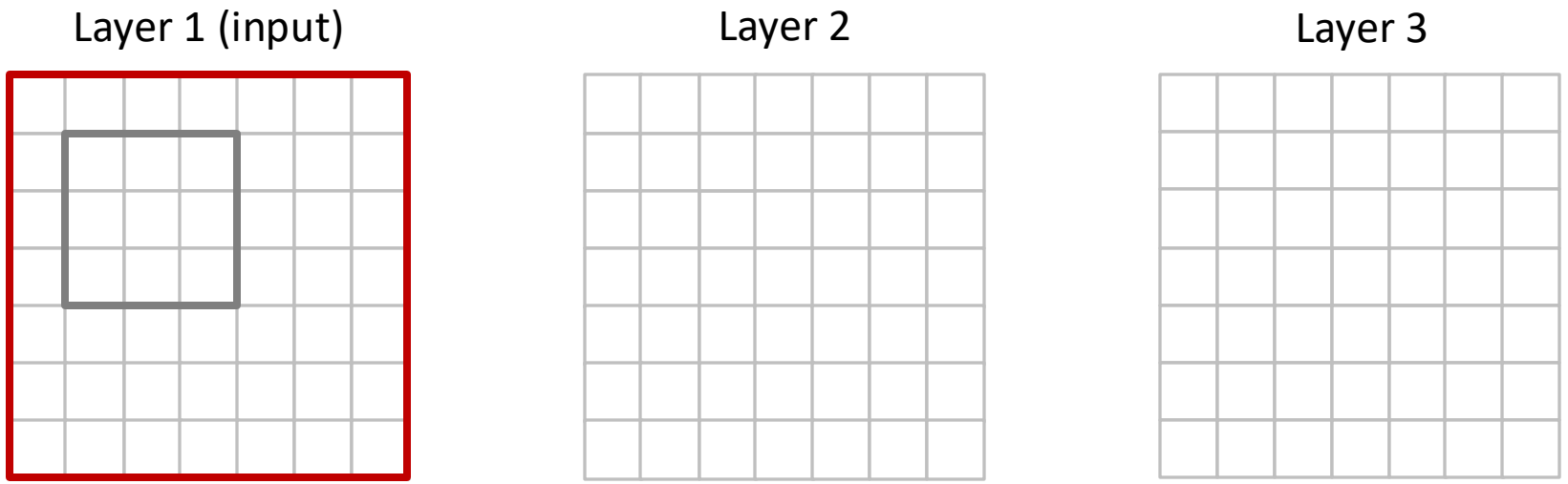
- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.



https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

VGGNet

- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.

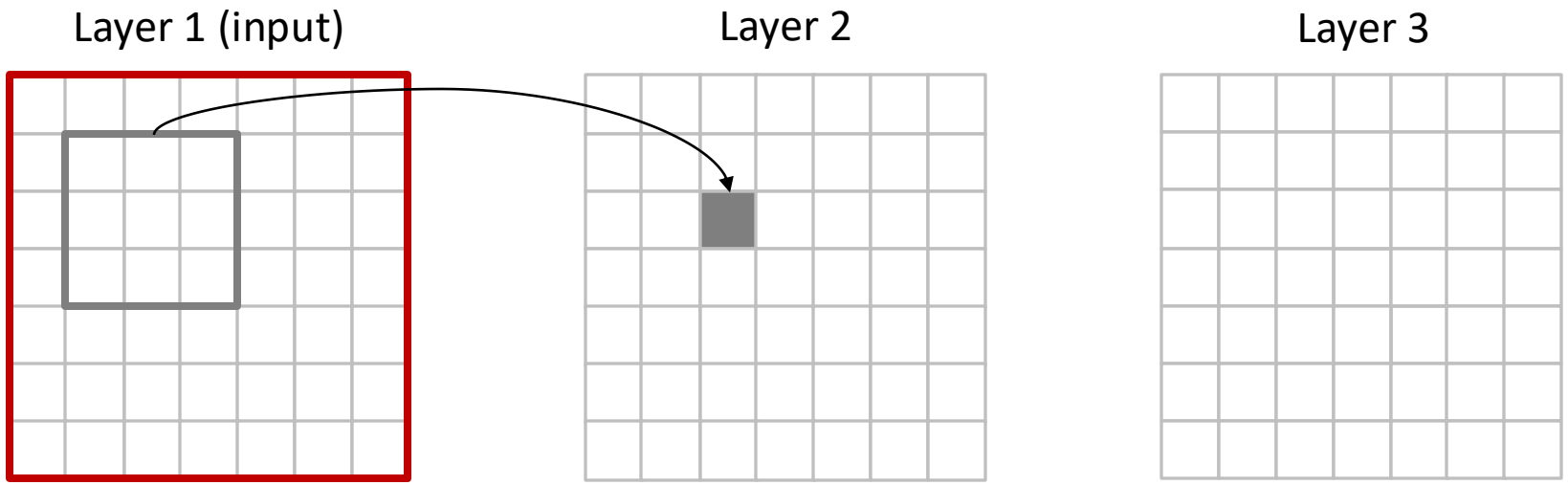


Each 3x3 filter covers a 3x3 area of the 7x7 input image

https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

VGGNet

- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.

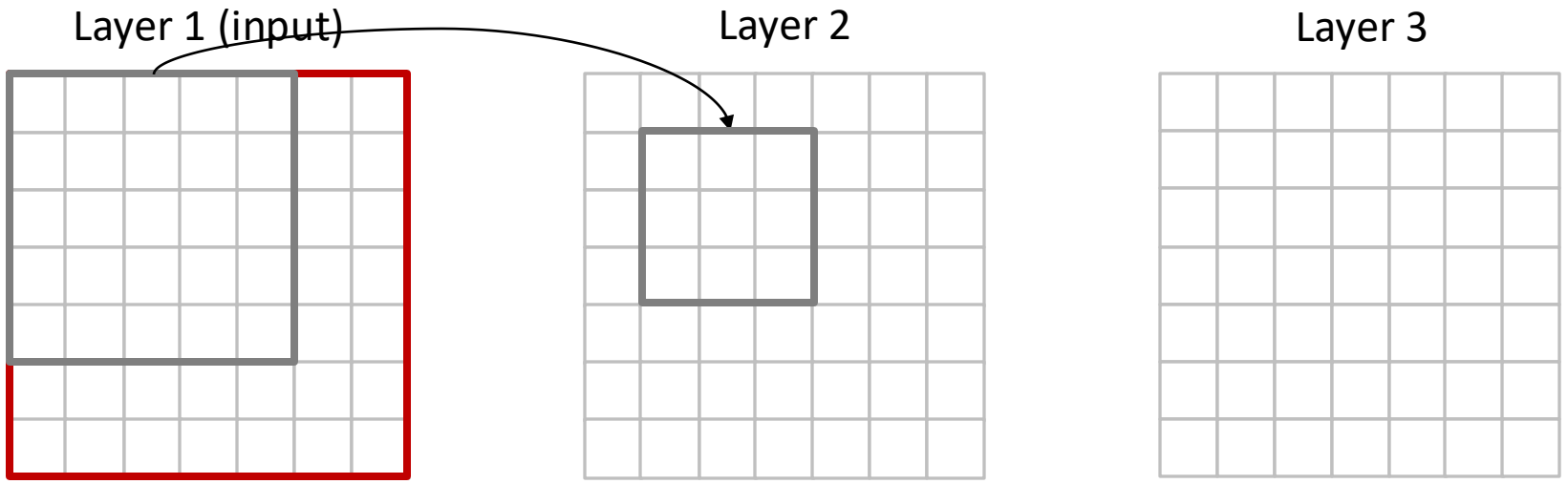


Each 3x3 filter covers a 3x3 area of the 7x7 input image

https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

VGGNet

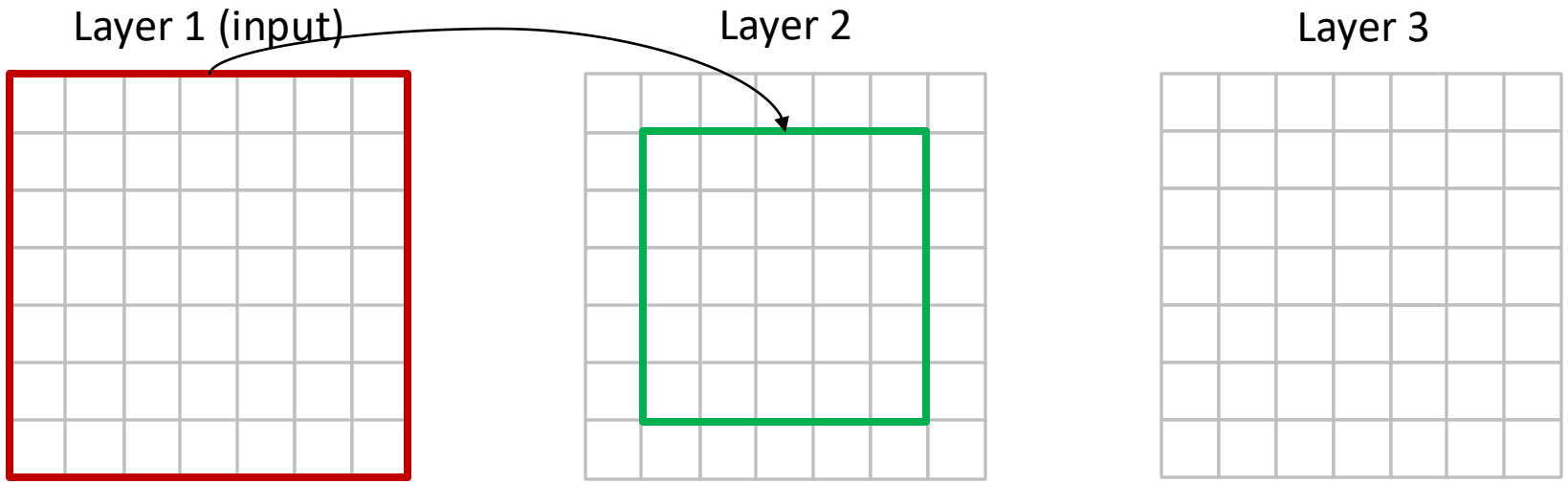
- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.



https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

VGGNet

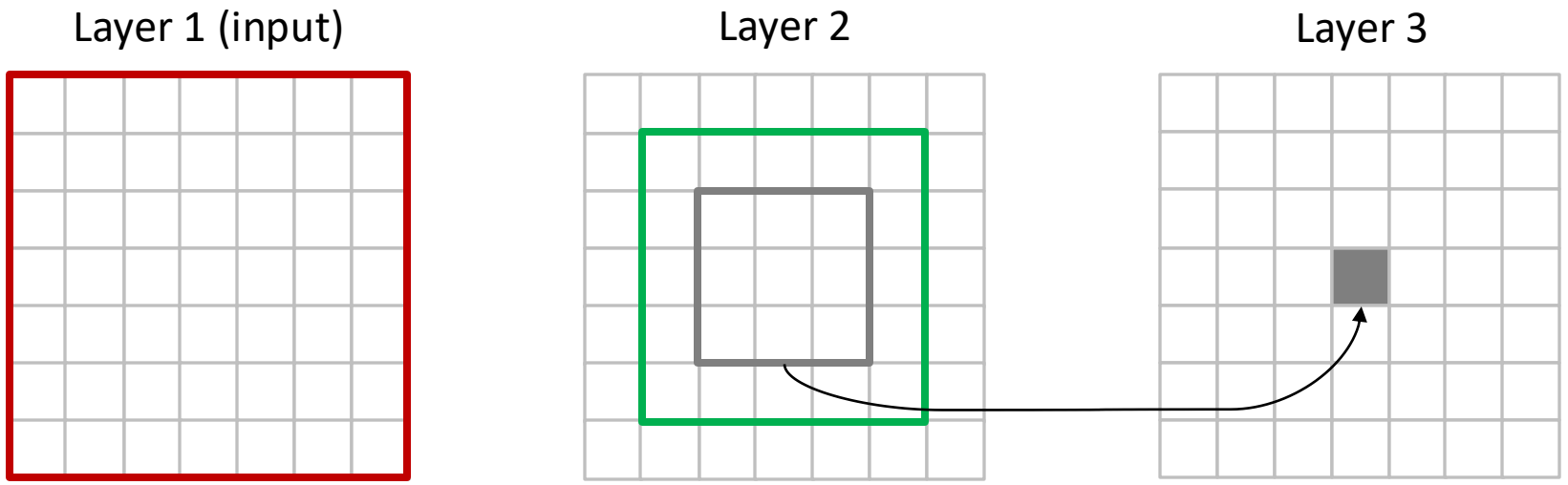
- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.



https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

VGGNet

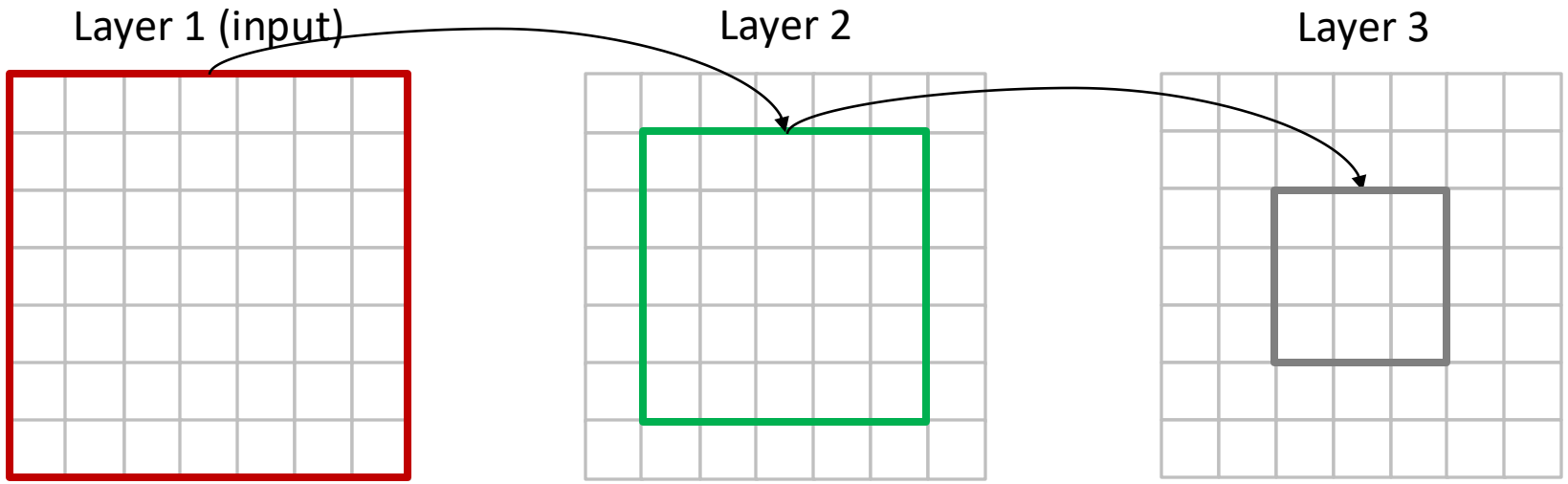
- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.



https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

VGGNet

- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).
  - By using 2 layers of 3×3 filters, network has already covered 5×5 area.
  - By using 3 layers of 3×3 filters, network has already covered 7×7 effective area.
  - Thus, large-size filters such as 11×11 in AlexNet and 7×7 in ZFNet indeed are not needed.



https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11

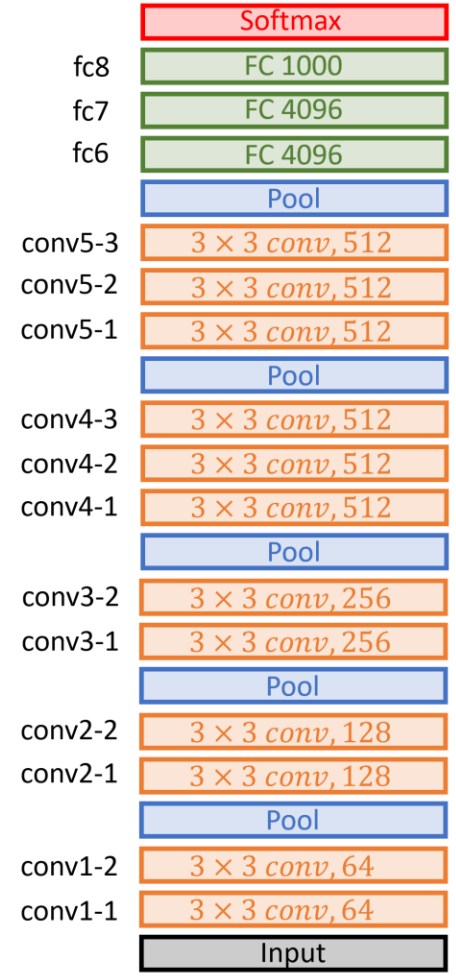
VGGNet

- **Q:** Why use smaller filters?
- **A:** Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer (like ZFNet).

- **Q:** What is the number of parameters of one 7x7 filter vs. three 3x3 filters?
- **A:** Assuming 3 channels
  - 1x7x7x3 = 147
  - 3x3x3x3 = 81
  - Formula: *num_layers x kernel_size x kernel_size x channels*
- **So fewer parameters**
- **But deeper, so more non-linearities**

Memory and number of parameters

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 = 16,777,21
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000



VGG16

Memory and number of parameters

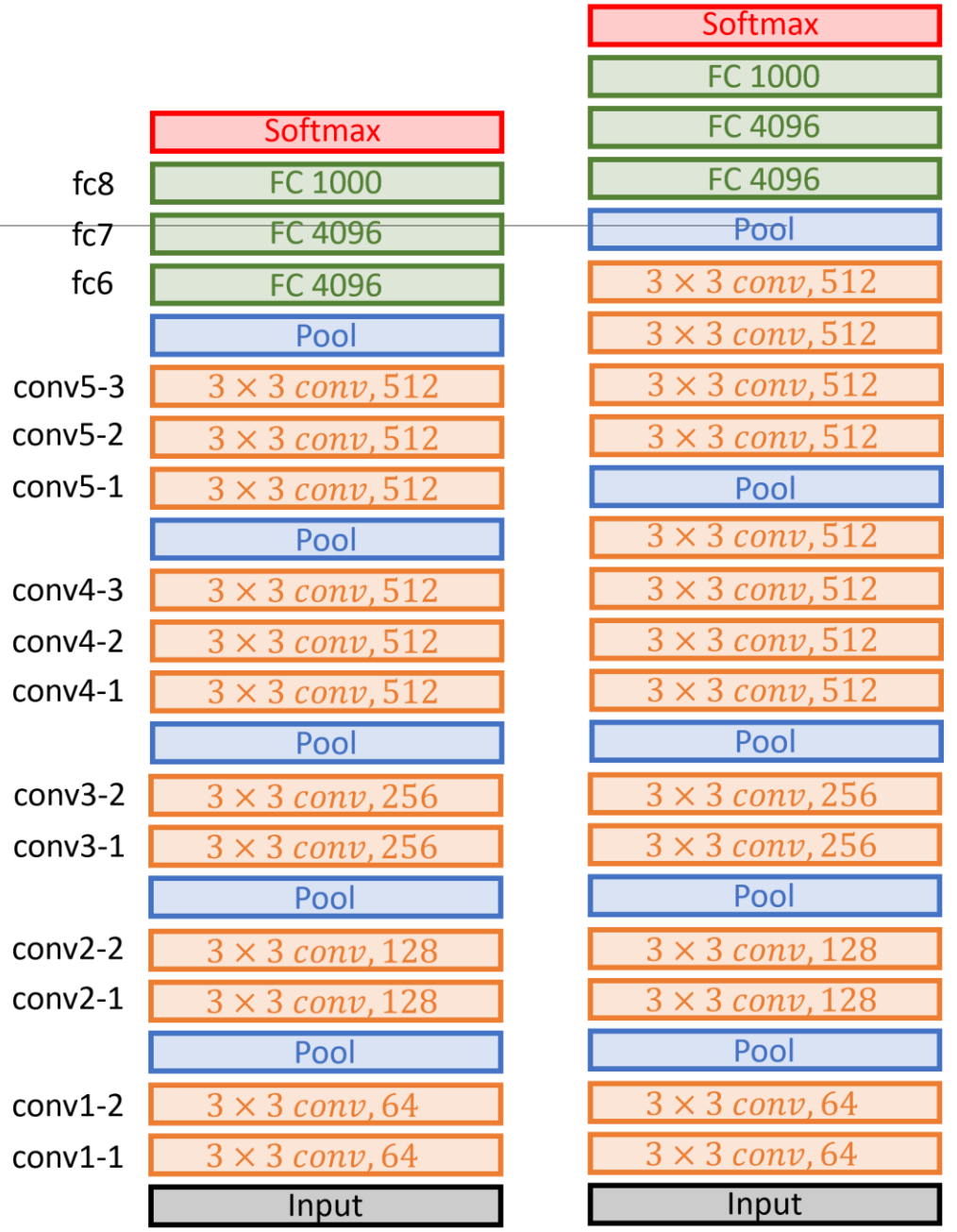
INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M 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 = 16,777,21
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory:
24M * 4 bytes = 96MB/image
(for a forward pass)
(~*2 for backward pass)

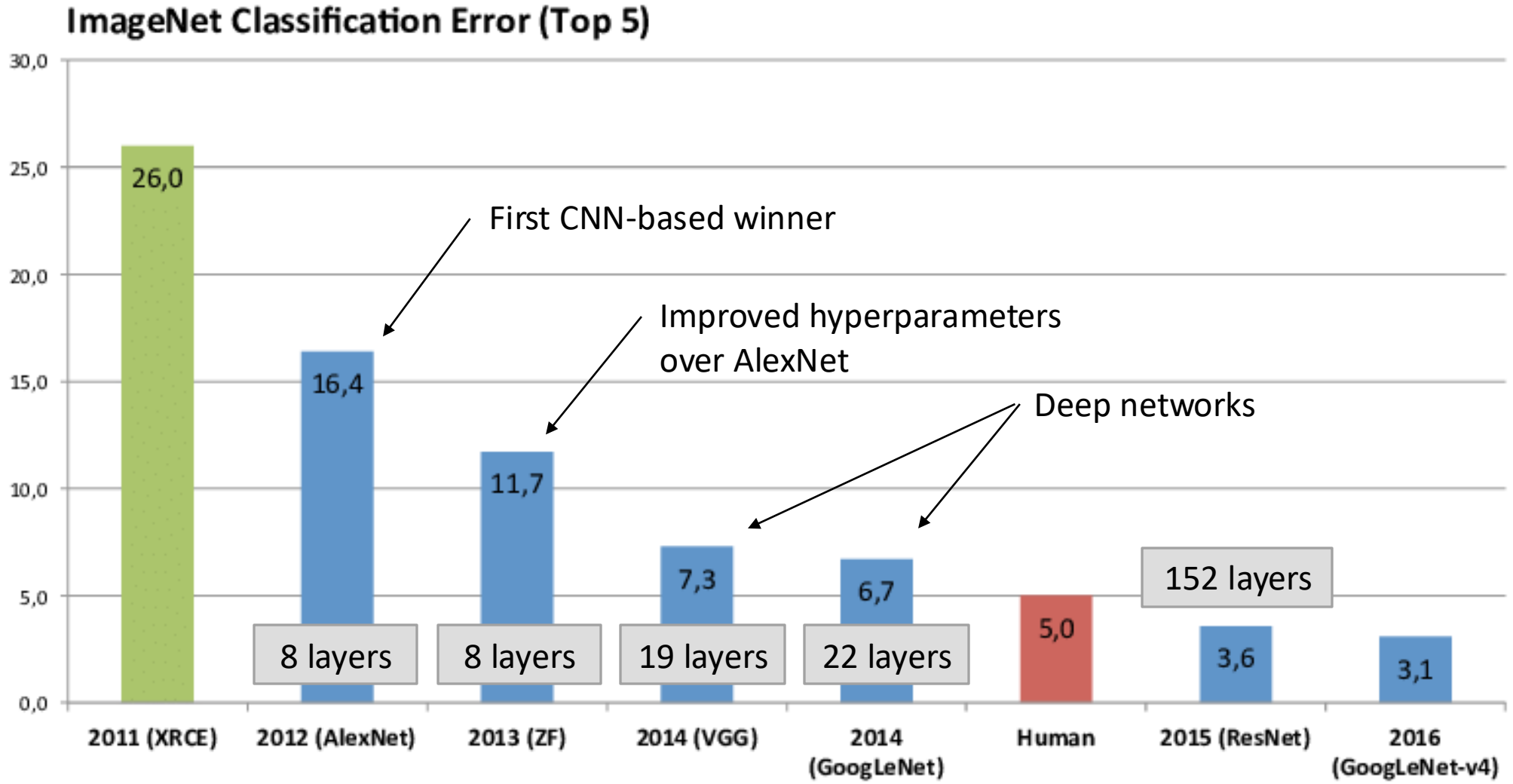
TOTAL params:
138M parameters
552 MB

VGGNet details

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as AlexNet paper
- No Local Response Normalization (paper was written before batch norm had been published)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks
- Source: https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



https://arxiv.org/abs/1311.2901

GoogleLeNet

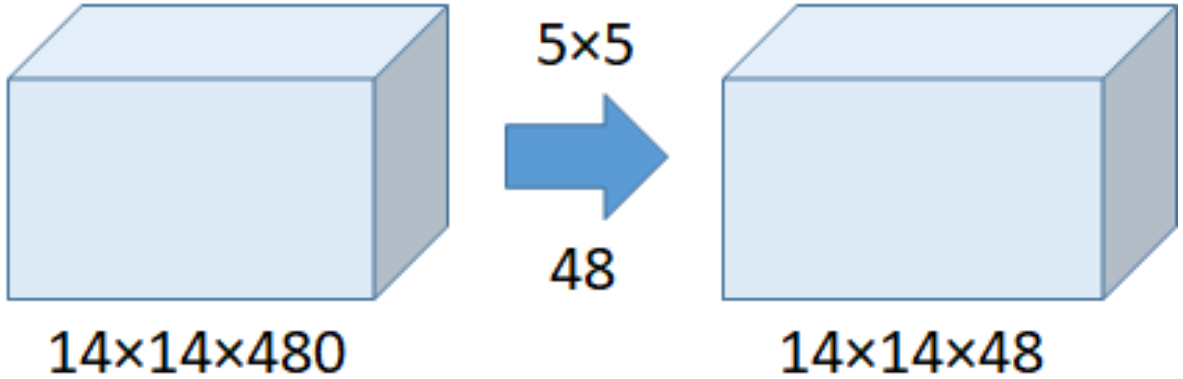
GoogLeNet (or Inception V1)

- Architecture quite different from VGGNet, ZFNet, and AlexNet.
- Contains **1×1 convolutions** at the middle of the network.
- And **global average pooling** is at the end of the network **instead of fully connected (FC) layers**.
- These two techniques are from another paper "Network In Network" (NIN).
- Another technique, called **inception module**, is to have different sizes/types of convolutions for the same input and stacking all the outputs.
- Paper: https://arxiv.org/abs/1409.4842

https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7

The 1×1 Convolution

- Originally, NIN uses 1x1 convolution for introducing more non-linearity to increase the representational power.
- In GoogLeNet, 1×1 convolution is **used as a dimension reduction module** to reduce the number of computations.
- By reducing the computation bottleneck, depth and width of network can be increased.
- Suppose we need to perform 5×5 convolution **without** the use of 1×1 convolution as below:

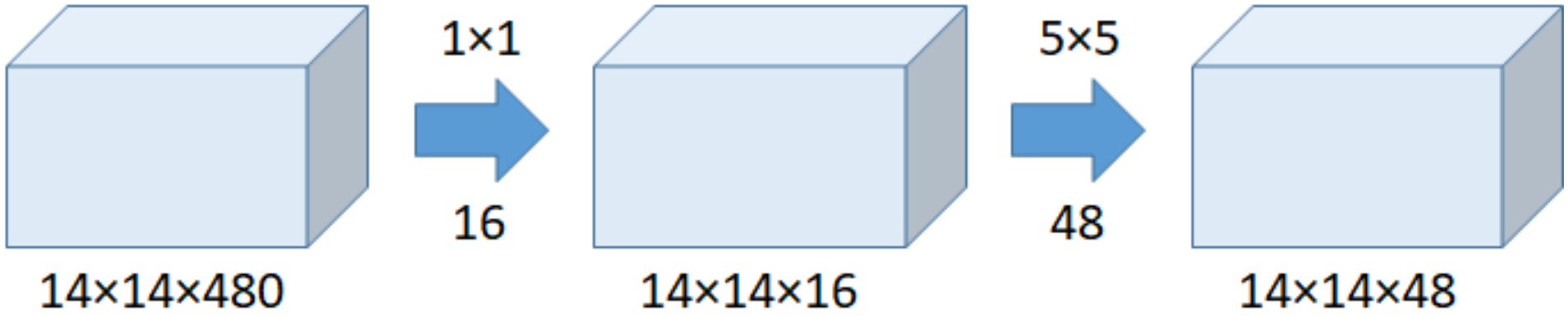


Number of operations = output_size * filter_size = (14×14×48)×(5×5×480) = 112.9M

https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7

The 1×1 Convolution

- With the use of 1×1 Convolution (example)
- Number of operations for 1×1 = (14×14×16)×(1×1×480) = 1.5M
- Number of operations for 5×5 = (14×14×48)×(5×5×16) = 3.8M
- Total number of operations = 1.5M + 3.8M = 5.3M (much smaller than 112.9M !!!)
- 1×1 convolution can help to reduce model size (and reduce the overfitting problem!!!)

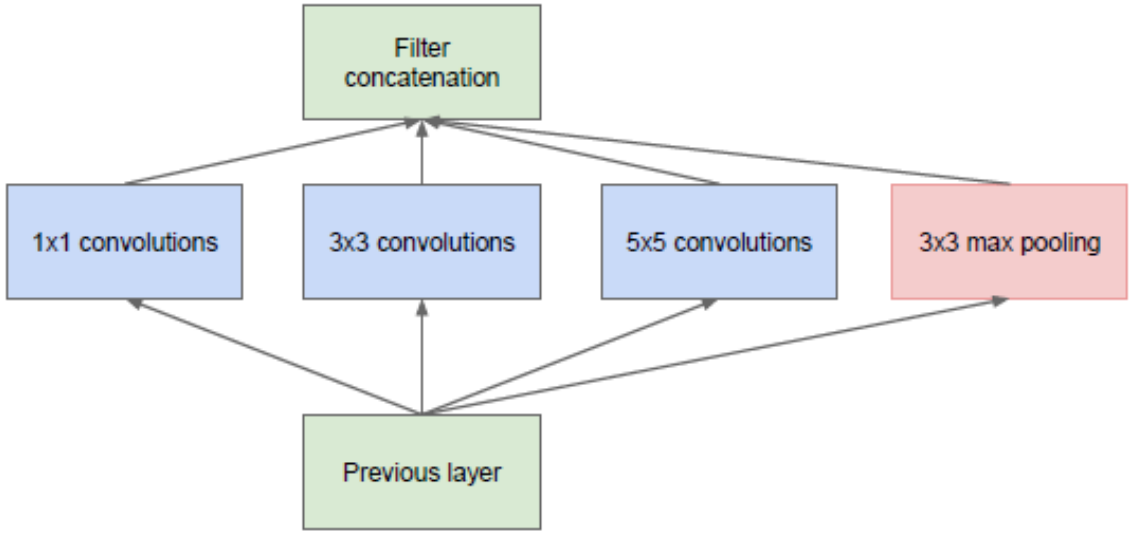


Think "dimension reduction module"

https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7

Inception Module

- Design a good local network topology (**network within a network**) and then **stack these modules** on top of each other.
- The inception module (naive version, without 1×1 convolution) is as on the right.
- Conv size is no longer fixed for each layer: Now, **1×1 conv**, **3×3 conv**, **5×5 conv**, and **3×3 max pooling** are done altogether for the previous input and stacked together again at output.
- Different kinds of features are extracted.

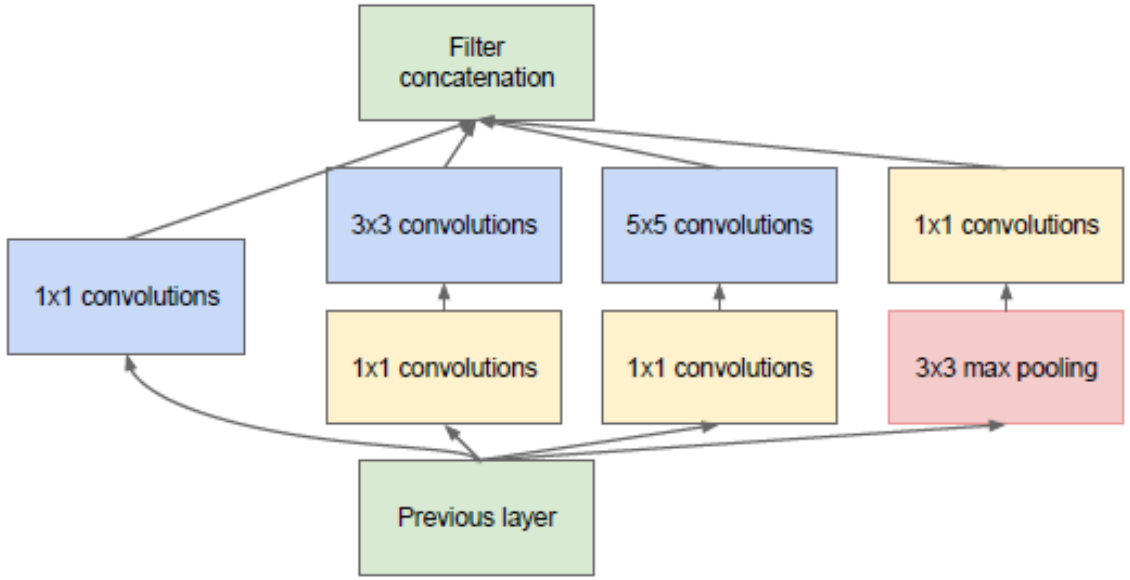


(a) Inception module, naïve version

https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7

Inception Module

- For each module, all feature maps are stacked (concatenated) together as the input to the next module.
- **High computational complexity:** Naively concatenating feature maps, we can imagine how large the number of operations becomes!
- 1×1 convolution is inserted into the inception module for **dimension reduction**.



(b) Inception module with dimensionality reduction

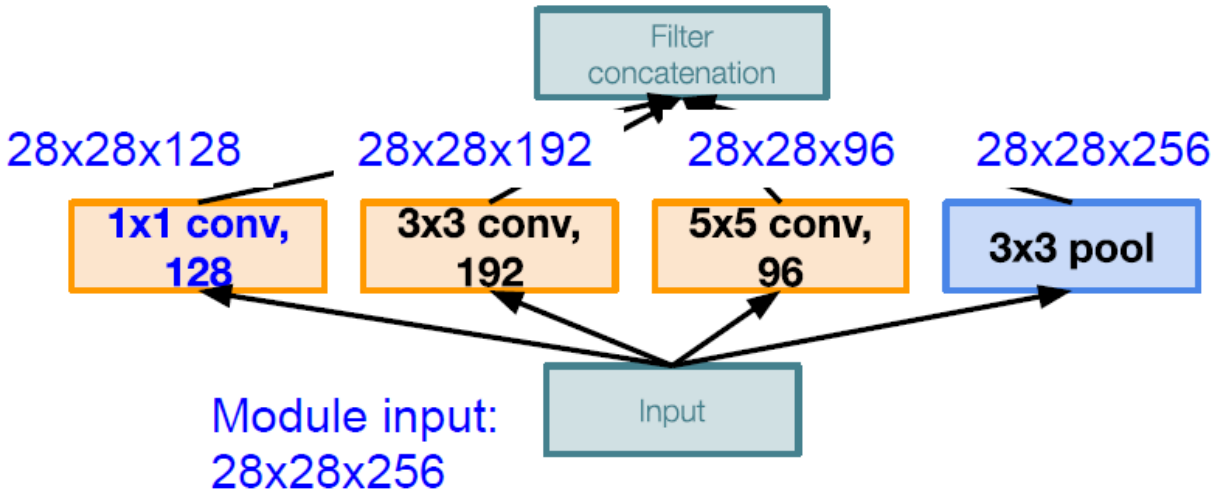
https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7

Inception Module

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672



Naive Inception module

Conv Ops:

[1x1 conv, 128] 28x28x128x1x1x256

[3x3 conv, 192] 28x28x192x3x3x256

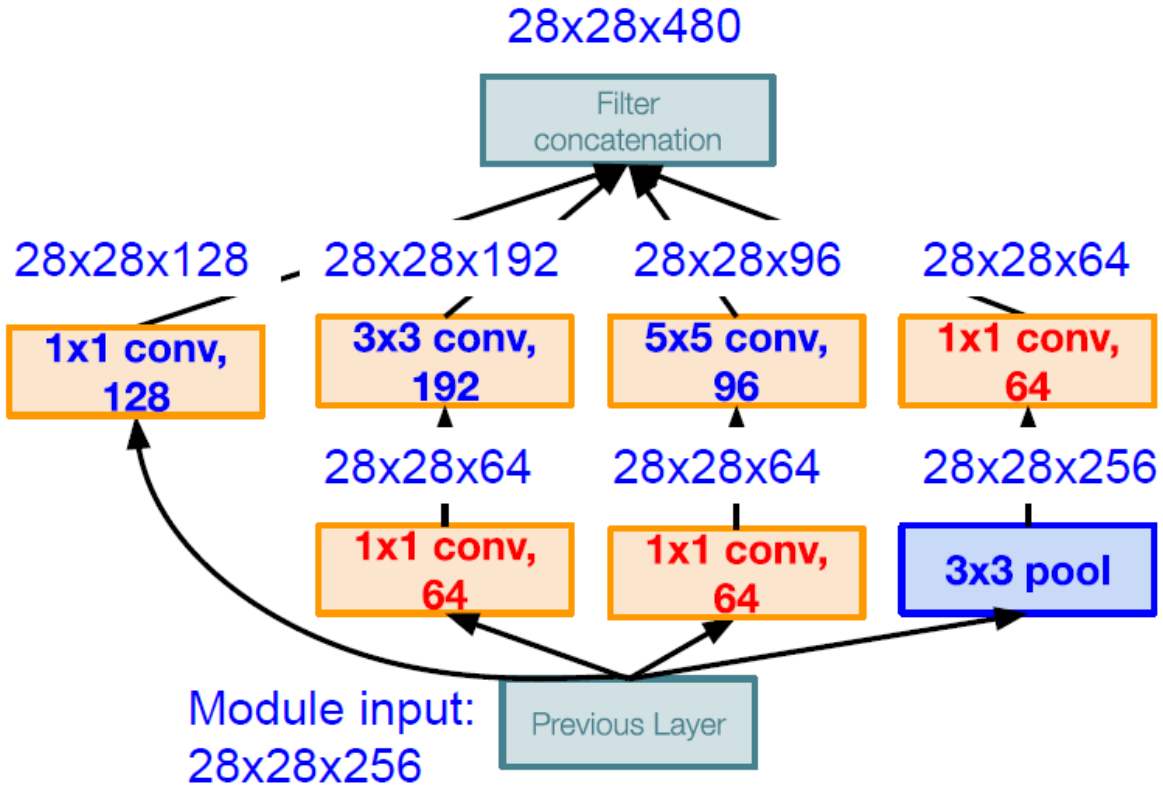
[5x5 conv, 96] 28x28x96x5x5x256

Total: 854M ops

Very expensive compute

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

Inception Module



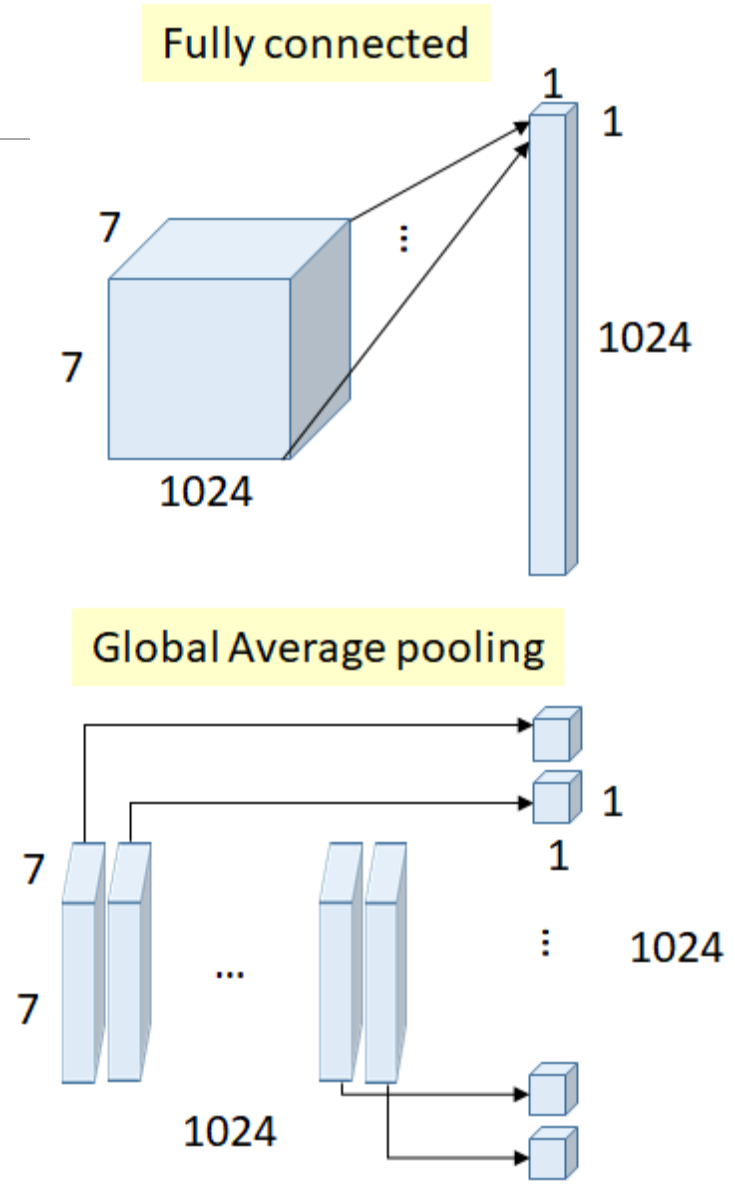
Inception module with dimension reduction

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

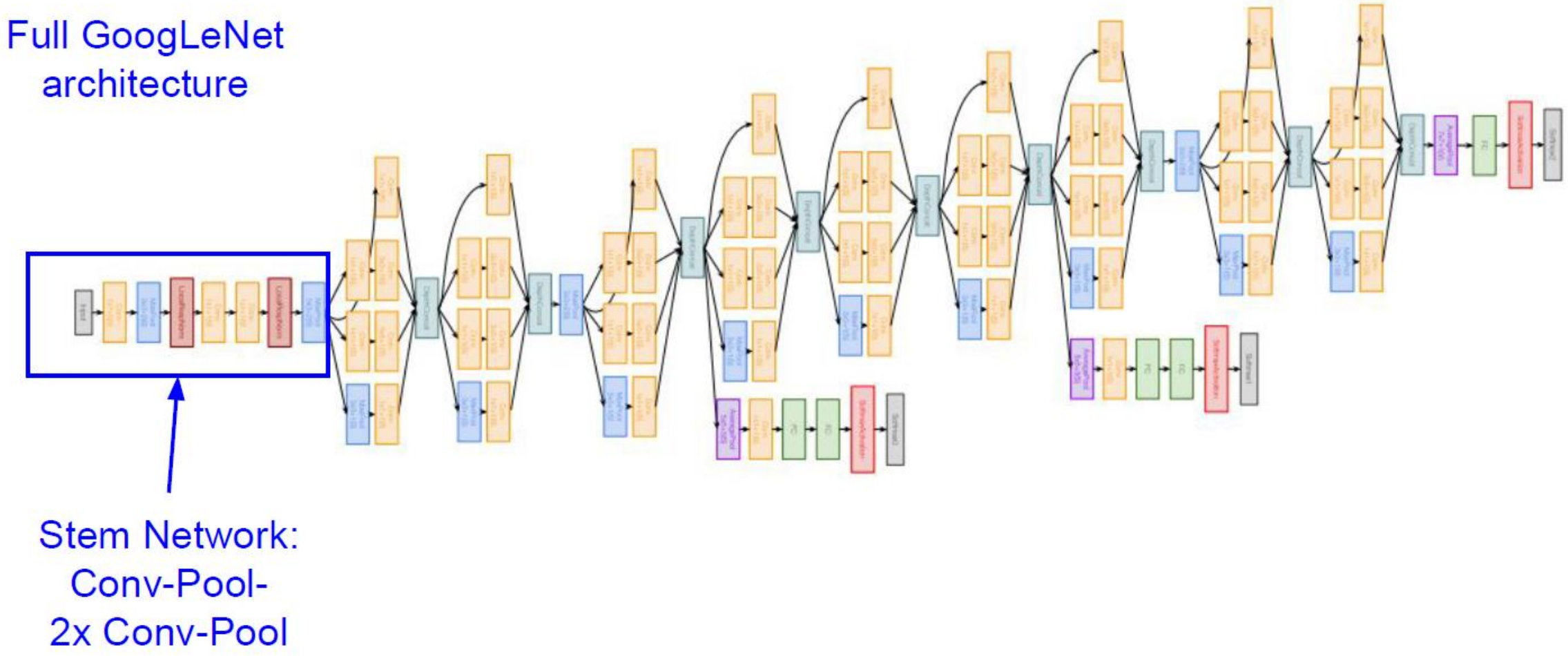
Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

Global Average Pooling

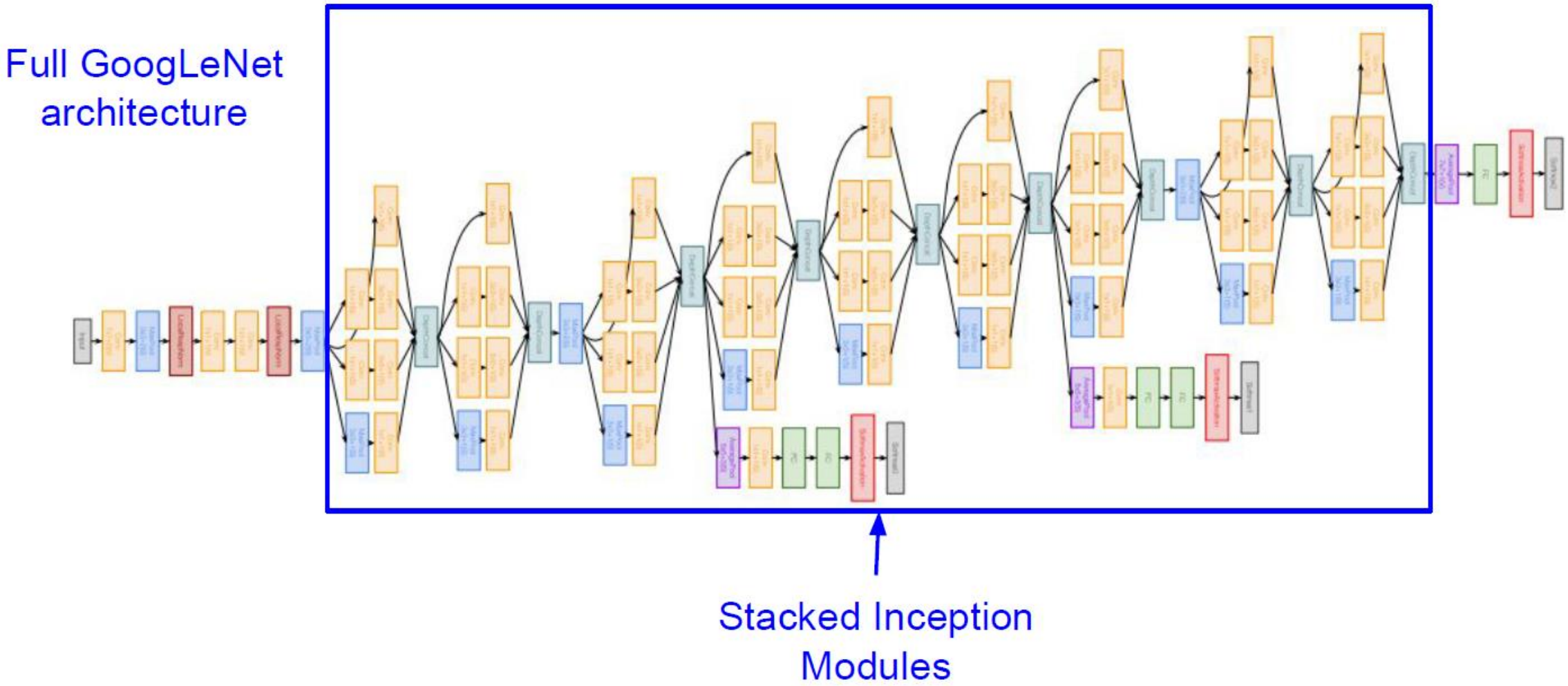
- In AlexNet, VGGNet and ZFNet, **FC layers are used at the end of network.**
- All inputs are connected to each output.
- Number of weights or connections in FC example (top figure): 7×7×1024×1024 = 51.3M
- In GoogLeNet, **global average pooling** is used nearly at the end of network by **averaging each feature map from 7×7 to 1×1**, as in the bottom figure.
- Number of weights = 0
- Authors found that a move from FC layers to **average pooling improved the top-1 accuracy by about 0.6%.** This is the idea from NIN, which can be **less prone to overfitting.**



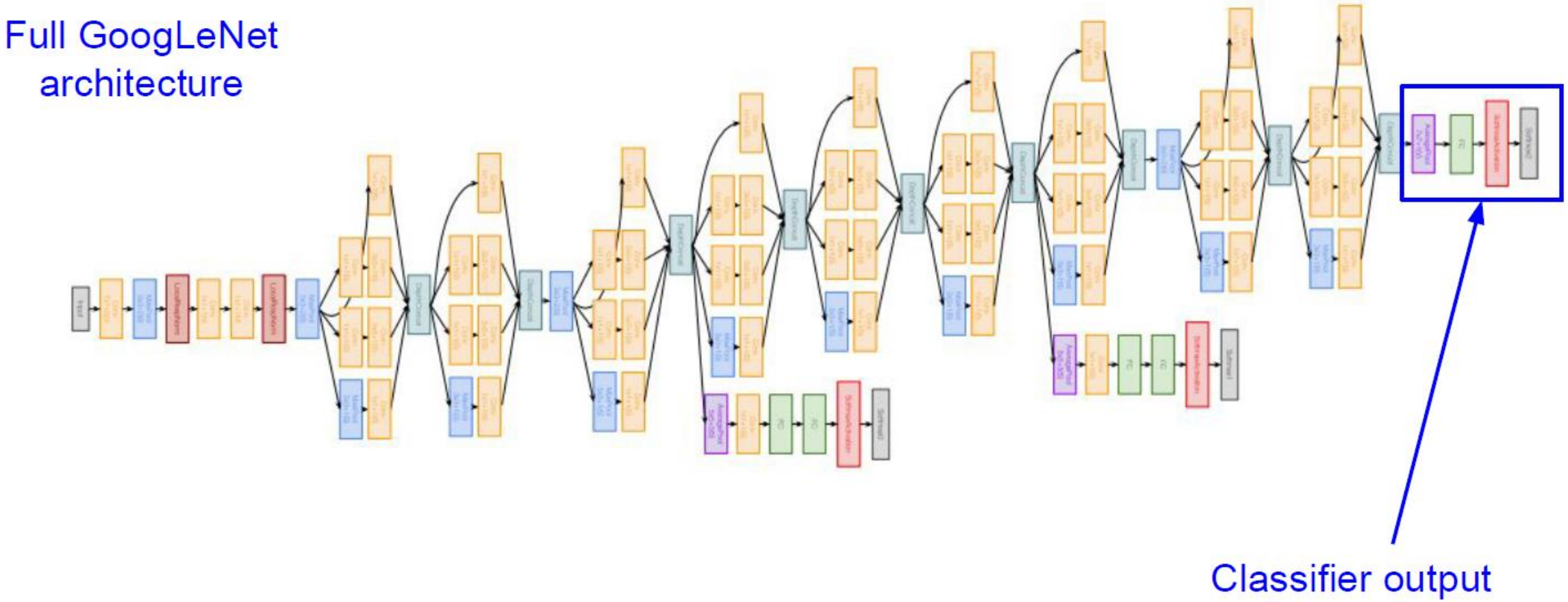
Full architecture



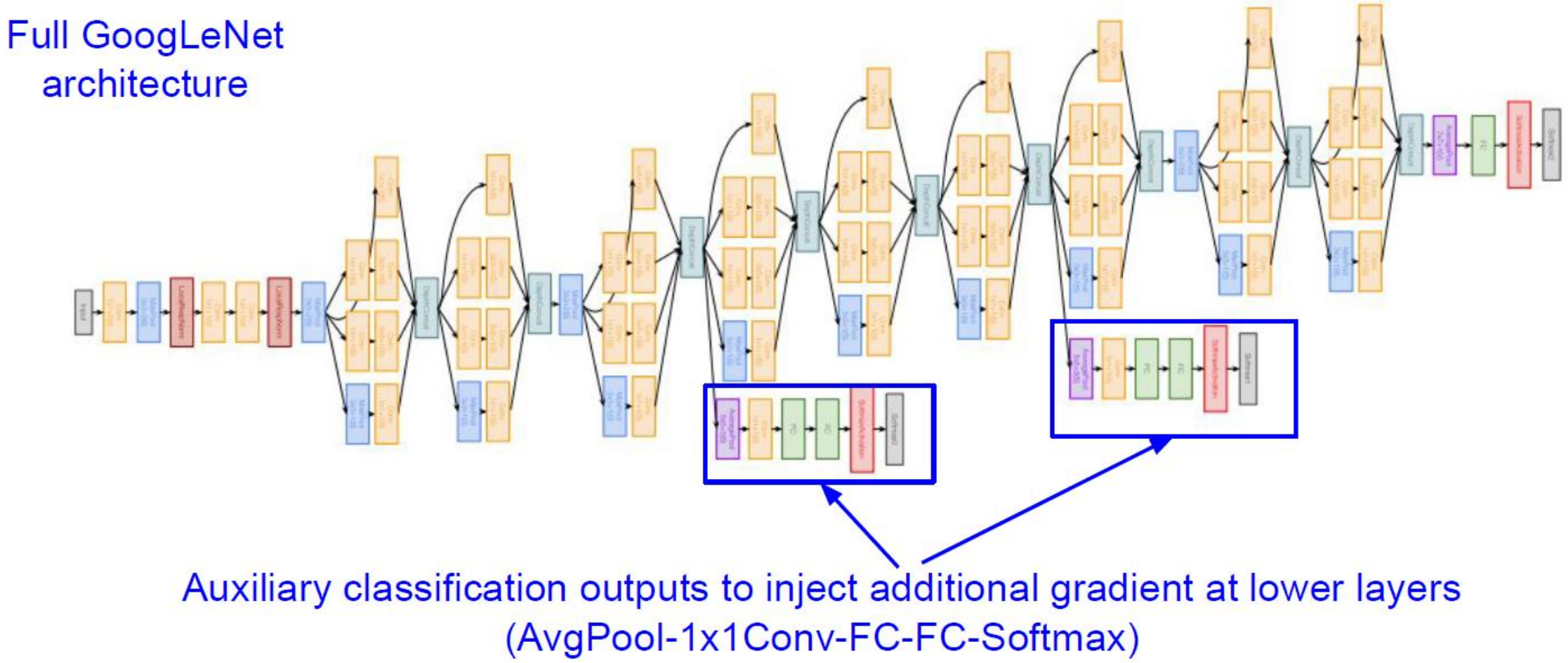
Full architecture



Full architecture



Full architecture



Auxiliary classifiers

- As we can see there are some **intermediate softmax branches** at the middle
- They are **used for training only**.
- These branches are auxiliary classifiers which consist of:
  - 5×5 Average Pooling (Stride 3)
  - 1×1 Conv (128 filters)
  - 1024 FC
  - 1000 FC
  - Softmax
- The loss is added to the total loss, with weight 0.3.
- **Authors claim it can be used for combating the vanishing gradient problem, also providing regularization.**
- Auxiliary classifiers are NOT used at test/inference time.

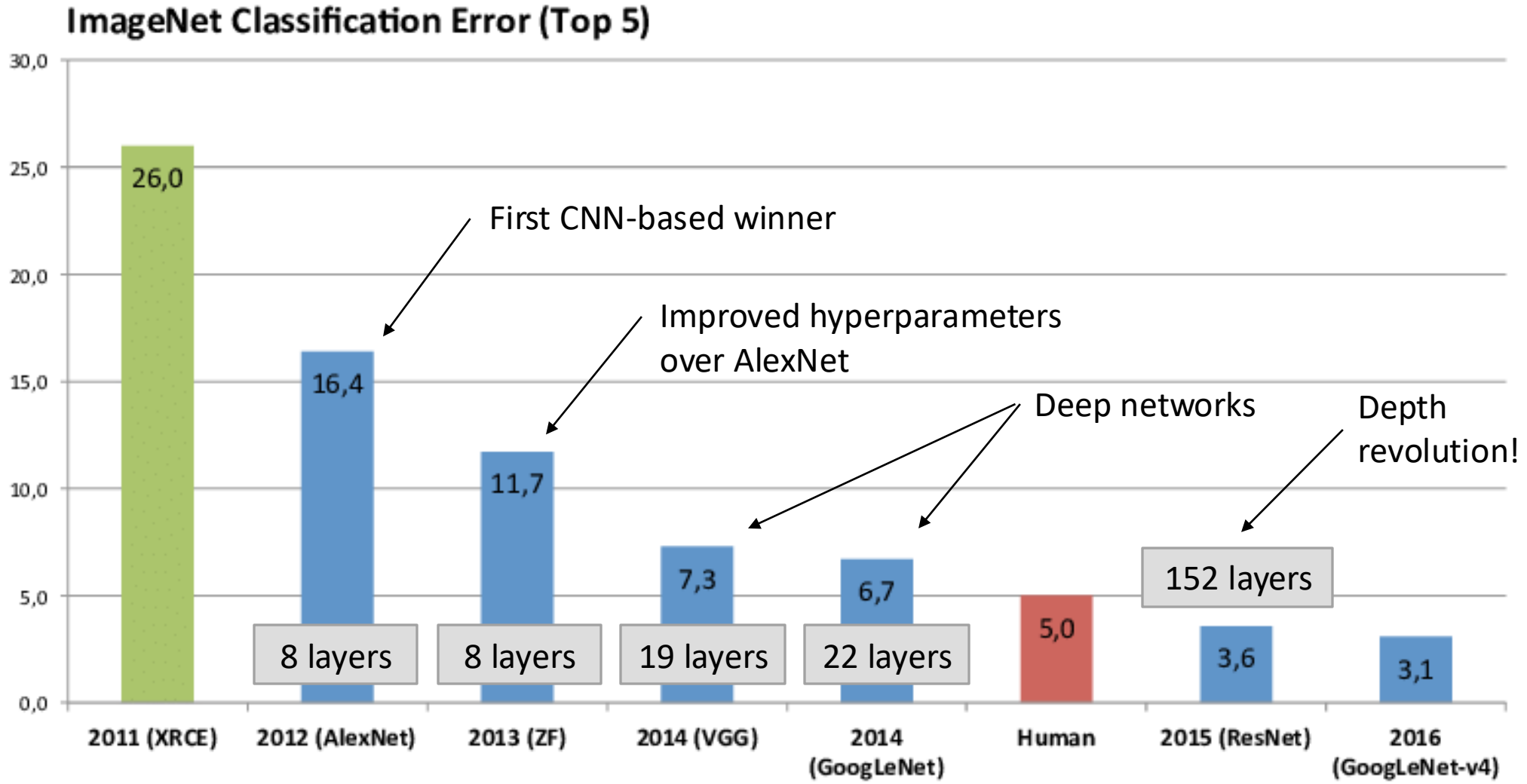
GoogLeNet details

- Deeper network, with computational efficiency
- 22 layers
- Efficient “Inception” module
- Avoids expensive FC layers
- Only 5 million parameters (**12x less than AlexNet**)
- ILSVRC’14 classification winner (6.7% top 5 error)

Inception V1 successors

- Inception V2 (early 2015)
  - Batch normalization
  - https://arxiv.org/pdf/1502.03167.pdf
- Inception V3 (late 2015)
  - Decomposed the 5×5 and 7×7 filters with multiple 3×3s.
  - Factorization of n×n convolutions by a 1×n convolution followed by n×1 convolution.
  - https://arxiv.org/abs/1512.00567
- Inception V4 (2016)
  - Residual connections (based on ResNet)
  - https://arxiv.org/abs/1602.07261

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

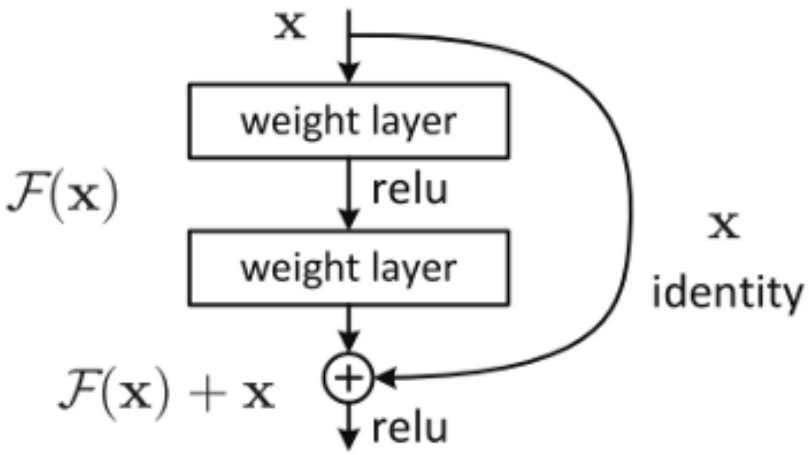


https://arxiv.org/abs/1311.2901

ResNet

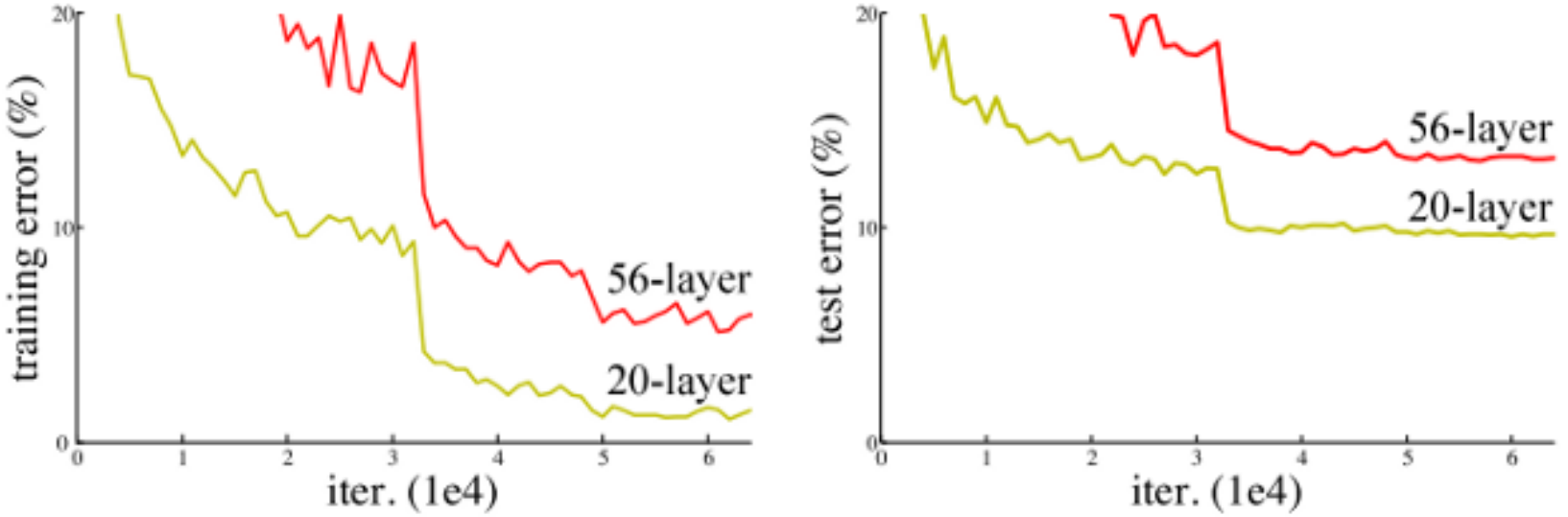
ResNet: Deep residual networks

- 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 ILSVRC2015 and COCO2015!
- Paper: https:/https://arxiv.org/abs/1512.03385
- Review: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035



ResNet

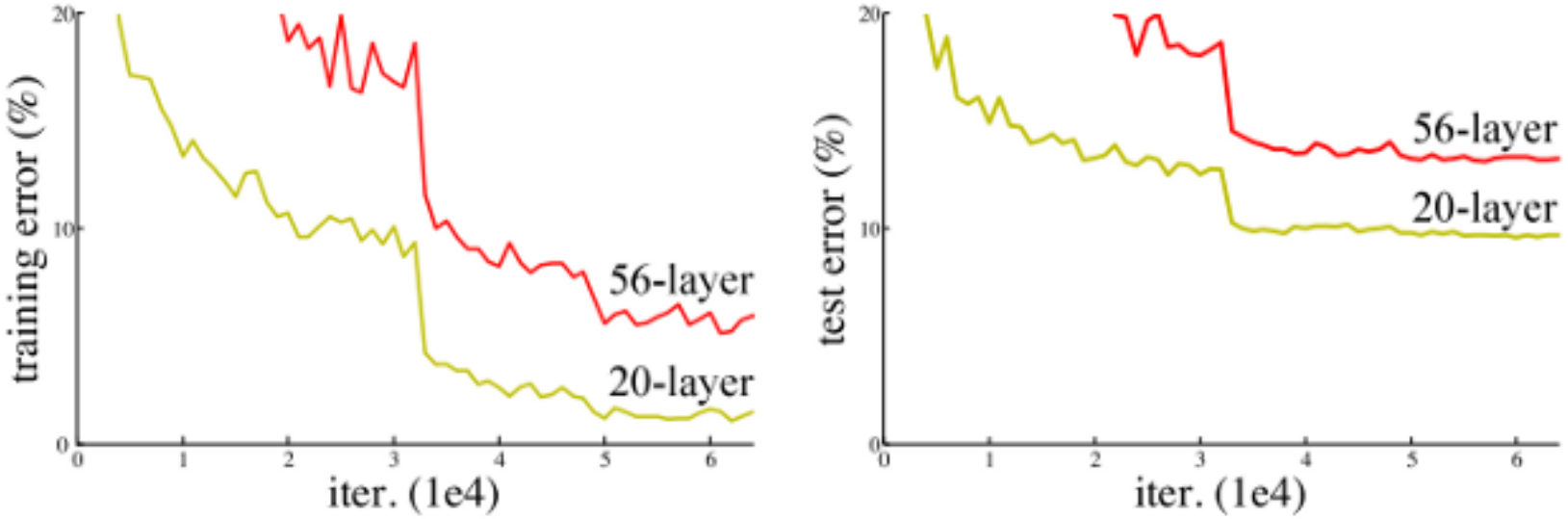
- **Motivation for going deeper:** Increase model capacity so that we can – in principle – represent any function.
- **Q:** What happens when we continue stacking deeper layers on a "plain" convolutional neural network?



Q: What's strange about these curves?

ResNet

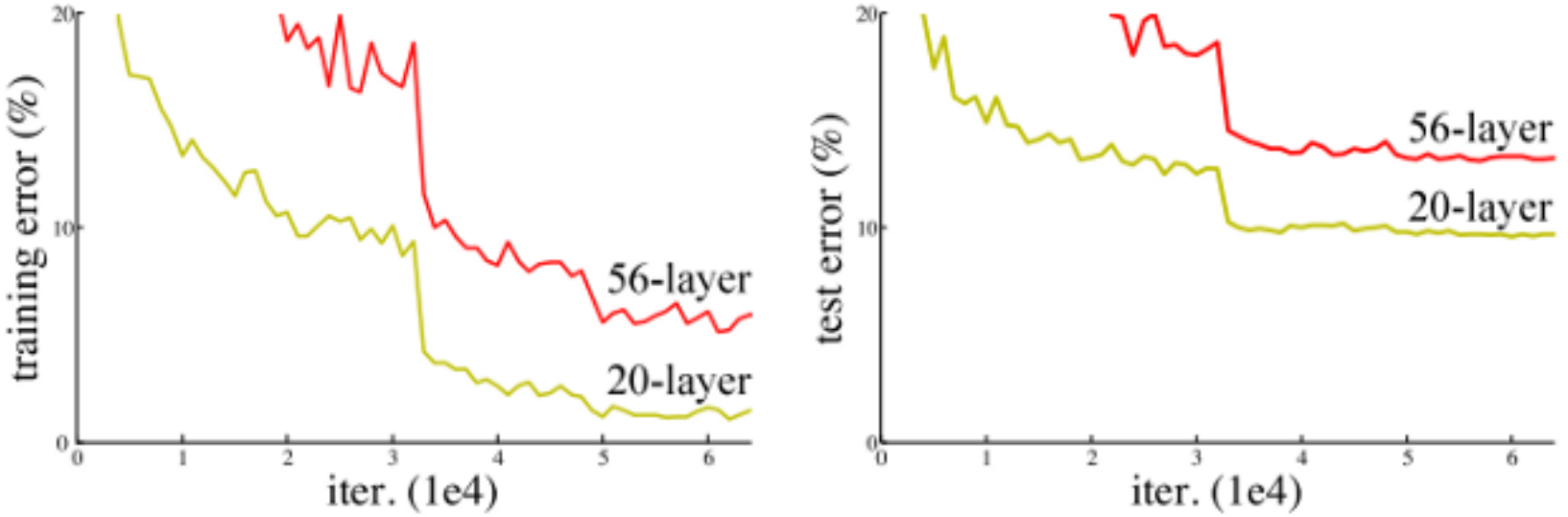
- **A:** The deeper model (56 layers) performs worse on both training and test, but it's not caused by overfitting!
- **Hypothesis:** the problem is an *optimization* problem – deeper models are harder to optimize due to vanishing gradients.



Q: What's strange about these curves?

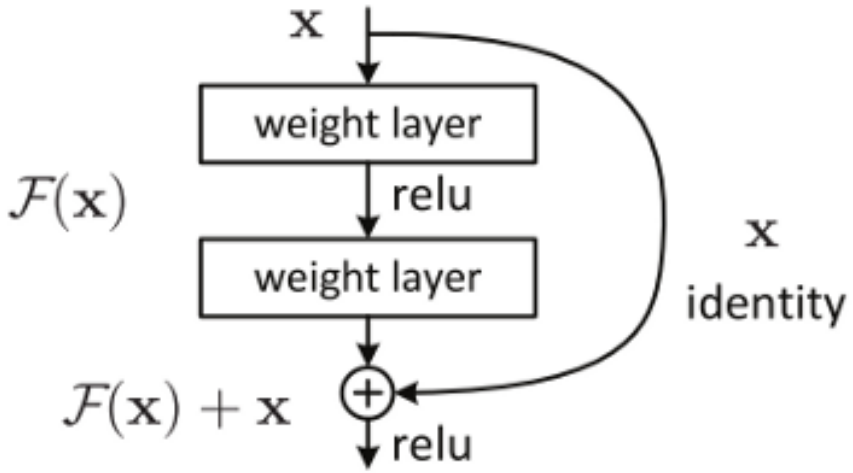
ResNet

- **Observation:** 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.



ResNet

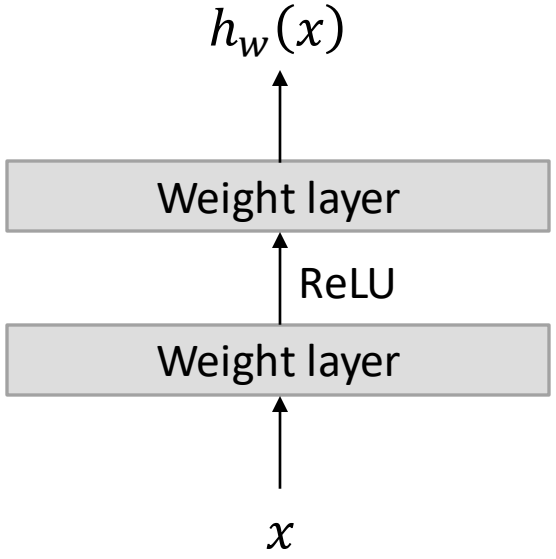
- The **core idea** of ResNet is introducing a so-called “identity shortcut connection” (or “skip connection”) that skips one or more layers, as shown in the figure.
- Stacking layers shouldn’t degrade the network performance, because we could simply stack identity mappings (layer that doesn’t do anything) upon the current network, and the resulting architecture would perform the same.
- The network fits a **residual mapping** instead of directly trying to fit a desired underlying mapping. The residual block on the right explicitly allows to do precisely that.



Residual block

Residual block

- **Recall:** Stacking layers (or blocks of layers) shouldn't degrade the network performance.
- **Solution:** Stack layers or blocks that are able to learn the identity mapping.
- We want our block to be able to learn the identity function: $h_w(x) = x$
- Note, we are implicitly assuming that the input and output are of the same dimensions.



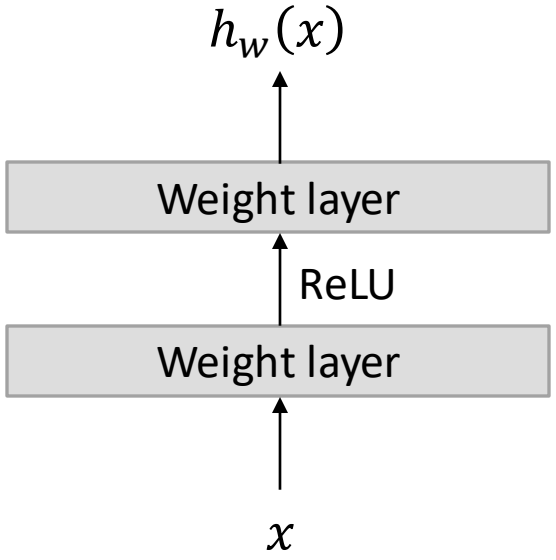
"Plain" block

Normally, $h_w(x)$ would consist of a few stacked non-linear layers.

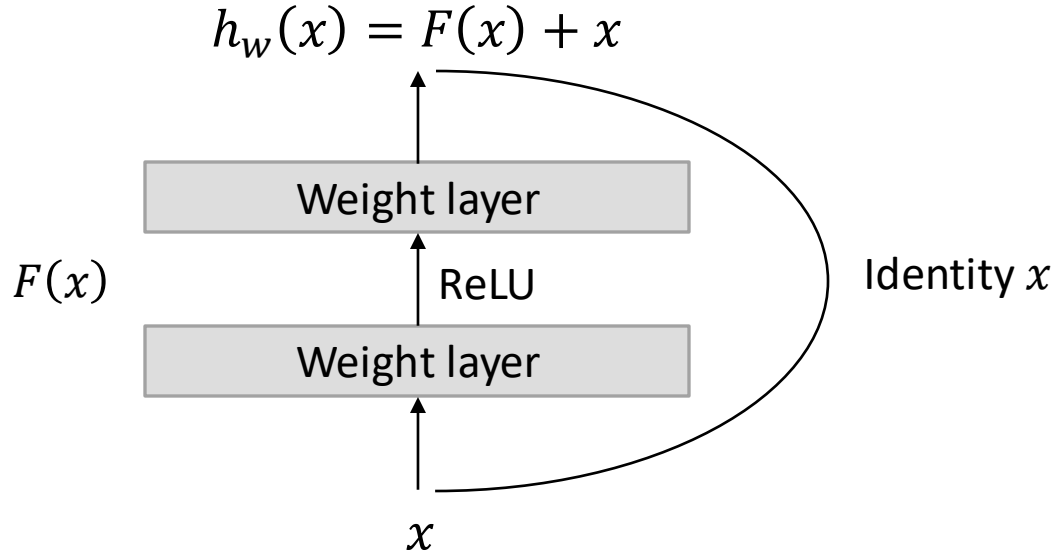
Having to learn $h_w(x) = x$ by a stack of non-linear layers seems overly complicated.

Residual block

- Let us instead define a residual function $F(x) = h_w(x) - x$, which can be reframed into $h_w(x) = F(x) + x$, where $F(x)$ is a block of stacked non-linear layers.
- The author's **hypothesis** is that it is easier to optimize the residual mapping function $F(x)$ than to optimize the original, unreferenced mapping $h_w(x)$. If the identity mapping is optimal, we can easily push the residuals to zero ($F(x) = 0$).



"Plain" block



Residual block

Skip connection and shape

- The skip connection (or “identity shortcut connection”) can be written as two lines of code:

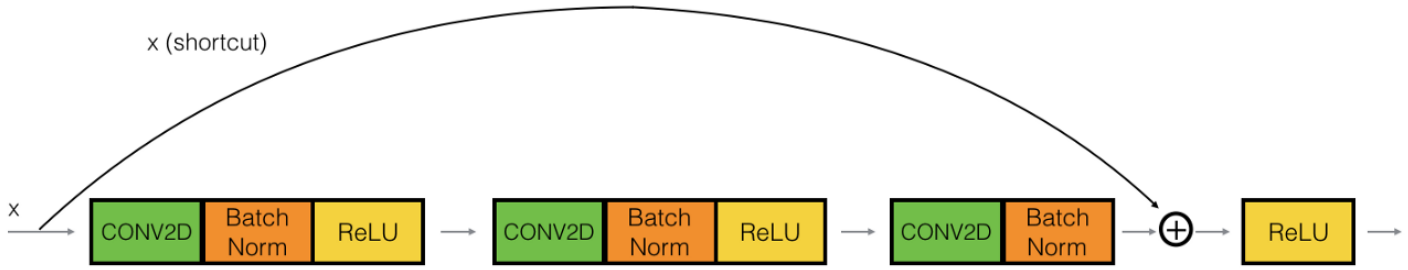
```
X_shortcut = X # Store the initial value of X in a variable
## Perform block of convolution + batch norm operations on X
X = Add()([X, X_shortcut]) # SKIP Connection
```

- Important consideration: You can add `X` and `X_shortcut` only if they have the same shape.

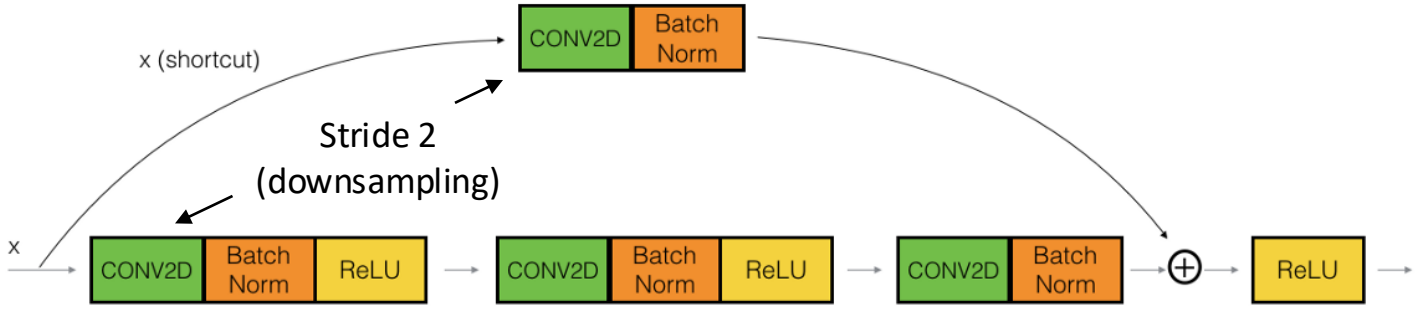
https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33

Skip connection and shape

- **Type 1:** If the block operations preserve shape, then we can simply add them as shown below.

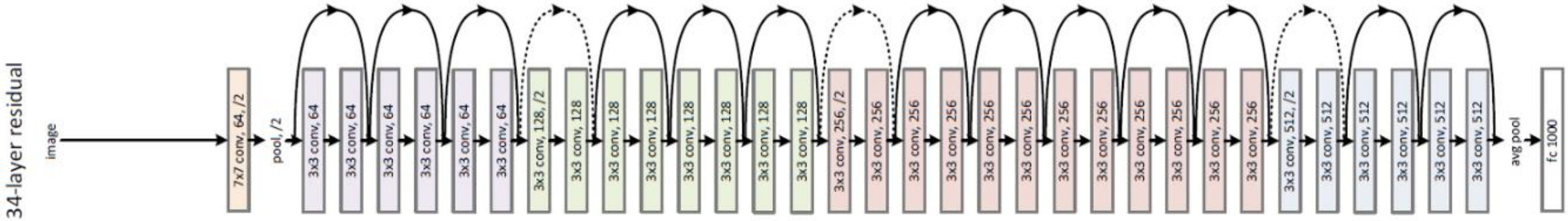


- **Type 2:** Otherwise, the `X_shortcut` goes through a convolution layer chosen such that the output from it is the same dimension as the output from the convolution block:



https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33

Full architecture (ResNet-34)



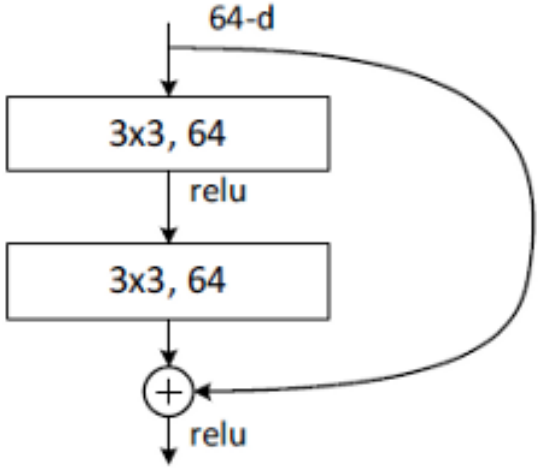
- Stack residual blocks
- Additional conv layer + max pool at the beginning
- No FC layers at the end (only FC 1000 to output classes)
- Global average pooling after last conv layer

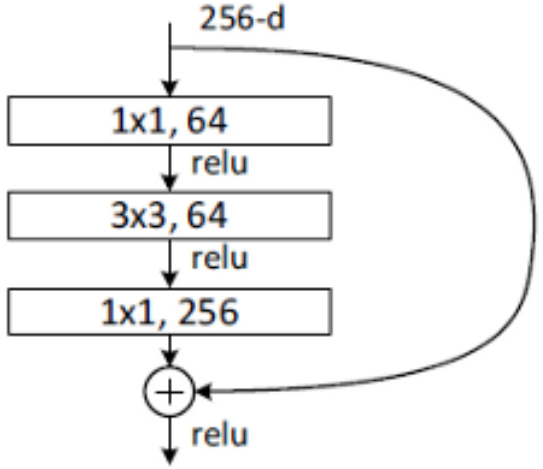
- Every residual block has two 3x3 conv layers
- Periodically, double number of filters and downsample spatially using stride 2 (marked with /2)
- Solid skip connections are Type 1 (preserve shape)
- Dashed skip connections are Type 2 (modify shape by downsampling)

Bottleneck layers

- For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to 1x1 convolutions used in GoogLeNet)
- 1×1 convolution reduces the number of connections (parameters) while not degrading the performance of the network so much.
- With the bottleneck design, 34-layer ResNet becomes 50-layer ResNet.

Basic block



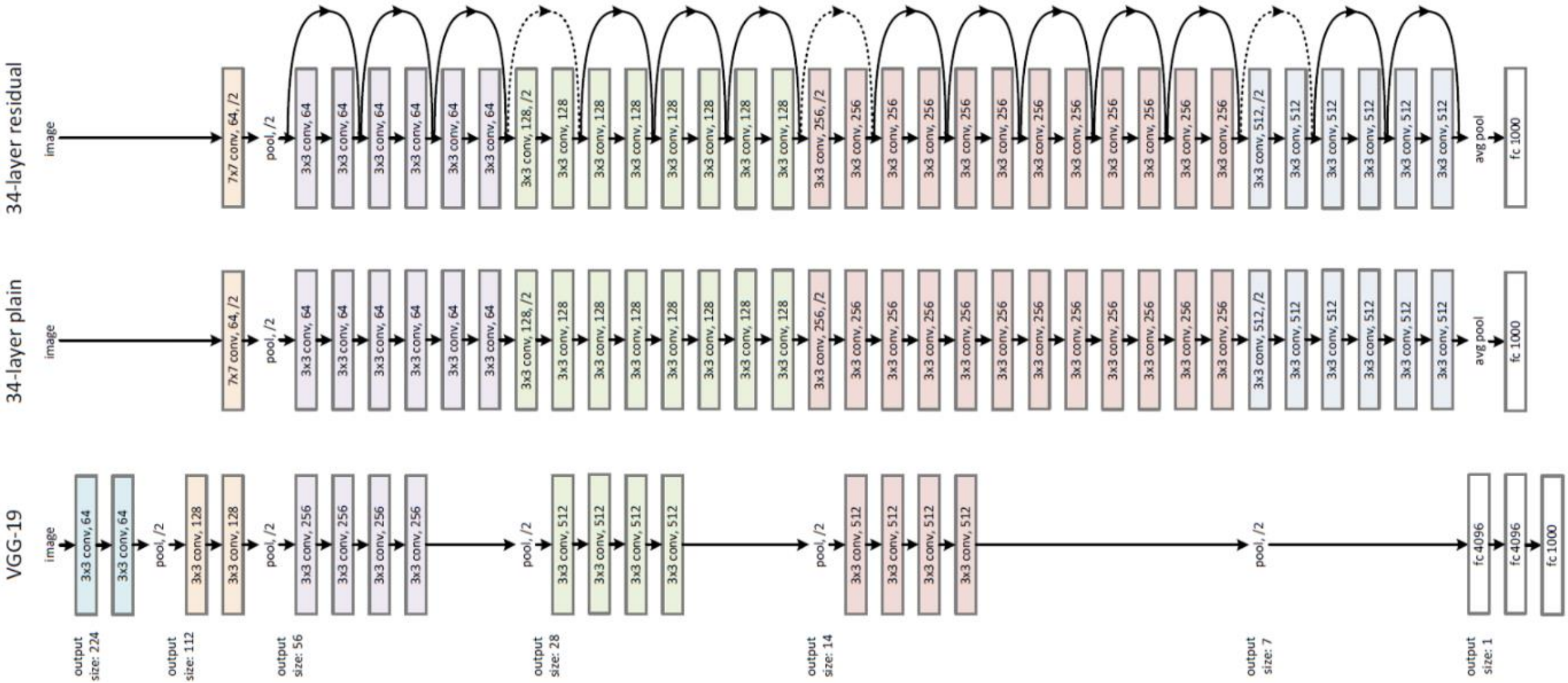


Bottleneck block

Variants

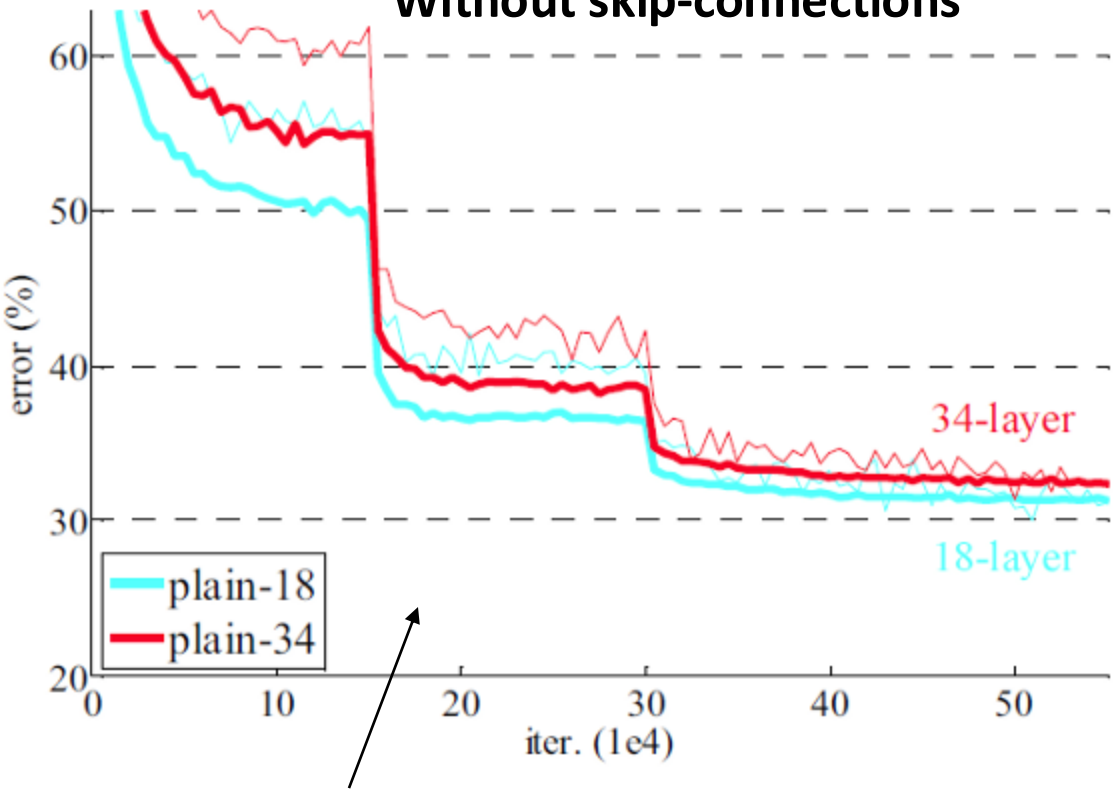
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Comparison with/without skip



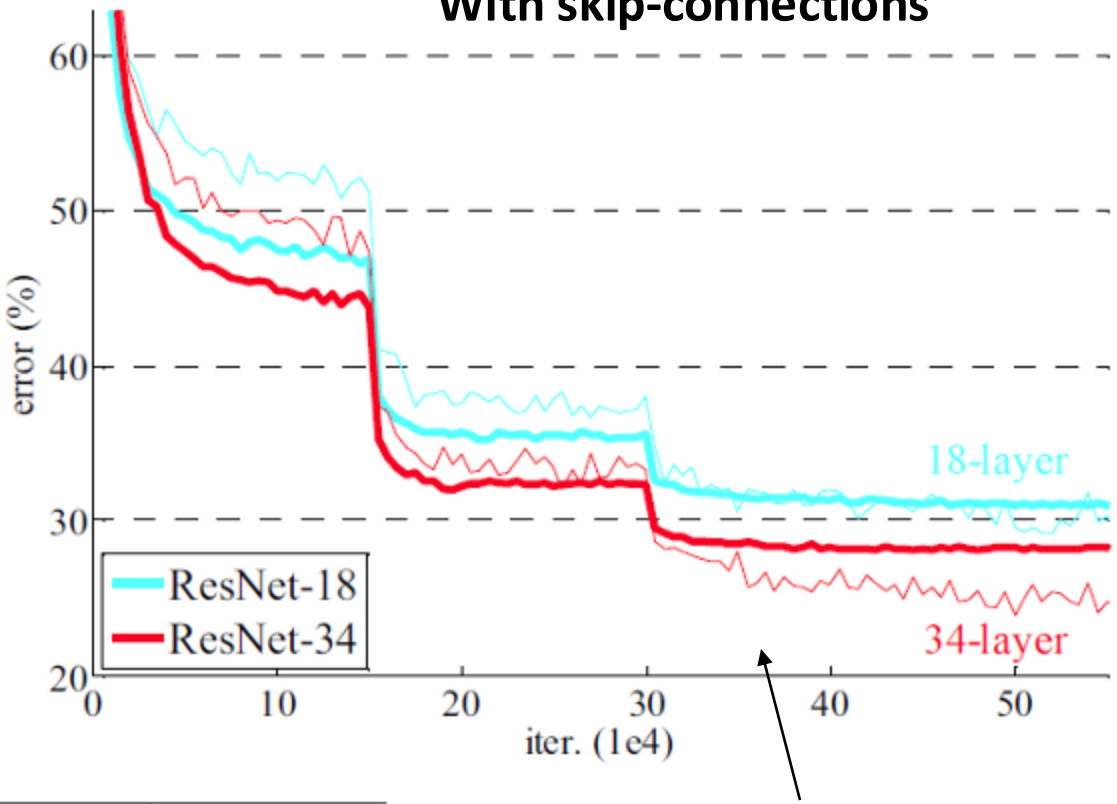
Comparison with/without skip

Without skip-connections



34-layer model performs worse than 18-layer model – vanishing gradient problem!

With skip-connections

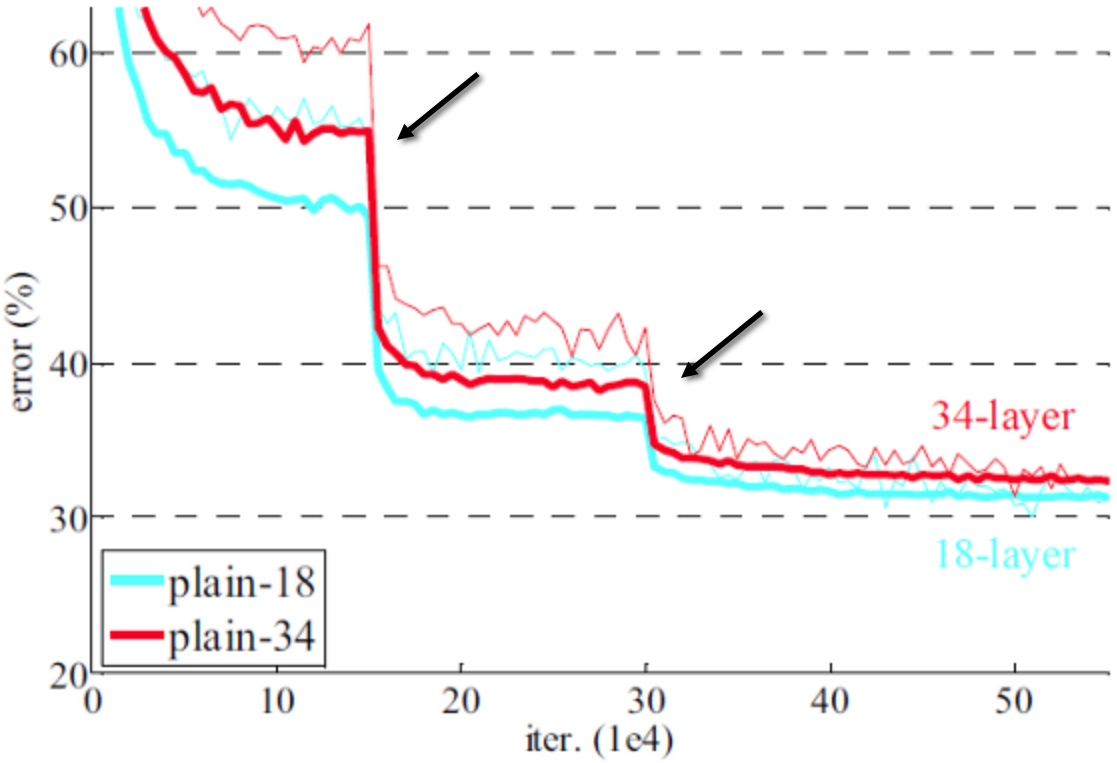


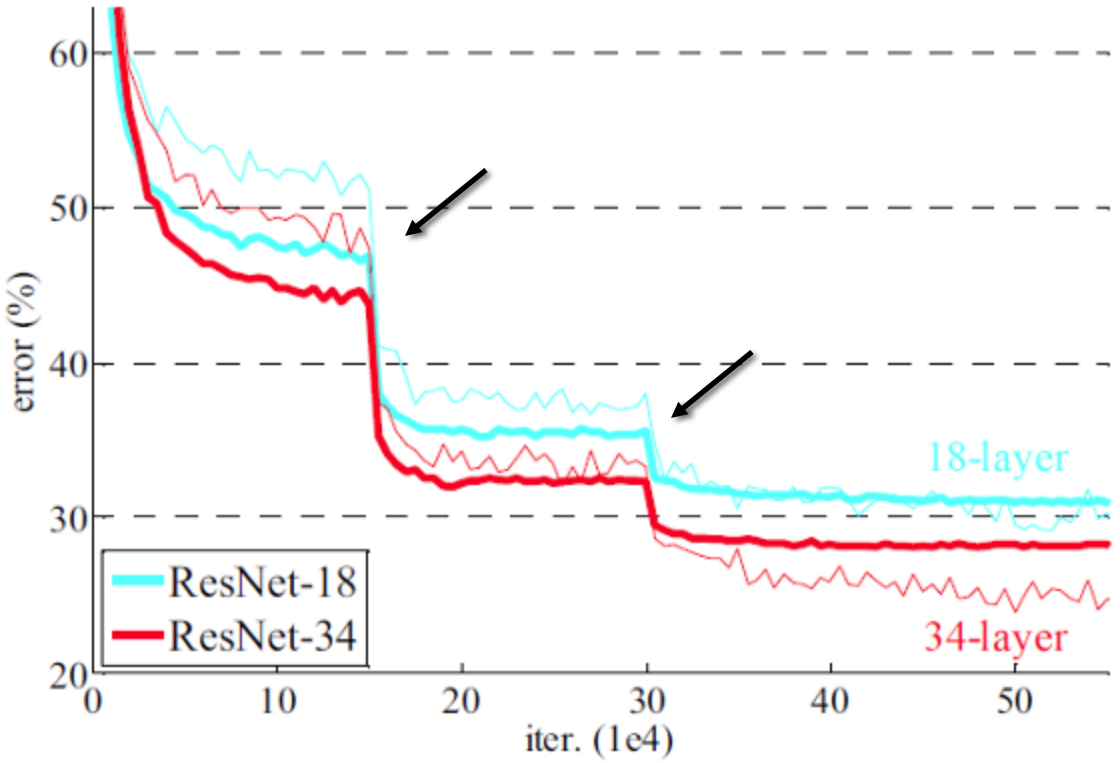
Much better and consistent with what we want.

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8

Question (not related to today's topic)





Q: What causes these dips in the loss?

A: Learning rate decay

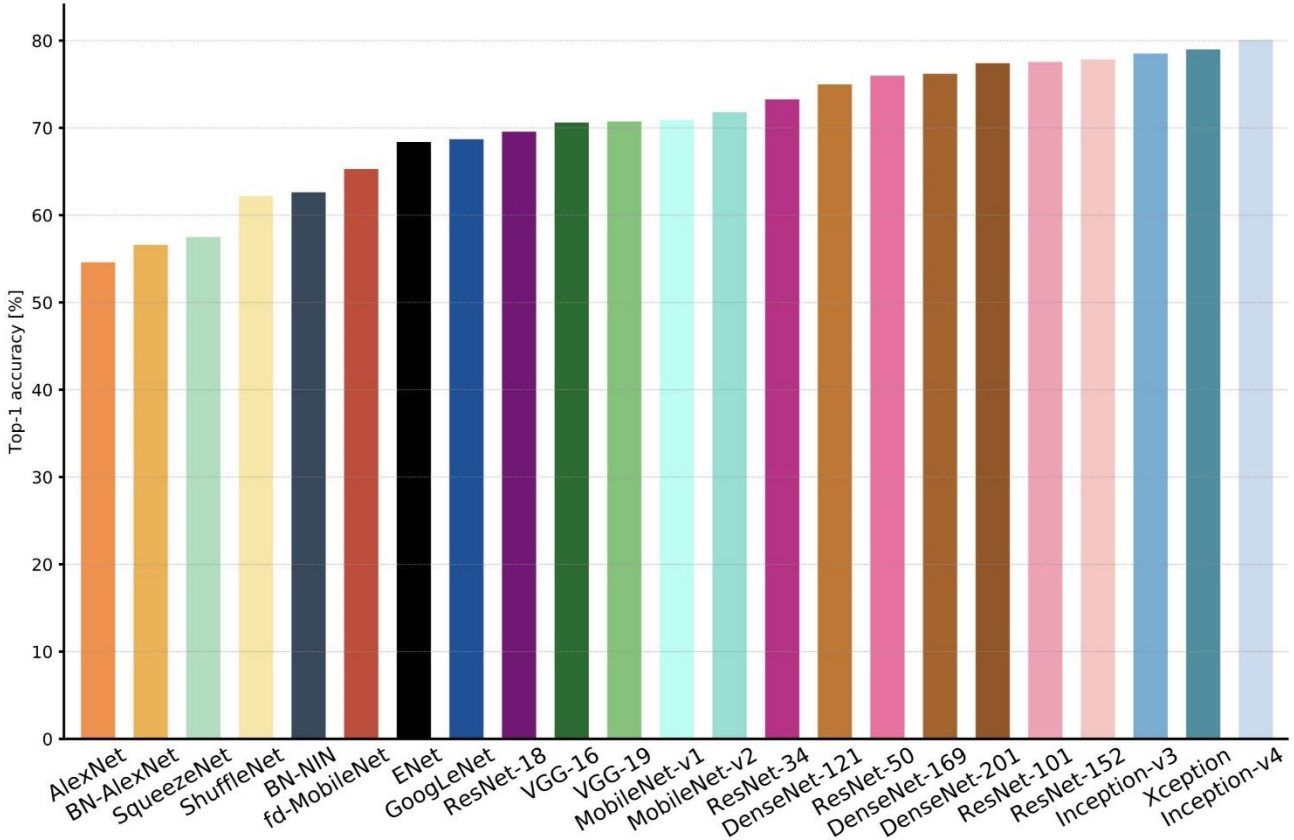
Procedure: Loss decreases until it reaches a plateau. Decrease learning rate to tune in on the optimum.

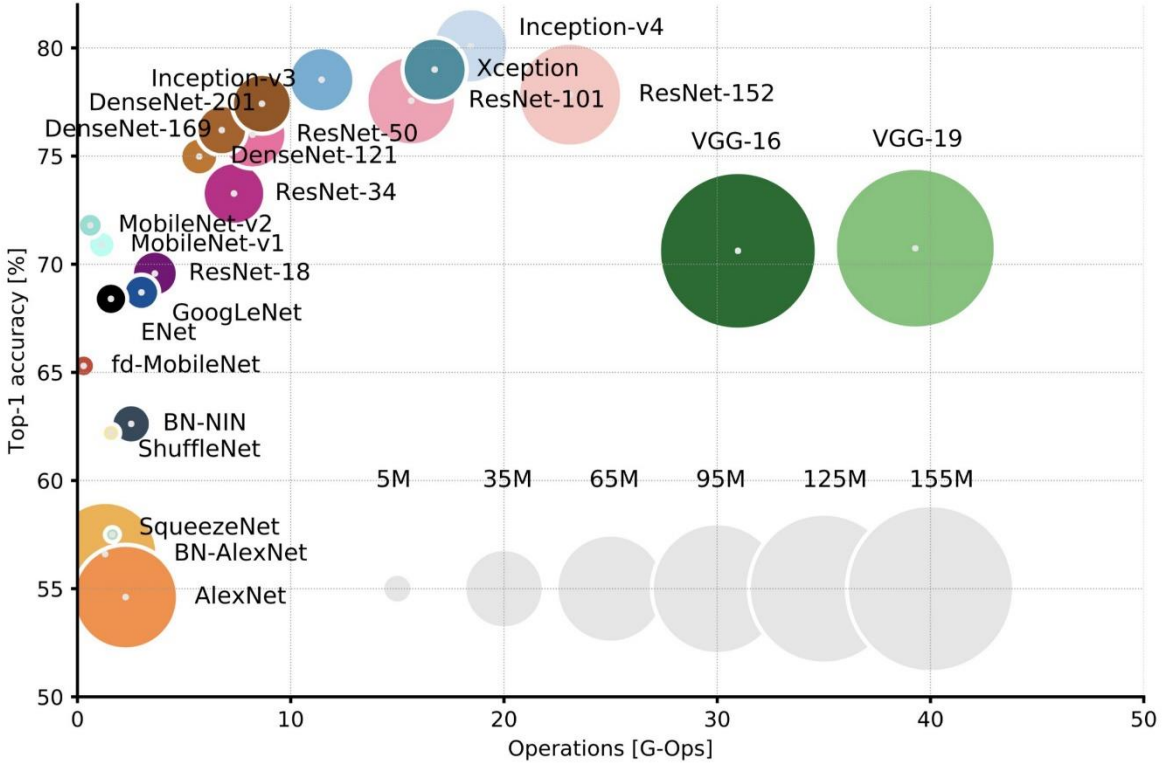
https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8

Training ResNet in practise

- 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

Comparing complexity

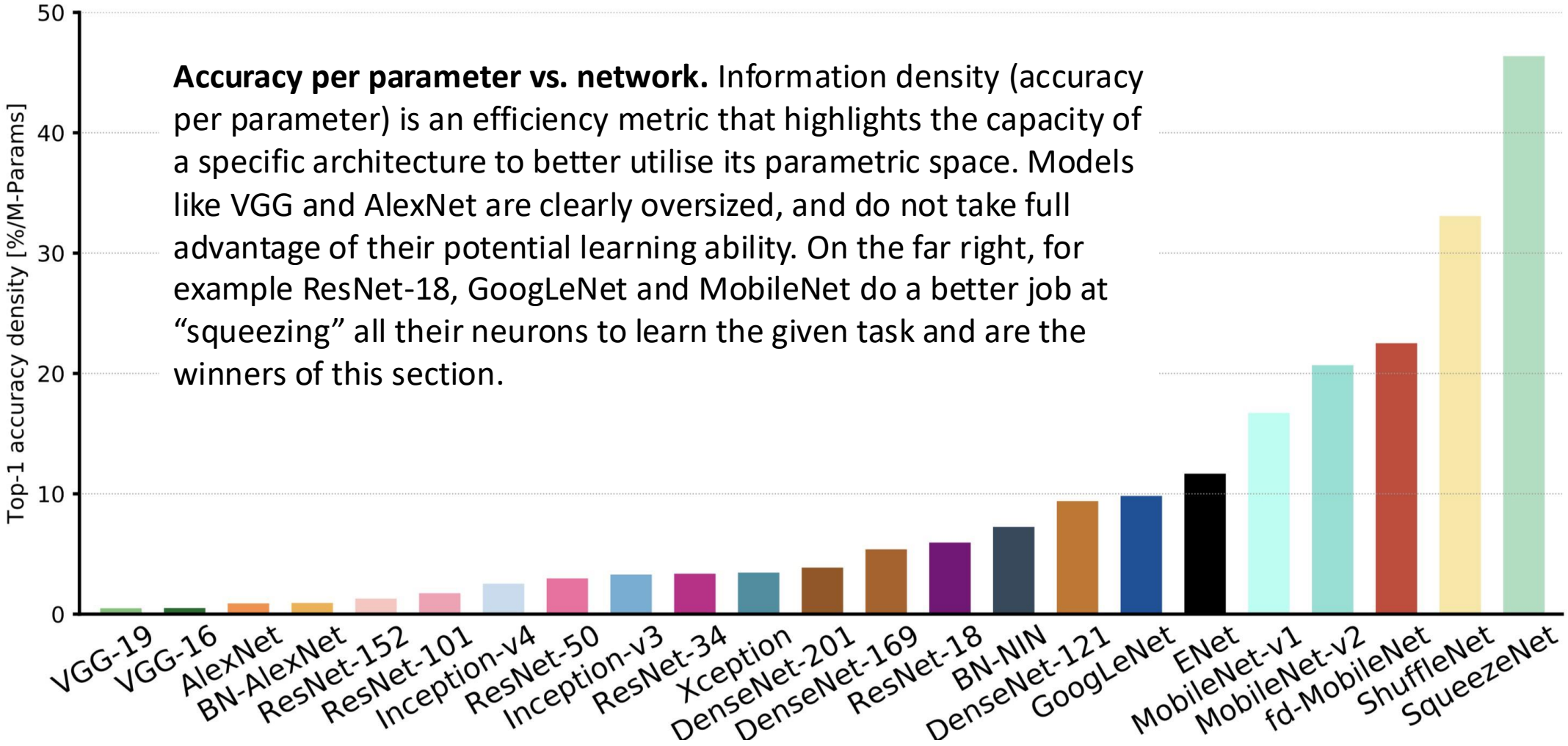




Top-1 accuracy versus number of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters

https://medium.com/@culurciello/analysis-of-deep-neural-networks-dcf398e71aae

Comparing complexity



Accuracy per parameter vs. network. Information density (accuracy per parameter) is an efficiency metric that highlights the capacity of a specific architecture to better utilise its parametric space. Models like VGG and AlexNet are clearly oversized, and do not take full advantage of their potential learning ability. On the far right, for example ResNet-18, GoogLeNet and MobileNet do a better job at "squeezing" all their neurons to learn the given task and are the winners of this section.

https://medium.com/@culurciello/analysis-of-deep-neural-networks-dcf398e71aae

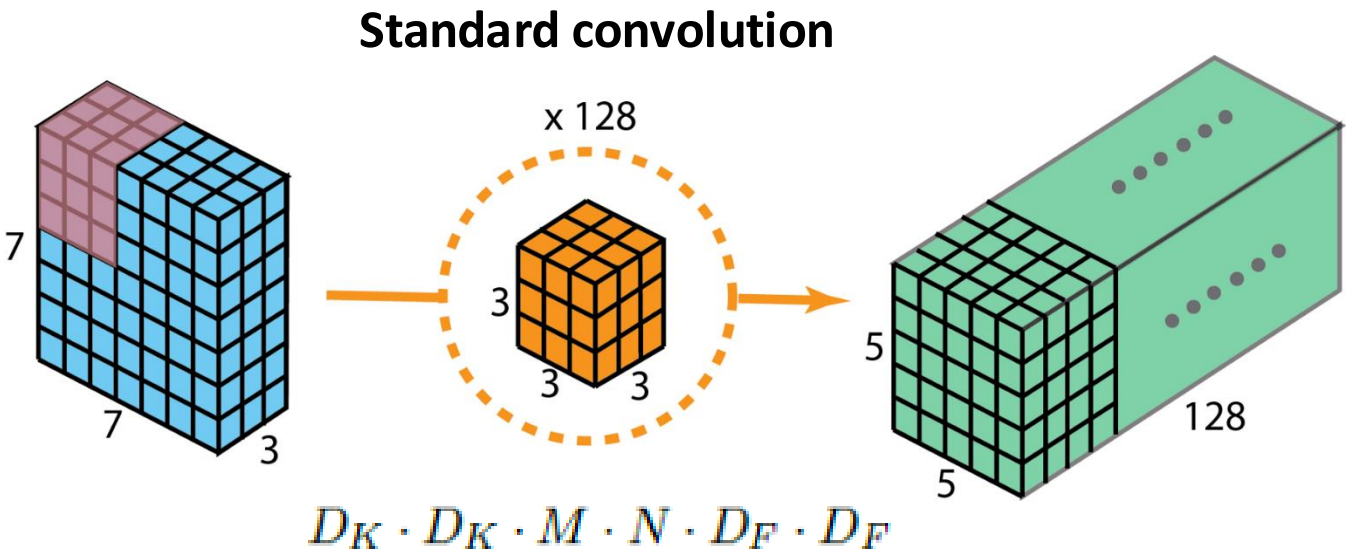
MobileNet

Basic idea

- Depthwise Separable Convolution is used to reduce the model size and complexity
- MobileNet is particularly useful for mobile and embedded vision applications.
- **Smaller model size**: Fewer number of parameters
- **Lower complexity**: Fewer multiplications and additions (Multi-Adds)
- Review: https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69
- Paper: https://arxiv.org/abs/1704.04861

Standard convolution

- https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

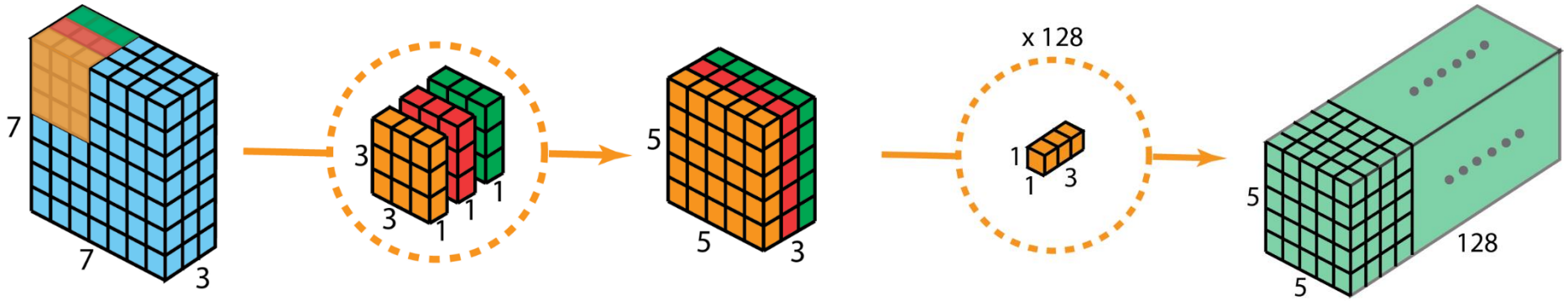


$$3 \cdot 3 \cdot 3 \cdot 128 \cdot 5 \cdot 5 = 86{,}400$$

Depthwise Separable Convolution

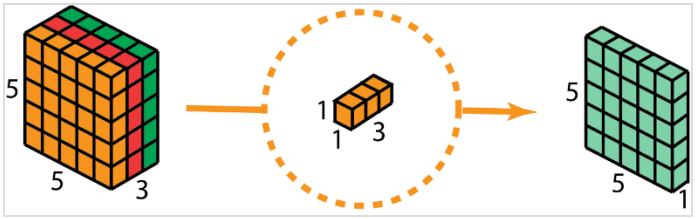
- https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

Depthwise Separable Convolution



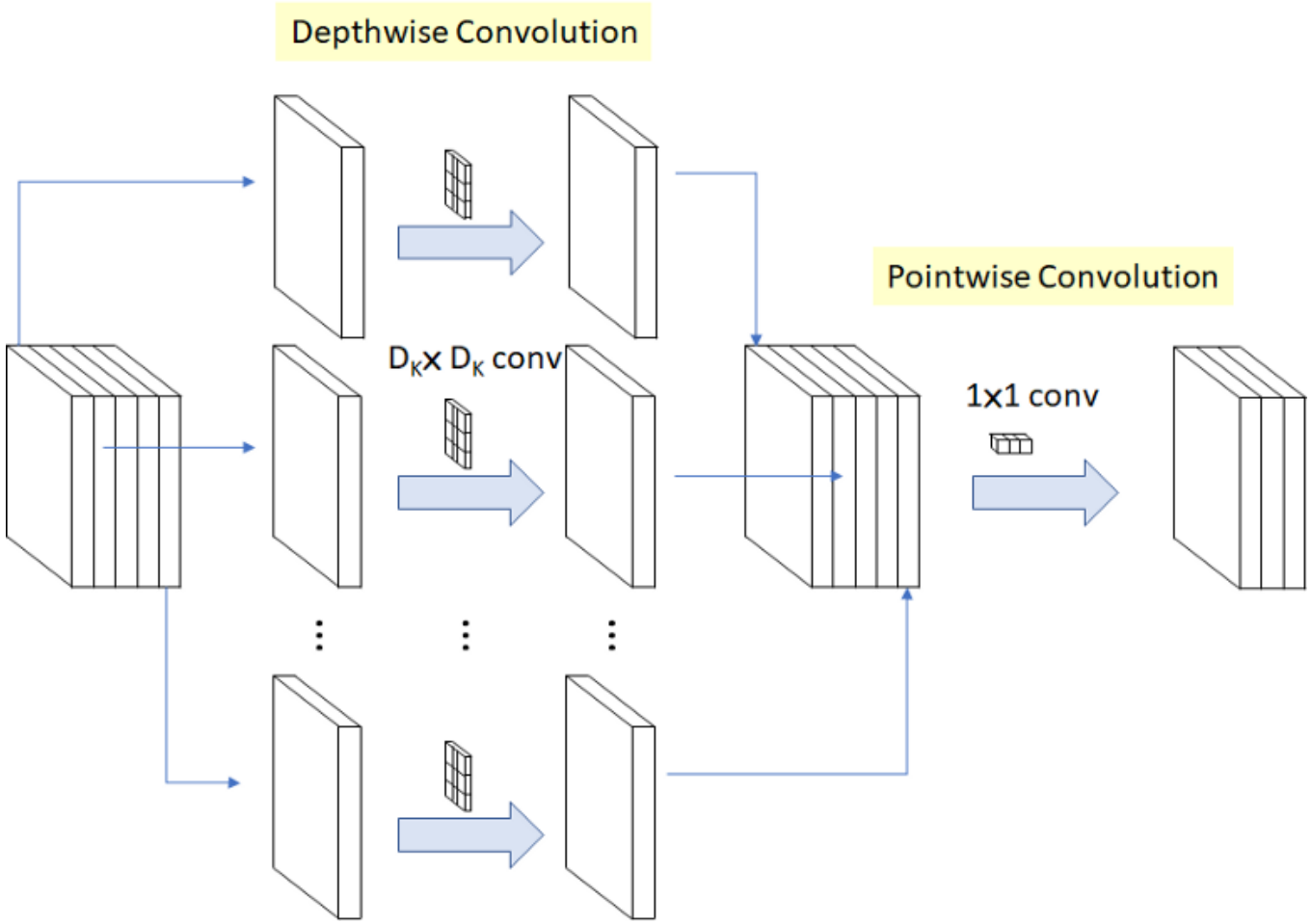
$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

$$3 \cdot 3 \cdot 3 \cdot 5 \cdot 5 + 3 \cdot 128 \cdot 5 \cdot 5 =$$
$$675 + 9{,}600 = 10{,}275$$



Depthwise Separable Convolution

- Depthwise separable convolution is a **depthwise convolution** followed by a **pointwise convolution** as follows:
- **Depthwise convolution** is the channel-wise $D_K \times D_K$ spatial convolution. Suppose in the figure, we have 5 channels, then we will have 5 $D_K \times D_K$ spatial convolution.
- **Pointwise convolution** is a 1×1 convolution to change the number of channels.



Depthwise Separable Convolution

- The operation cost is: $D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$

- where M: Number of input channels, N: Number of output channels, D_K: Kernel size, and D_F: Feature map size.
- For standard convolution, it is: $D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$

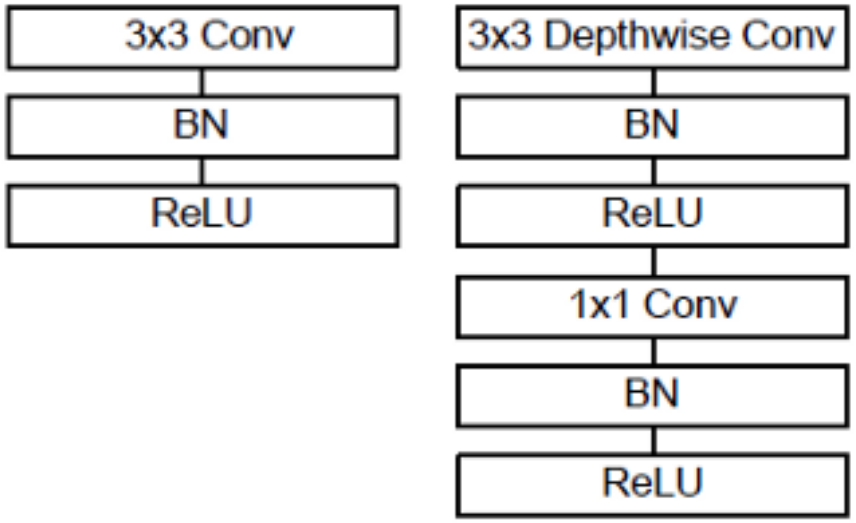
- Thus, the computation reduction is:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

- When $D_K \times D_K$ is 3×3, 8 to 9 times less computation can be achieved, but with only small reduction in accuracy.

Network architecture

- It is noted that Batch Normalization (BN) and ReLU are applied after each convolution:



Standard Convolution (Left), Depthwise separable convolution (Right) with BN and ReLU

Table 1. MobileNet Body Architecture

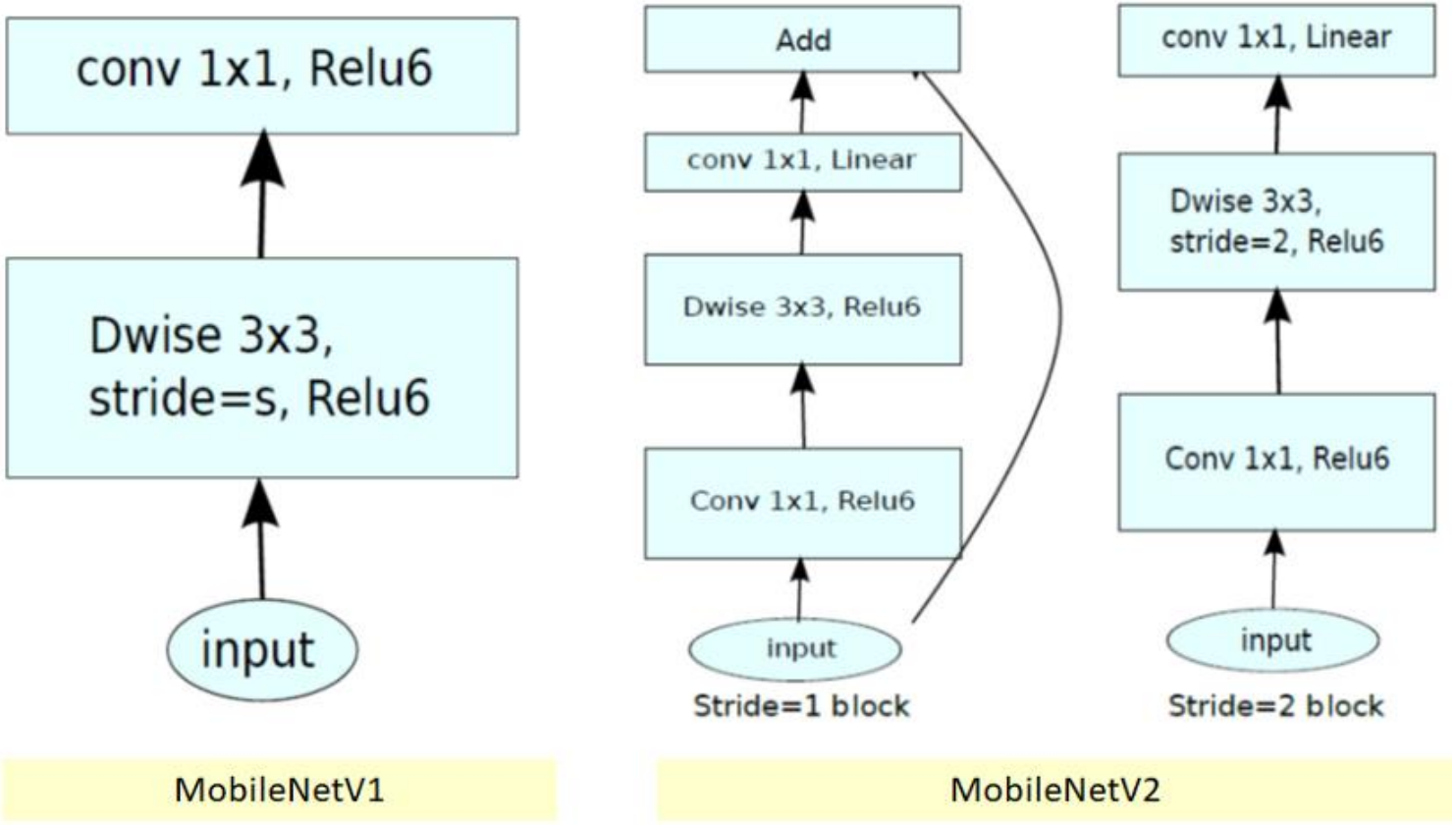
Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
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	$1 \times 1 \times 1000$

MobileNet summary

- Separable convolutions replace standard convolutions by factorizing them into a depthwise convolution and a 1x1 convolution.
- Much more efficient with little loss in accuracy.
- Follow-up MobileNetV2 work in 2018

MobileNet v2

- **Q:** Can you guess what has been added?
- **A:** ResNet blocks

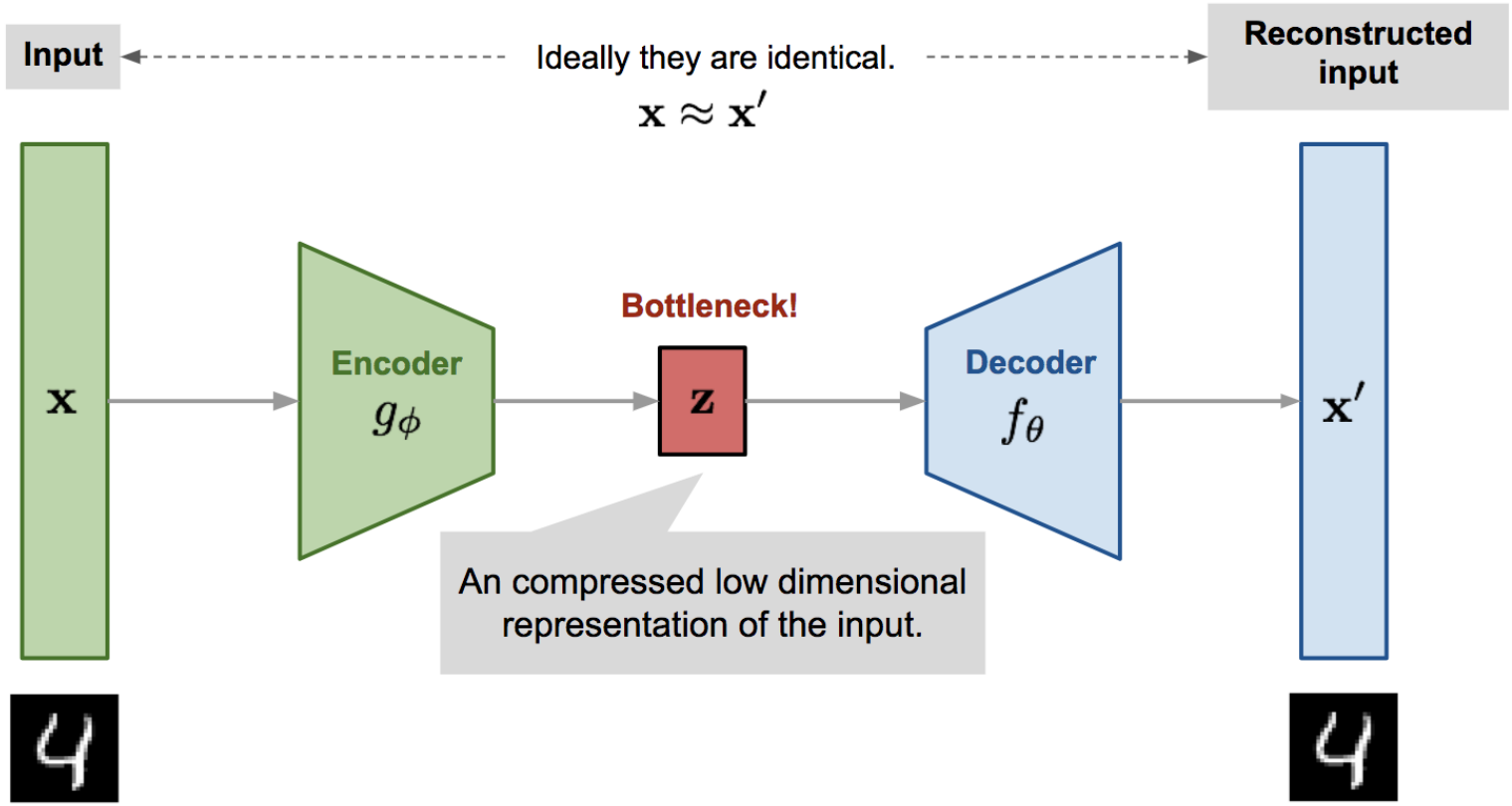


https://towardsdatascience.com/review-mobilenetv2-light-weight-model-image-classification-8febb490e61c

Other types of architectures

Autoencoder

- An autoencoder is an unsupervised machine learning algorithm that takes an image as input and tries to reconstruct it back using a fewer number of bits from the latent space representation.



https://medium.com/analytics-vidhya/building-a-convolutional-autoencoder-using-keras-using-conv2dtranspose-ca403c8d144e

Convolutional autoencoder

- The architecture consists of an **encoder** with convolutional layers, and a **decoder** with upsamling layers (sometimes mistakenly called deconvolution layers).
- At the **bottleneck** the image is converted into a vector or a volume.
- Different options for upsampling, but most often **transposed convolution**.
- You can read more about transposed convolution here (section 6): https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              
=================================================================
input_4 (InputLayer)         (None, 28, 28, 1)         
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 26, 26, 32)        
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 32)        
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 11, 11, 64)        
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 64)          
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 3, 3, 64)          
_________________________________________________________________
flatten_4 (Flatten)          (None, 576)               
_________________________________________________________________
dense_4 (Dense)              (None, 49)                
_________________________________________________________________
reshape_4 (Reshape)          (None, 7, 7, 1)           
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 14, 14, 64)        
_________________________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 64)        
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 28, 28, 64)        
_________________________________________________________________
batch_normalization_9 (Batch (None, 28, 28, 64)        
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 28, 28, 32)        
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 28, 28, 1)         
=================================================================
Total params: 140,850 Trainable params: 140,594 N
```

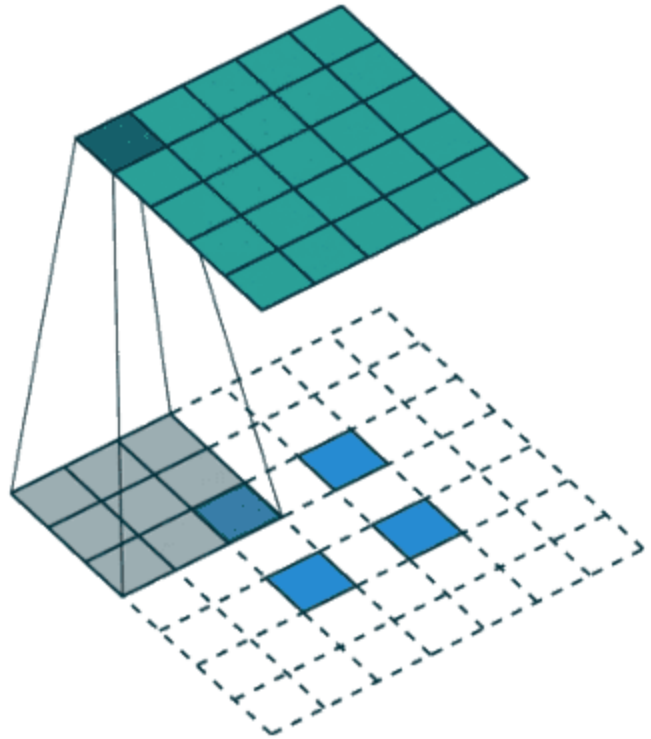
Encoder

Bottleneck

Decoder

Convolutional autoencoder

- The architecture consists of an **encoder** with convolutional layers, and a **decoder** with upsamling layers (sometimes mistakenly called deconvolution layers).
- At the **bottleneck** the image is converted into a vector or a volume.
- Different options for upsampling, but most often **transposed convolution**.
- You can read more about transposed convolution here (section 6): https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215



Up-sampling a 2 x 2 input to a 5 x 5 output.

Convolutional autoencoder in Keras

```
#ENCODER
inp = Input((28, 28,1))
e = Conv2D(32, (3, 3), activation='relu')(inp)
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e)
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e)
l = Flatten()(e)
l = Dense(49, activation='relu')(l)
```

```
#DECODER
d = Reshape((7,7,1))(l)
d = Conv2DTranspose(64,(3, 3), strides=2, activation='relu', padding='same')(d)
d = BatchNormalization()(d)
d = Conv2DTranspose(64,(3, 3), strides=2, activation='relu', padding='same')(d)
d = BatchNormalization()(d)
d = Conv2DTranspose(32,(3, 3), activation='relu', padding='same')(d)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(d)
ae = Model(inp, decoded)
```

https://medium.com/analytics-vidhya/building-a-convolutional-autoencoder-using-keras-using-conv2dtranspose-ca403c8d144e

Convolutional autoen

```
#ENCODER
inp = Input((28, 28,1))
e = Conv2D(32, (3, 3), activation='relu')(i
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3), activation='relu')(e
l = Flatten()(e)
l = Dense(49, activation='relu')(l)
```

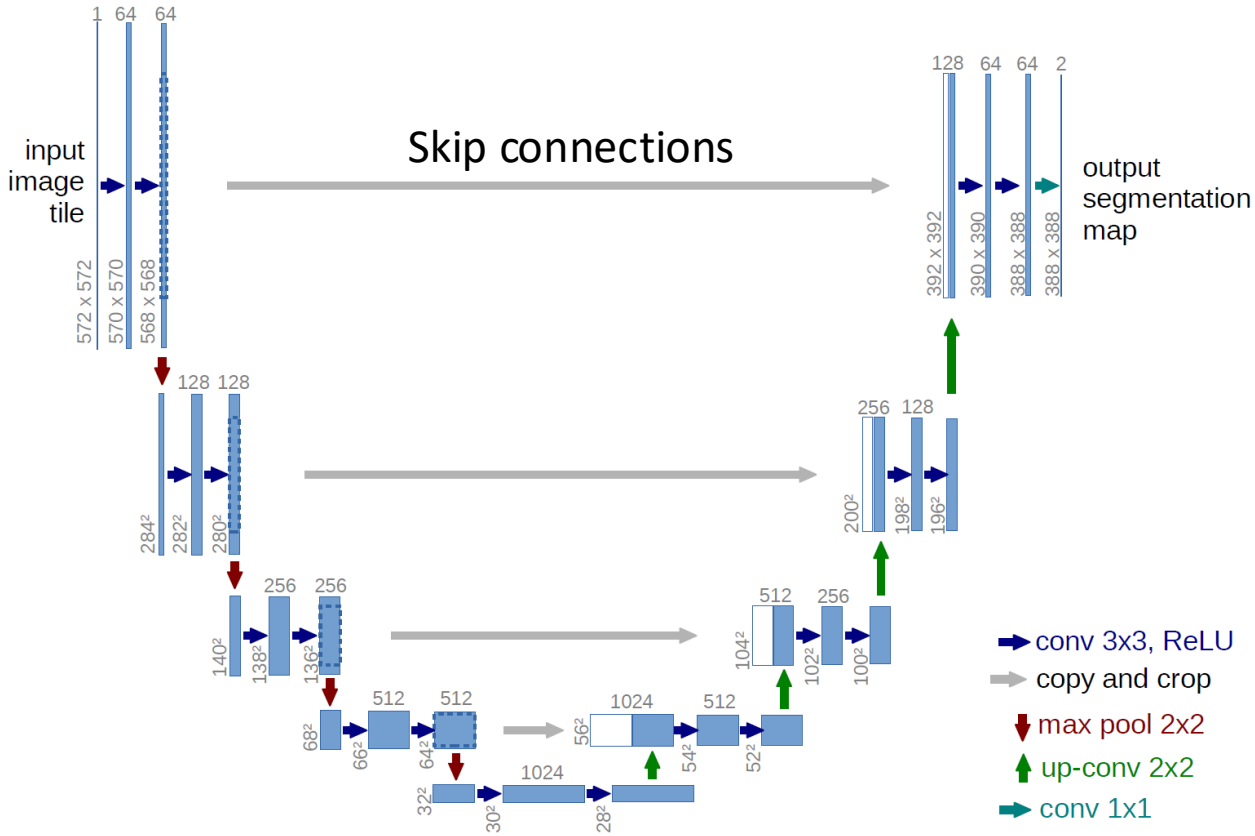
```
#DECODER
d = Reshape((7,7,1))(l)
d = Conv2DTranspose(64,(3, 3), strides=2, a
d = BatchNormalization()(d)
d = Conv2DTranspose(64,(3, 3), strides=2, a
d = BatchNormalization()(d)
d = Conv2DTranspose(32,(3, 3), activation='
decoded = Conv2D(1, (3, 3), activation='sig
ae = Model(inp, decoded)
```

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_4 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_4 (Dense)              (None, 49)                28273
_________________________________________________________________
reshape_4 (Reshape)          (None, 7, 7, 1)           0
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 14, 14, 64)        640
_________________________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 28, 28, 64)        36928
_________________________________________________________________
batch_normalization_9 (Batch (None, 28, 28, 64)        256
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 28, 28, 32)        18464
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 28, 28, 1)         289
=================================================================
Total params: 140,850 Trainable params: 140,594 Non-trainable params:
```

https://medium.com/analytics-vidhya/building-a-convolutional-autoencoder-using-keras-using-conv2dtranspose-ca403c8d144e

U-Net

- The convolutional autoencoder tends to **lose fine image** details at the bottleneck.
- U-Net uses the feature maps of the encoder to expand the bottleneck vector to an image (via **skip connections**).
- Hence, expansion combines upsampling from the previous decoding layer with corresponding encoding layer.
- This preserves the structural integrity of the image, allowing to maintain fine details.
- Note: U-Net was originally designed for image segmentation.
- Paper: https://arxiv.org/abs/1505.04597



https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

U-Net

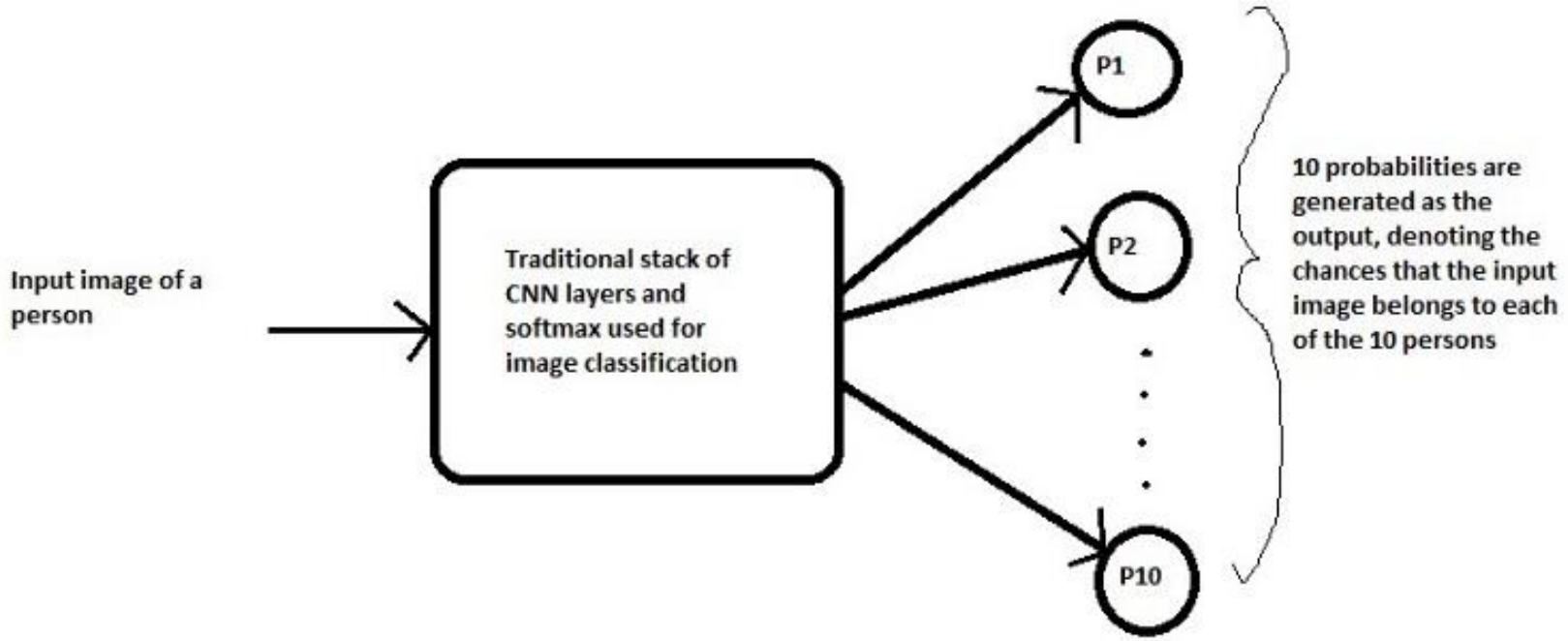
- The interesting part of the U-Net is at the concatenation layers.
- This is where the representation from the previous upsampling layer (say u6) is combined with the feature map of the encoder of the same shape (say c4)

```
# Contracting Path (Encoder)
c1 = conv2d_block(input_img, n_filters * 1, kernel_size = 3, batchnorm = batchnorm)
p1 = MaxPooling2D((2, 2))(c1)
c2 = conv2d_block(p1, n_filters * 2, kernel_size = 3, batchnorm = batchnorm)
p2 = MaxPooling2D((2, 2))(c2)
c3 = conv2d_block(p2, n_filters * 4, kernel_size = 3, batchnorm = batchnorm)
p3 = MaxPooling2D((2, 2))(c3)
c4 = conv2d_block(p3, n_filters * 8, kernel_size = 3, batchnorm = batchnorm)
p4 = MaxPooling2D((2, 2))(c4)
c5 = conv2d_block(p4, n_filters = n_filters * 16, kernel_size = 3, batchnorm = batchnorm)
# Expansive Path (Decoder)
u6 = Conv2DTranspose(n_filters * 8, (3, 3), strides = (2, 2), padding = 'same')(c5)
→ u6 = concatenate([u6, c4])
c6 = conv2d_block(u6, n_filters * 8, kernel_size = 3, batchnorm = batchnorm)
u7 = Conv2DTranspose(n_filters * 4, (3, 3), strides = (2, 2), padding = 'same')(c6)
→ u7 = concatenate([u7, c3])
c7 = conv2d_block(u7, n_filters * 4, kernel_size = 3, batchnorm = batchnorm)
u8 = Conv2DTranspose(n_filters * 2, (3, 3), strides = (2, 2), padding = 'same')(c7)
→ u8 = concatenate([u8, c2])
c8 = conv2d_block(u8, n_filters * 2, kernel_size = 3, batchnorm = batchnorm)
u9 = Conv2DTranspose(n_filters * 1, (3, 3), strides = (2, 2), padding = 'same')(c8)
→ u9 = concatenate([u9, c1])
c9 = conv2d_block(u9, n_filters * 1, kernel_size = 3, batchnorm = batchnorm)
outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)
```

https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

Siamese networks – motivation

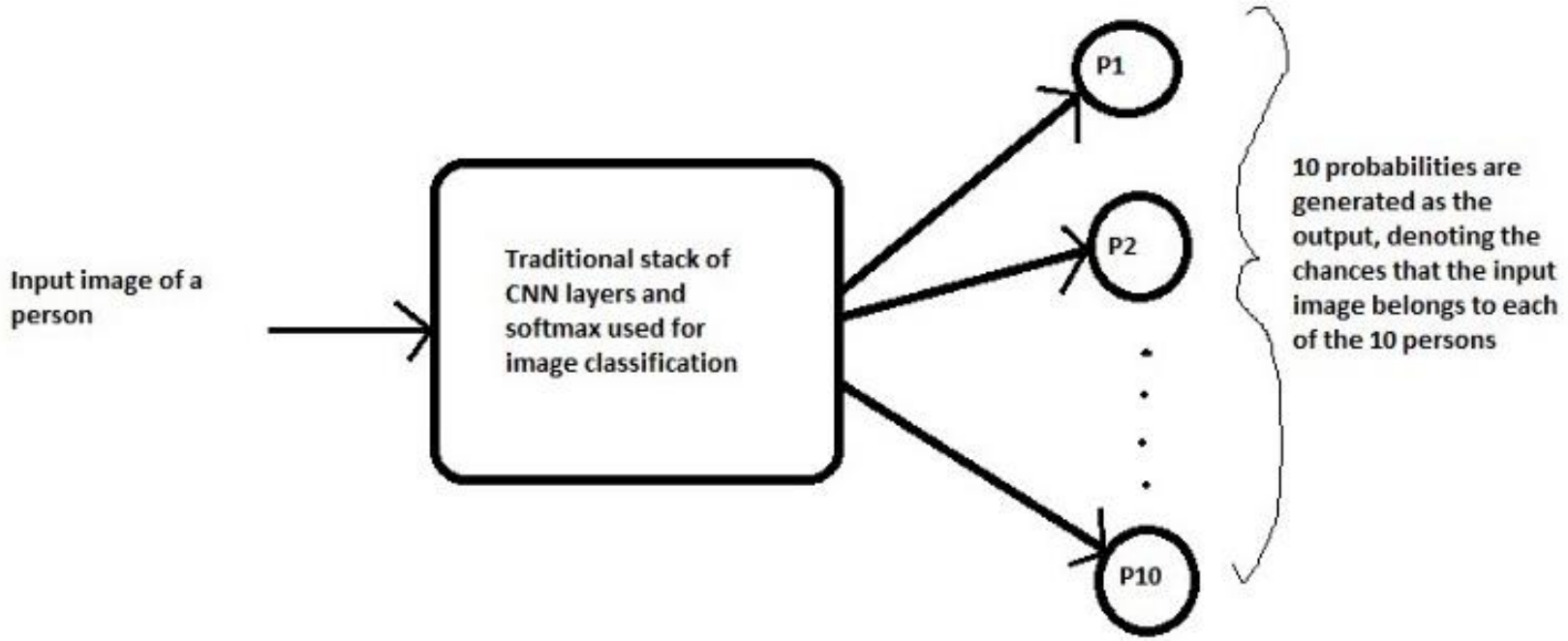
- Assume that we want to build a face recognition system for a small organization with only 10 employees (small numbers keep things simple).
- Using a traditional classification approach, we might come up with a system that looks as below:



https://towardsdatascience.com/one-shot-learning-with-siamese-networks-using-keras-17f34e75bb3d

Siamese networks – motivation

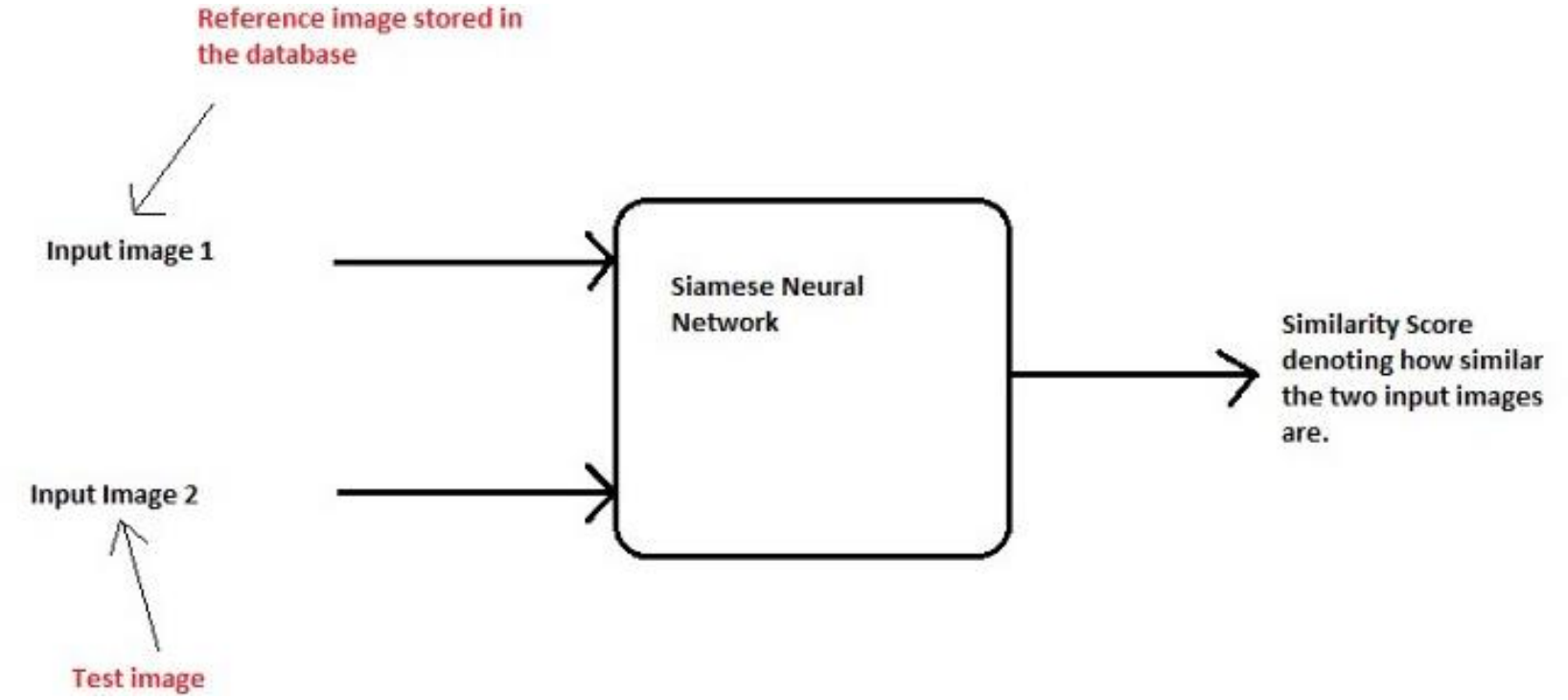
- **Problems:**
  - a) To train such a system, we first require **a lot of different images** of each of the 10 persons in the organization which might not be feasible.
  - b) What if a new person joins or leaves the organization? You need to take the pain of **collecting data again and re-train** the entire model again.



https://towardsdatascience.com/one-shot-learning-with-siamese-networks-using-keras-17f34e75bb3d

Siamese networks – motivation

- Siamese networks help solve both of the above issues. Instead of directly classifying an input(test) image to one of the 10 people in the organization, **this network instead takes an extra reference image of the person as input and produces a similarity score** denoting the chances that the two input images belong to the same person.



Notice that this network is **not learning to classify** an image directly to any of the output classes. Rather, it is learning a **similarity function**, which takes two images as input and expresses how similar they are.

https://towardsdatascience.com/one-shot-learning-with-siamese-networks-using-keras-17f34e75bb3d

Siamese networks – motivation

- How does this solve the two problems we discussed above?
  - a) To train this network, you **do not require too many instances of a class** and only few are enough to build a good model.
  - b) But the biggest advantage is that, let's say in case of face recognition, we have a new employee who has joined the organization. Now in order for the network to detect his/her face, **we only require a single image** of his/her face which will be stored in the database. Using this as the reference image, the network will calculate the similarity for any new instance presented to it. Thus, we say that network predicts the score in **one shot**.

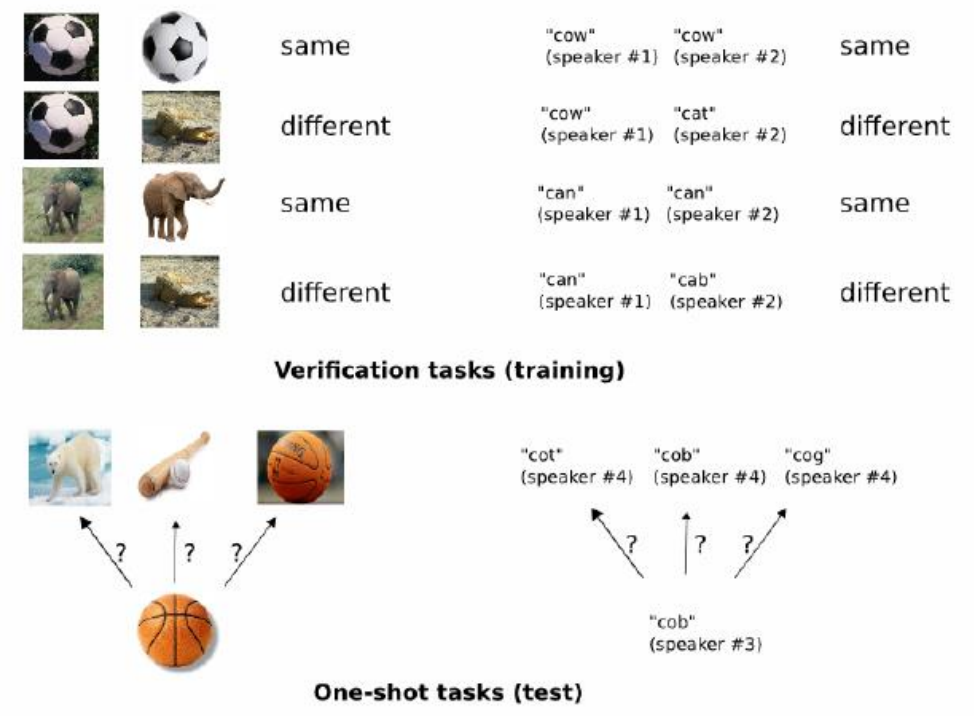
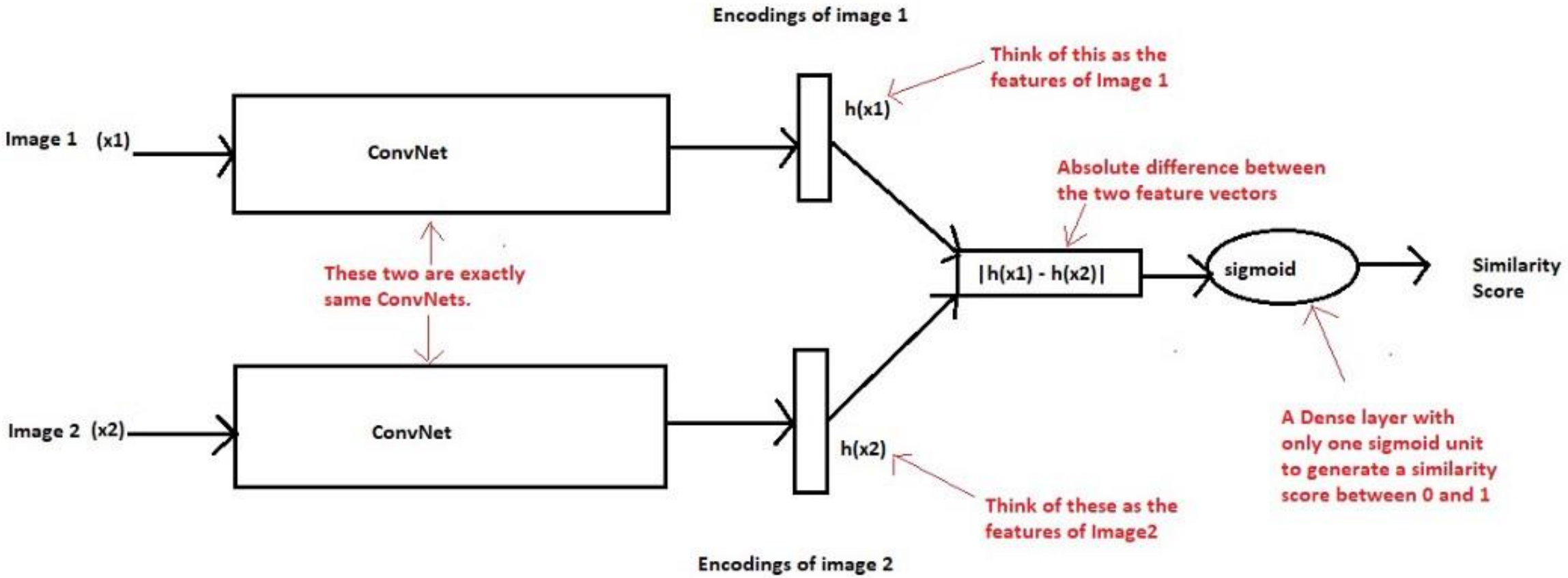


Figure 2. Our general strategy. 1) Train a model to discriminate between a collection of same/different pairs. 2) Generalize to evaluate new categories based on learned feature mappings for verification.

https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf

https://towardsdatascience.com/one-shot-learning-with-siamese-networks-using-keras-17f34e75bb3d

Siamese networks – architecture



Summary

- Many popular architectures available in model zoos
- ResNet or MobileNet are currently good defaults to use
- Networks have gotten increasingly deep over time
- Many other aspects of network architectures are also continuously being investigated and improved
- Even more recent trend towards meta-learning (learning to learn network architectures...).
- For more reviews of CNN architecture: https://towardsdatascience.com/@sh.tsang

References

- **Transfer learning:** http://cs231n.github.io/transfer-learning/
- **AlexNet:** https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
- **VGGNet:** https://arxiv.org/pdf/1409.1556.pdf
- **GoogLeNet:** https://arxiv.org/pdf/1409.4842.pdf
- **ResNet:** https://arxiv.org/pdf/1512.03385.pdf
- **MobileNet:** https://arxiv.org/pdf/1704.04861.pdf
- **U-Net:** https://arxiv.org/pdf/1505.04597.pdf
- **Siamese nets** (not the original, but this one is easier to read): https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf
- **Excellent introduction to different types of convolution:** https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215