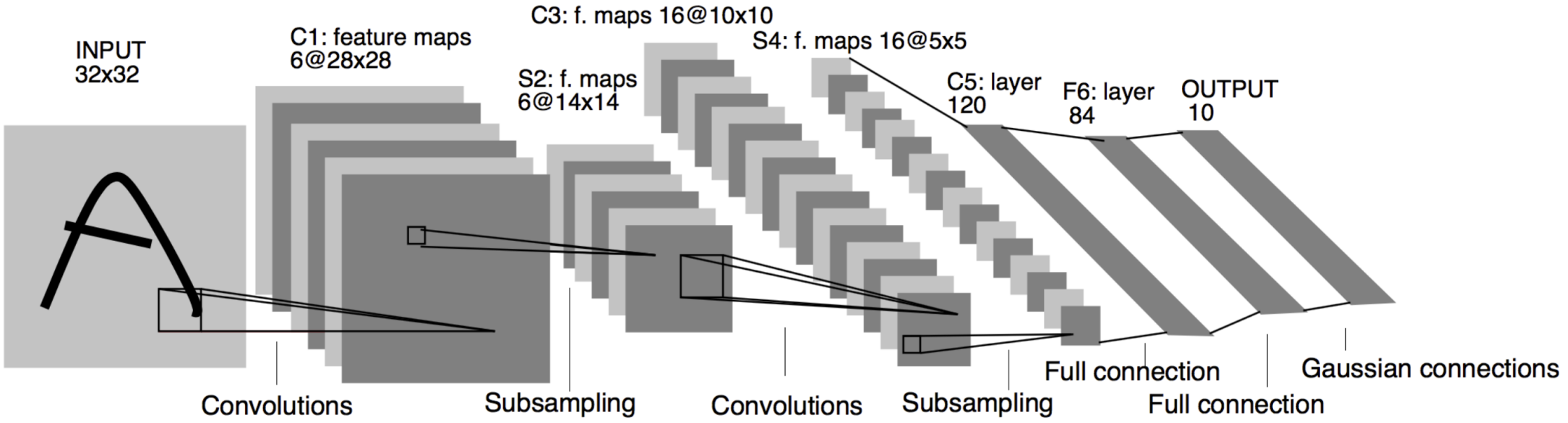
CSE 465
Lecture 7

CNN Architectures

LeNet-5 (1998)


INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections

LeNet-5

- By modern standards, LeNet-5 is a very simple network
  - It only has 7 layers
    - 3 convolutional layers (C1, C3, C5)
    - 2 sub-sampling (pooling) layers (S2 and S4)
    - 2 fully connected layer (F6) followed by the output layer
  - Convolutional layers use 5 by 5 convolutions with stride 1
  - Sub-sampling layers are 2 by 2 average pooling layers
  - Tanh activations are used throughout the network

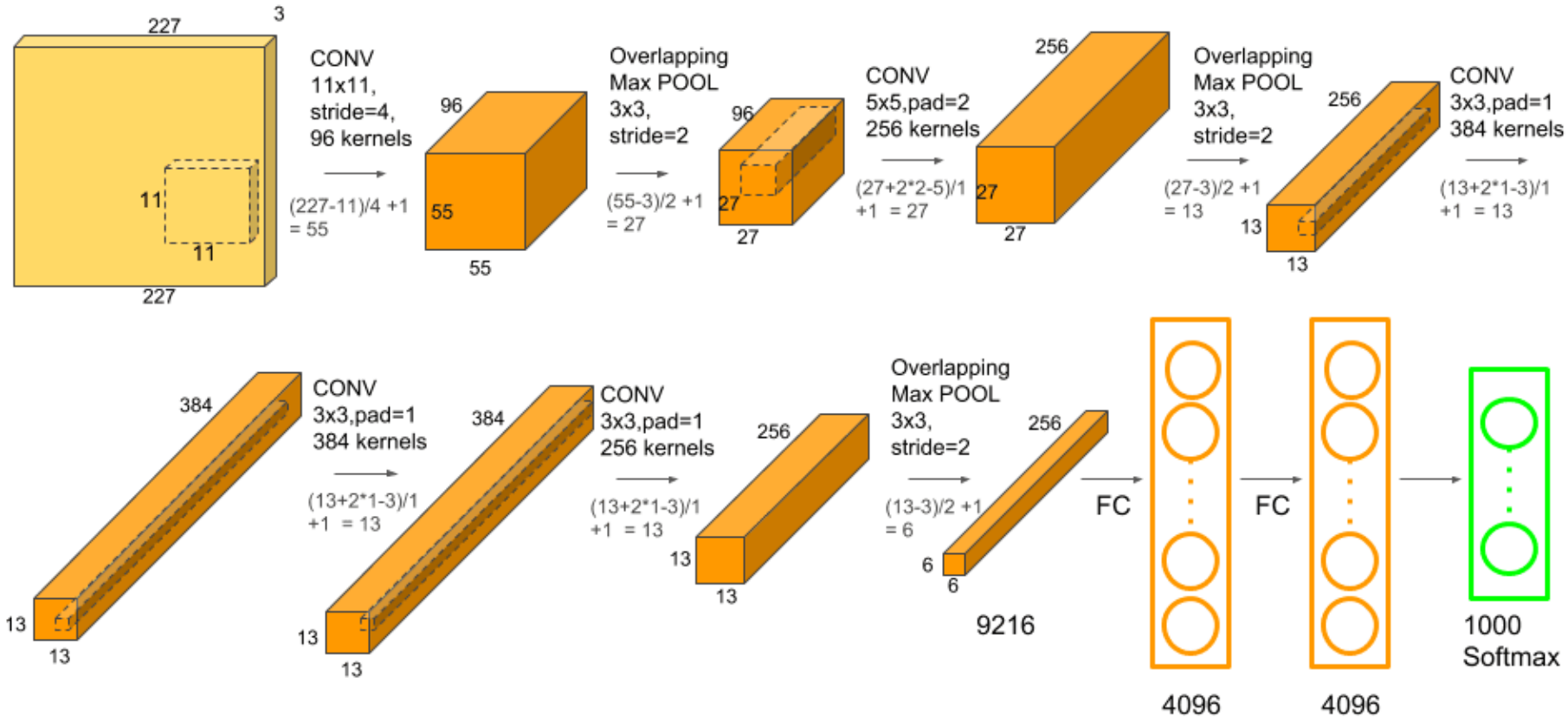
LeNet-5 Architecture Details

Layer	# filters / neurons	Filter size	Stride	Size of feature map	Activation function
Input	-	-	-	32 X 32 X 1	
Conv 1	6	5 * 5	1	28 X 28 X 6	tanh
Avg. pooling 1		2 * 2	2	14 X 14 X 6	
Conv 2	16	5 * 5	1	10 X 10 X 16	tanh
Avg. pooling 2		2 * 2	2	5 X 5 X 16	
Conv 3	120	5 * 5	1	120	tanh
Fully Connected 1	-	-	-	84	tanh
Fully Connected 2	-	-	-	10	Softmax

LeNet-5

- Individual convolutional kernels in the layer C3 do not use all the features produced by the layer S2, which is very unusual by today's standard
- Reason for that is to make the network less computationally demanding
- Another reason was to make convolutional kernels learn different patterns. This makes perfect sense: if different kernels receive different inputs, they will learn different patterns
- LeNet-5 was able to achieve error rate below 1% on the MNIST data set, which was very close to the state of the art at the time
- Around 60K parameters

AlexNet



- It contains 5 convolutional layers and 3 fully connected layers
- The network has 62.3 million parameters, and needs 1.1 billion computation units in a forward pass

AlexNet Parameter Count

```
Model: "alex_net"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              multiple                  34944
_________________________________________________________________
conv2d_1 (Conv2D)            multiple                  614656
_________________________________________________________________
conv2d_2 (Conv2D)            multiple                  885120
_________________________________________________________________
conv2d_3 (Conv2D)            multiple                  1327488
_________________________________________________________________
conv2d_4 (Conv2D)            multiple                  884992
_________________________________________________________________
max_pooling2d (MaxPooling2D) multiple                  0
_________________________________________________________________
flatten (Flatten)            multiple                  0
_________________________________________________________________
dense (Dense)                multiple                  37752832
_________________________________________________________________
dense_1 (Dense)              multiple                  16781312
_________________________________________________________________
dense_2 (Dense)              multiple                  4097000
=================================================================
Total params: 62,378,344
Trainable params: 62,378,344
Non-trainable params: 0
_________________________________________________________________
```

- In AlexNet convolution layers accounts for 6% of all the parameters consumes 95% of the computation. The main features of AlexNet are
  - Copy convolution layers into different GPUs; Distribute the fully connected layers into different GPUs.
  - Feed one batch of training data into convolution layers for every GPU (Data Parallel).
  - Feed the results of convolution layers into the distributed fully connected layers batch by batch (Model Parallel) When the last step is done for every GPU. Backpropogate gradients batch by batch and synchronize the weights of the convolution layers.
- Obviously, it takes advantage of the features: convolution layers have a few parameters and lots of computation, fully connected layers are just the opposite.

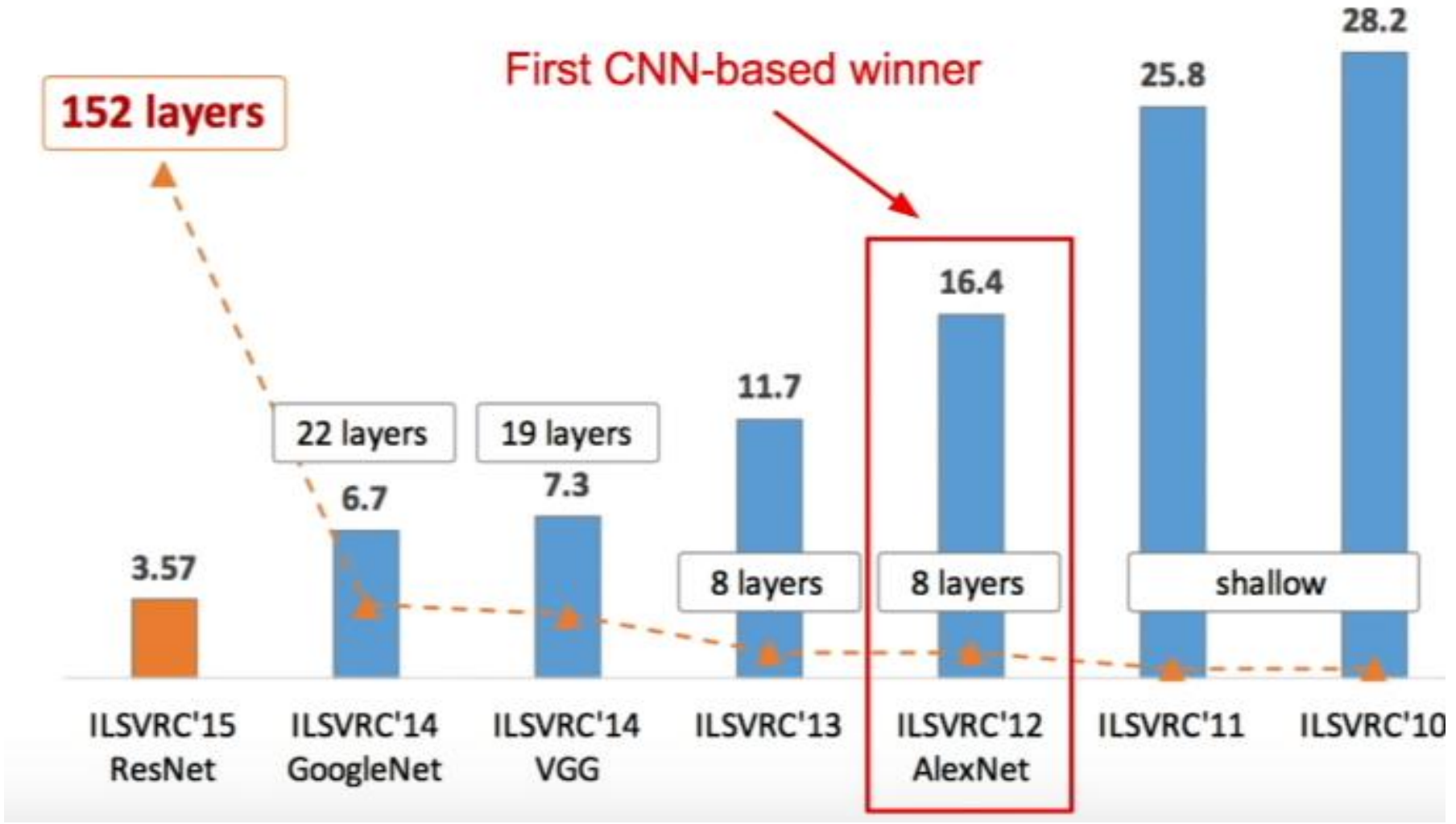
AlexNet Interesting Features

- They first used ReLU
- Used normalization layers
- Heavy data augmentation
- Dropout ratio 0.5
- Batch size 128
- SGD momentum 0.9
- Learning rate 1e-2 – reduced manually by 10

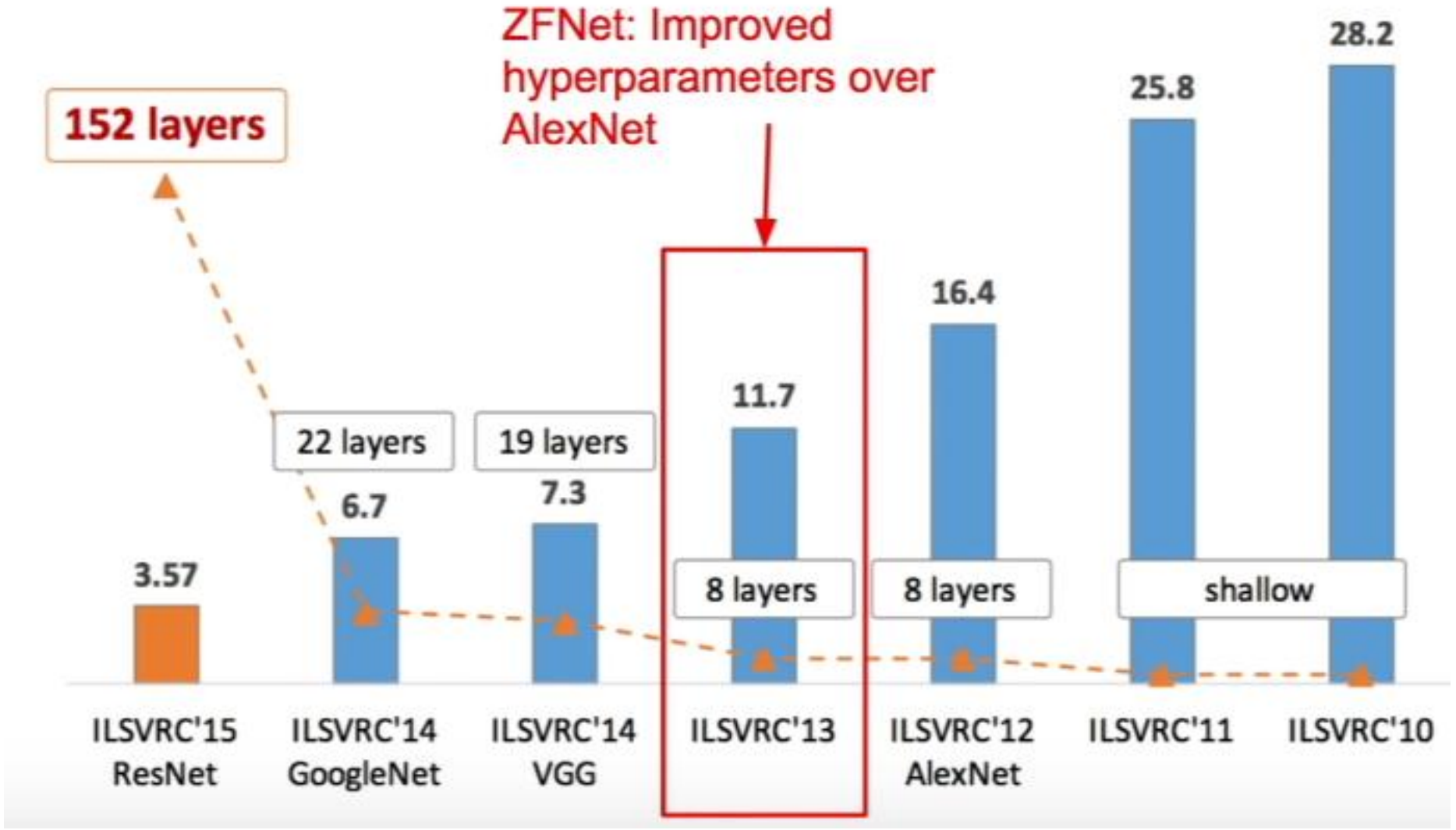
AlexNet Conclusion

- AlexNet famously won the 2012 ImageNet ILSVRC-2012 competition by a large margin (15.3% VS 26.2% (second place) error rates)
- The network takes 90 epochs in five or six days to train on two GTX 580 GPUs
- SGD with learning rate 0.01, momentum 0.9 and weight decay 0.0005 was used
- Learning rate was divided by 10 once the accuracy plateaus
- Bigger network than the LeNet-5
- Unlike LeNet-5 they used ReLU
- Basically used GPU to improve the performance

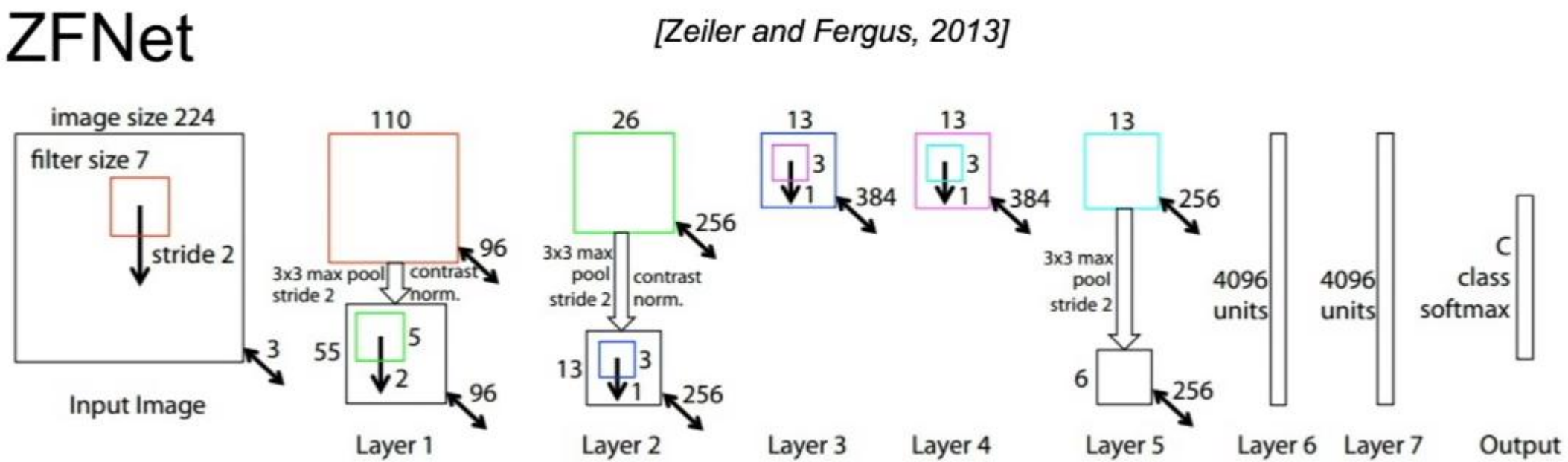
ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



ZFNet

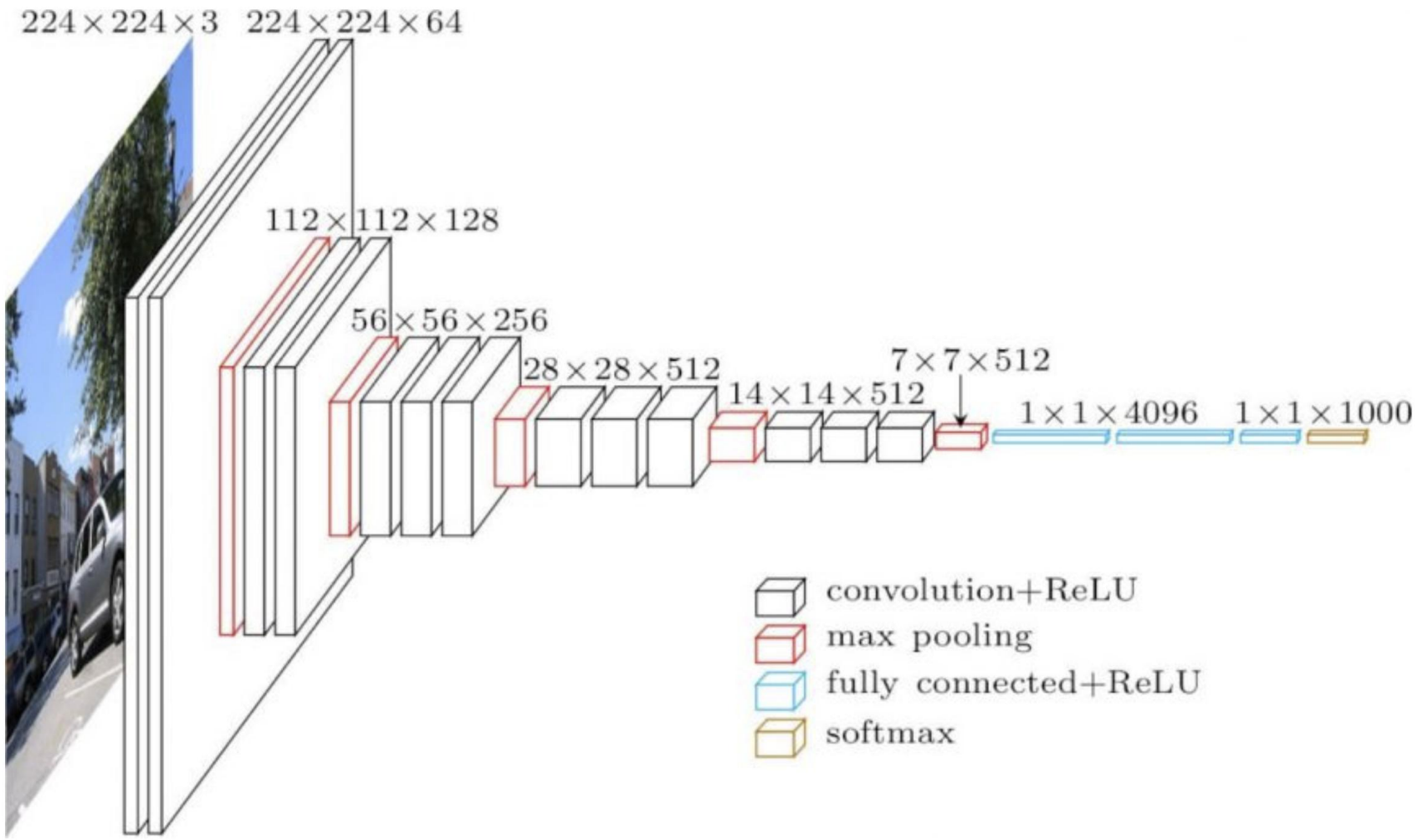


- Almost AlexNet – changing hyperparameters
  - Conv1: Changes from 11X11 stride 4 to 7X7 stride 2
  - Conv3,4,5: Instead of 384, 384, 256 filters use 512, 1024, 512

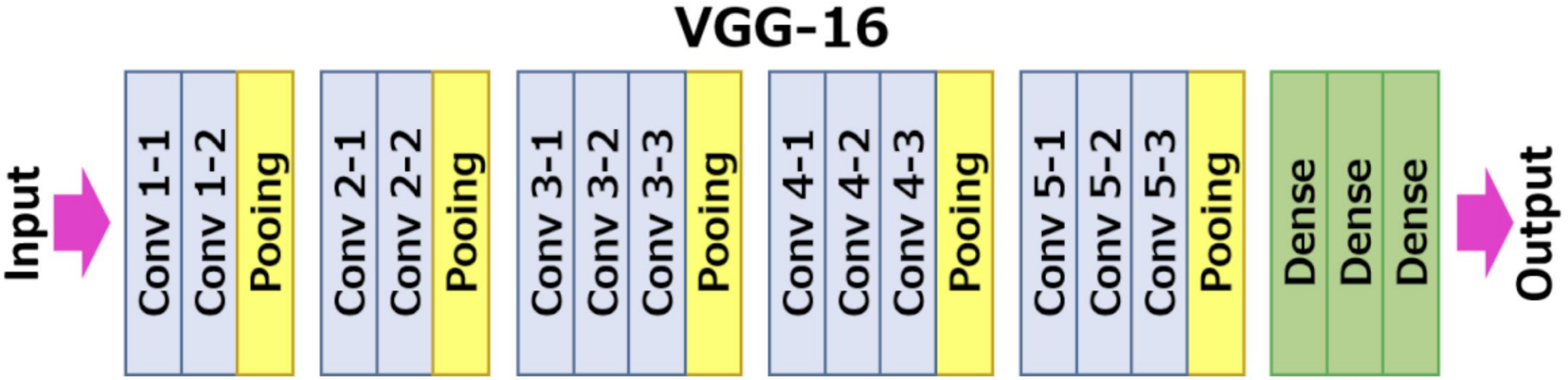
VGG-XX

- According to the universal approximation theorem, given enough capacity, we know that a feedforward network with a single layer is sufficient to represent any function
- However, the layer might be massive, and the network is prone to over-fitting the data
- Therefore, there is a common trend in the research community to use small blocks and make the networks deep
- Since AlexNet, the state-of-the-art CNN architecture is going deeper and deeper
- While AlexNet had only 5 convolution layers, the VGG network has more than 16 layers

VGG-16



VGG16 Architecture



Improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU's.

VGG-16

- The input to cov1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center).
- The convolution stride is fixed to 1 pixel
- Padding was used to keep the image size fixed i.e. the padding is 1-pixel for 3×3 convolution layers
- Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

VGG-16

- Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures)
  - The first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class)
  - The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.
- All hidden layers are equipped with the rectification (ReLU) non-linearity
- It is also noted that none of the networks (except for one) contain Local Response normalization (LRN), such normalization does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time

VGG16 Parameter Count

VGG16 - Structural Details													
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													138,423,208

VGG-16

- Unfortunately, there are two major drawbacks with VGGNet:
  - It is *painfully slow* to train
  - The network architecture weights themselves are quite large (concerning disk/bandwidth)
- Due to its depth and number of fully-connected nodes, VGG16 is over 533MB
- There are 138 millions of parameters to learn

VGG improvements over AlexNet

- Smaller filters (sometimes called kernels) reduce computations
- More parameters
  - May learn better if we can take care of the overfitting problems
- Somewhat modular
  - Need improvements? Use more blocks
- However, increasing network depth does not work by simply stacking layers together
- Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small
- As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly

Observations until now

- Choosing the right kernel size for convolution is always a big deal in CNNs
  - The same object can come in various sizes in different images
- To capture the features of different sizes, we, of course, need to have kernel sizes accordingly
  - Bigger kernels are generally good when the object covers most of the area
  - Smaller kernels are suitable for relatively small objects
- The deeper the network, the better it is! But stacking a lot of layers makes the gradient flow difficult and leads to overfitting
- In short, the depth of a network is somewhat restricted to a certain limit – beyond this limit, the network is not trainable anymore; it just overfits

Observation until now

- Some parts of the image can have extremely large variation in size
  - For instance, an image of a dog can be big, small, etc.
  - Even the same image can have dogs of different sizes
- Because of this variation in the location/proportion of the information, choosing the **right kernel size** for the convolution operation becomes tough
- A **larger kernel** is preferred for information that is covering a large part of the image
- A **smaller kernel** is preferred for information that is covering smaller portion of an image
- And Inception is born

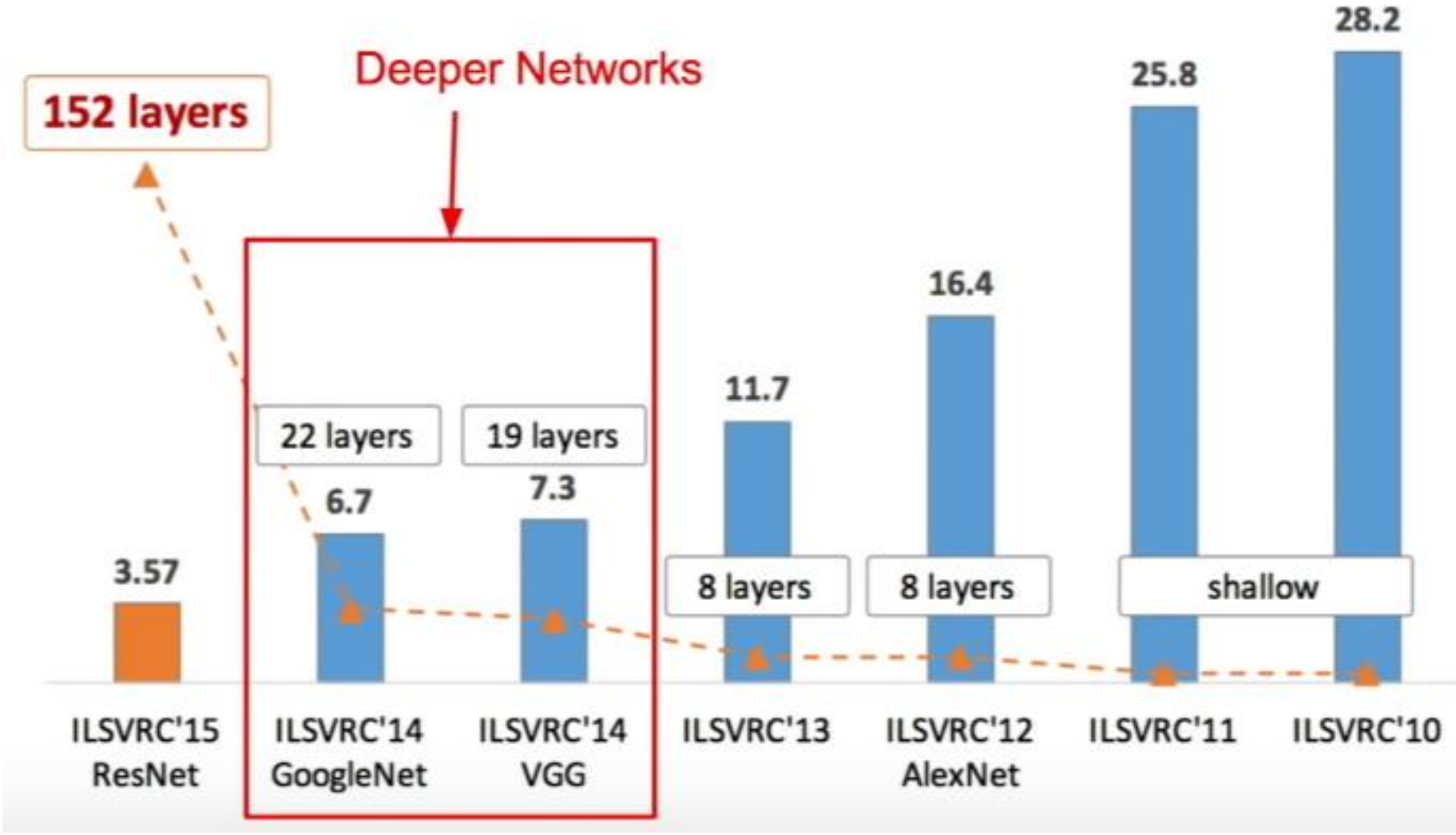
Observation until now

- **Very deep networks** are prone to **overfitting**
- It also hard to pass gradient updates through the entire network
- Naively stacking large convolution operations is **computationally expensive**
- And here comes Resnet

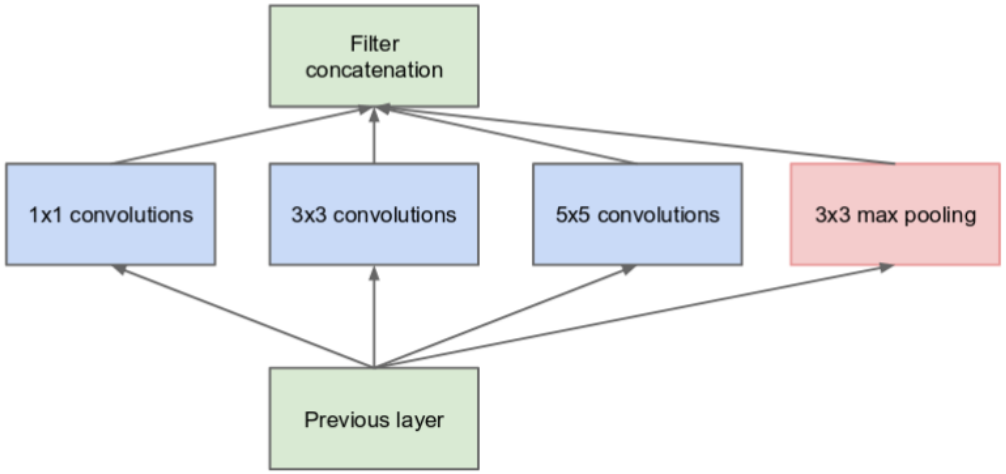
Inception networks

- The idea is to use different sizes for convolution kernels in parallel rather than using a single kernel size in a layer
- This way, the network now has option to choose kernels having different sizes and the network can now learn features through the kernel that's best suited to the job
- Arranging kernels parallel to each other also makes the architecture sparse, which helps to ease the training for deeper networks
- The typical InceptionNet uses three convolution kernels of sizes 1 x 1, 3 x 3, and 5 x 5 simultaneously
- The results from all these three kernels are concatenated to form a single output vector, which acts as the input for the next layer
- The inception layer also adds 1 x 1 convolutions before the 3 x 3 and 5 x 5 kernels in order to reduce the size

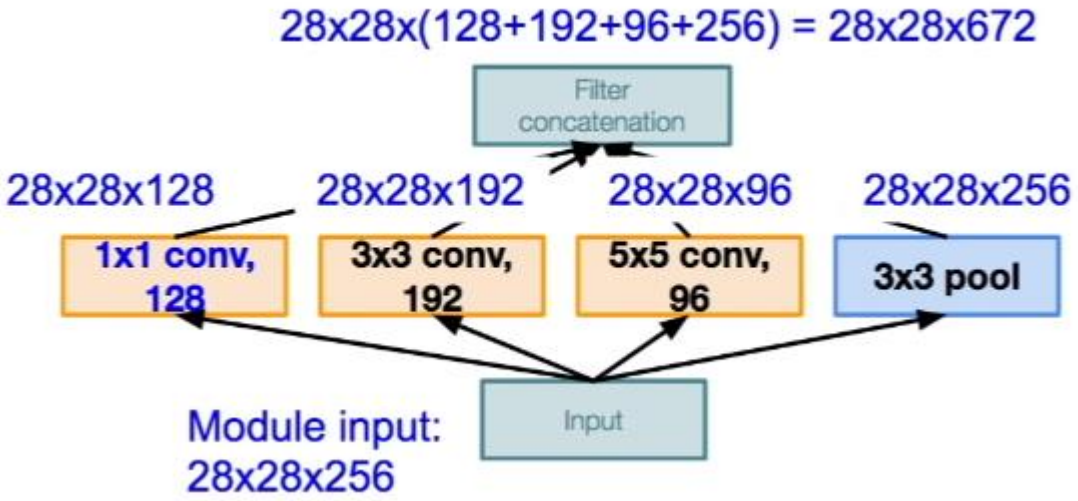
ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



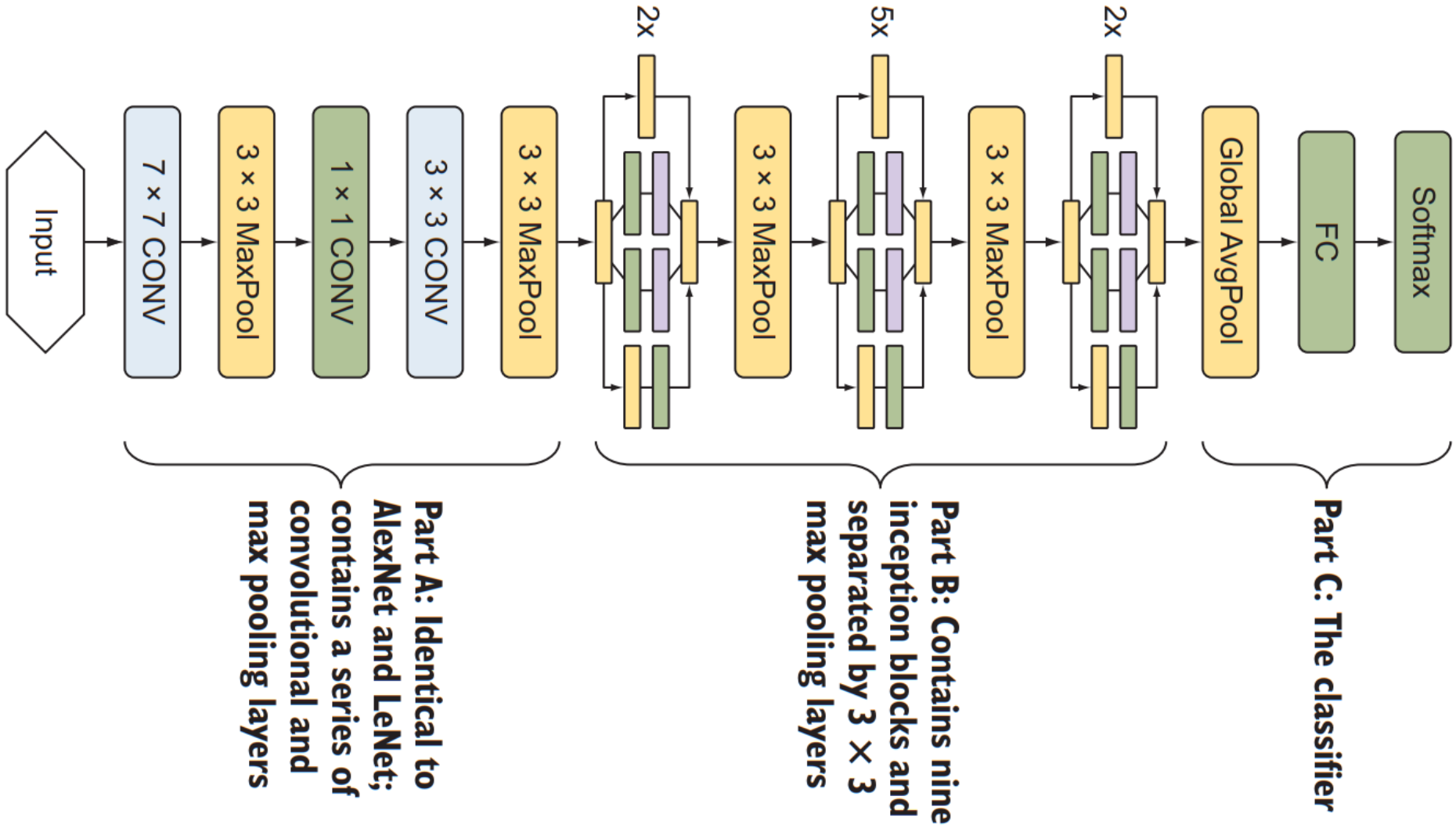
Inception Network (GoogleNet)



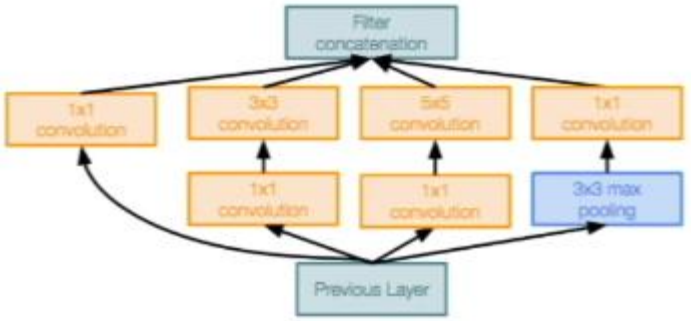
(a) Inception module, naïve version



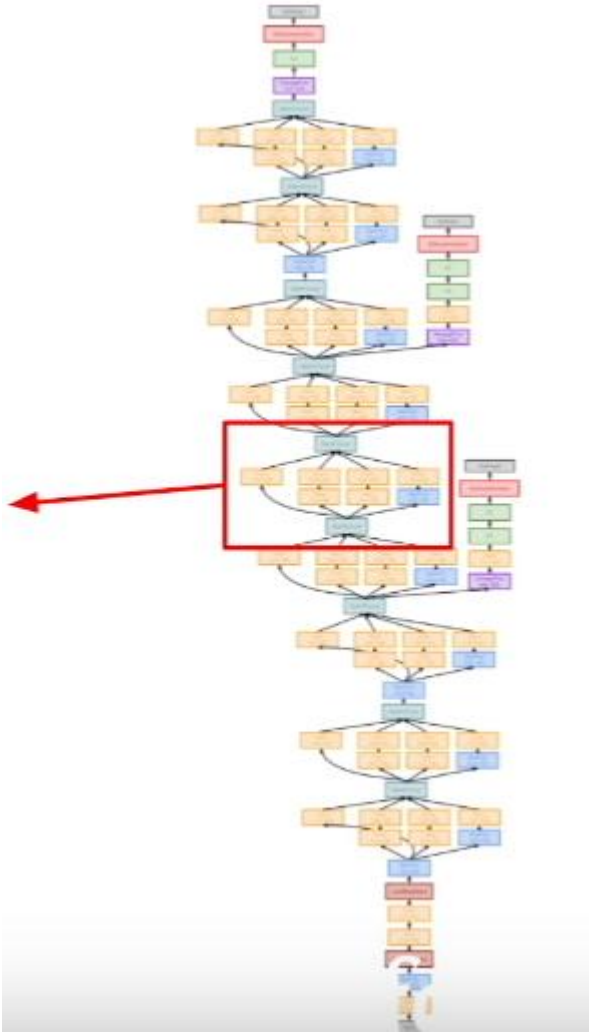
GoogleNet: The complete architecture



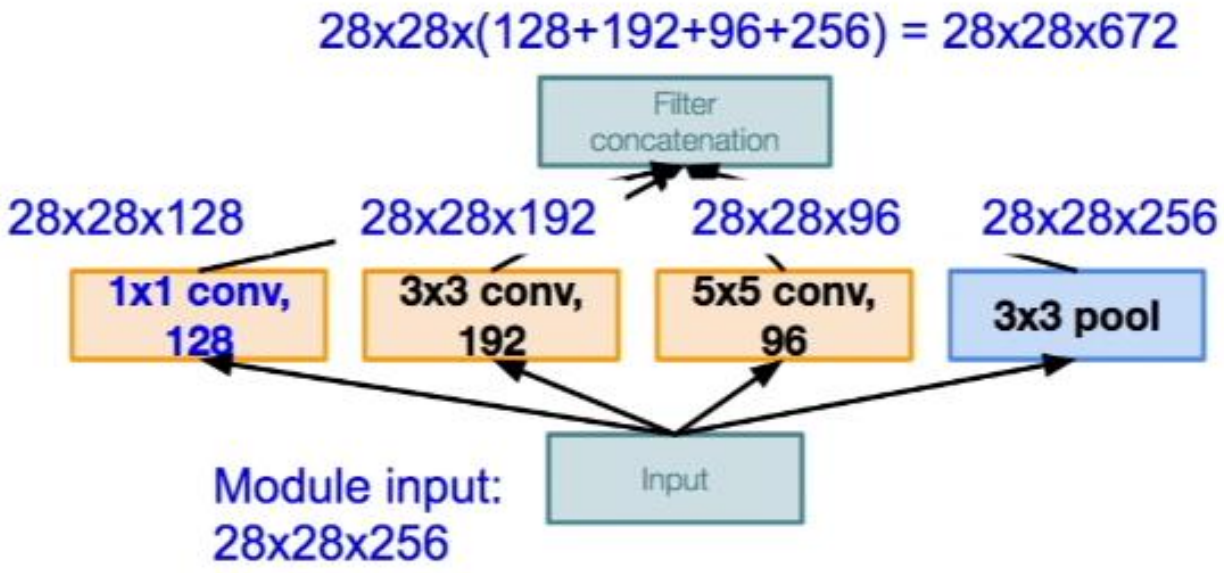
GoogleNet (collection of inception modules)



Inception module



However… The computational complexity



Naive Inception module

Conv Ops:

[1x1 conv, 128] 28x28x128x1x1x256

[3x3 conv, 192] 28x28x192x3x3x256

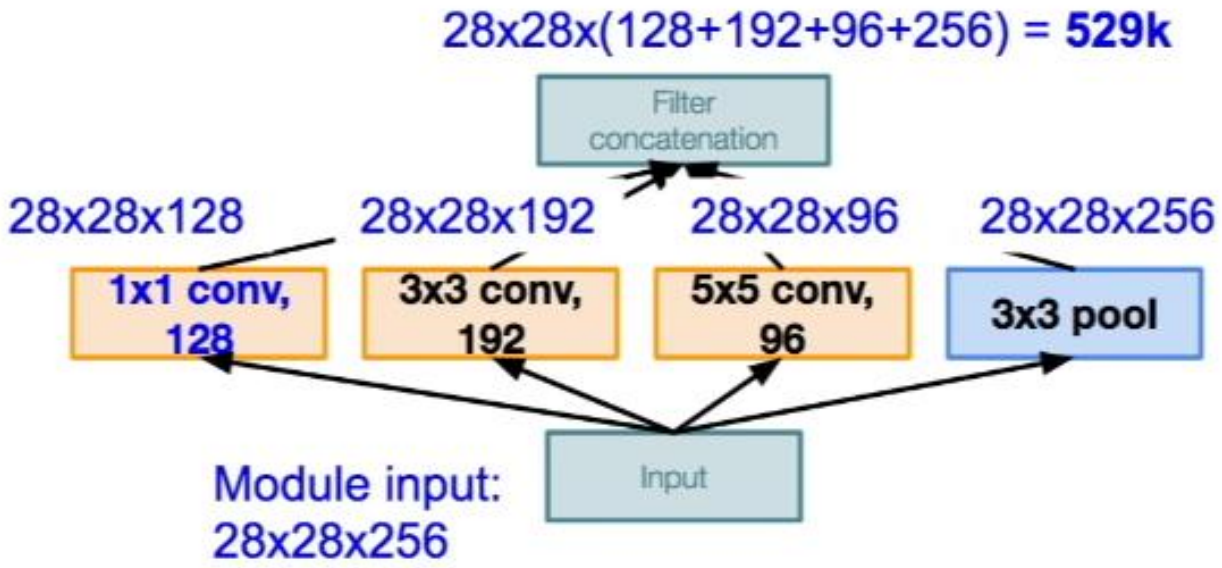
[5x5 conv, 96] 28x28x96x5x5x256

Total: 854M ops

Very expensive compute

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

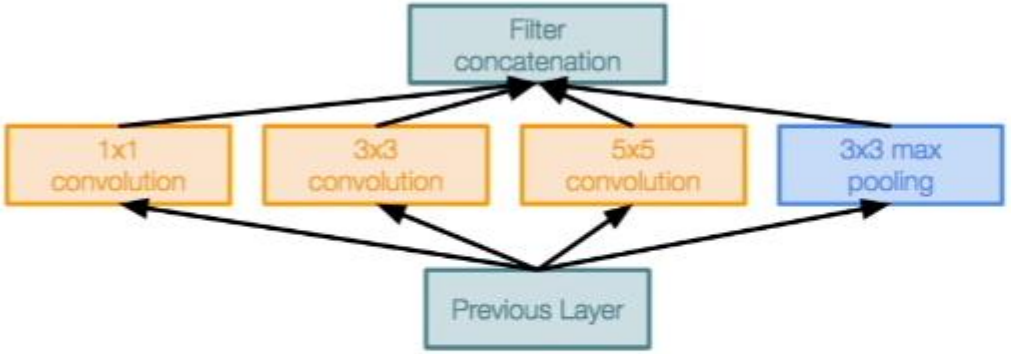
What do we do?



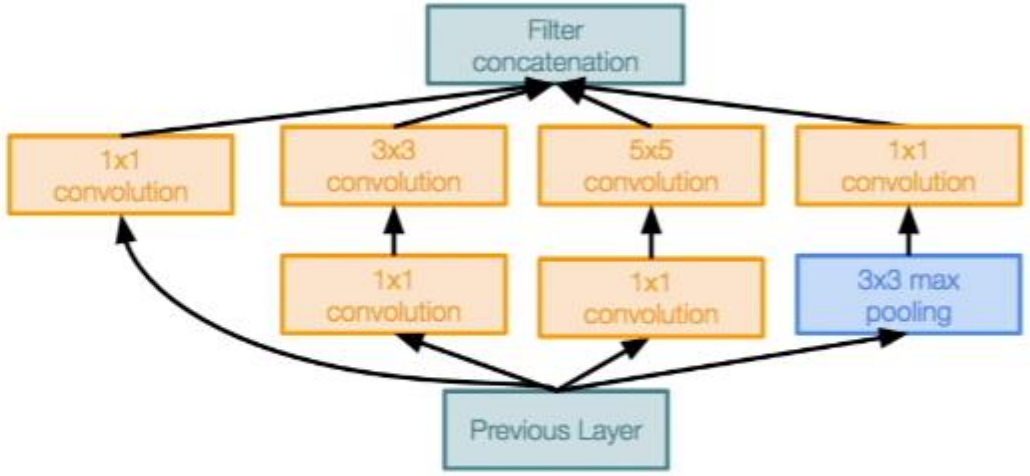
Naive Inception module

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

Computationally efficient inception modules

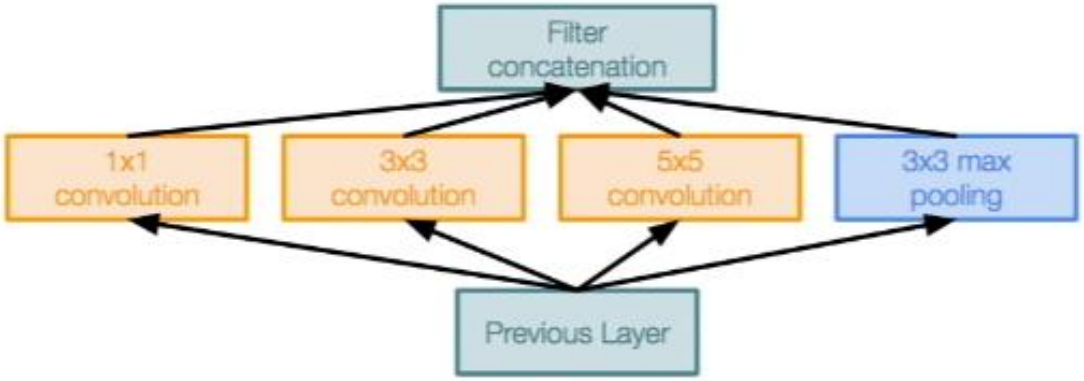


Naive Inception module



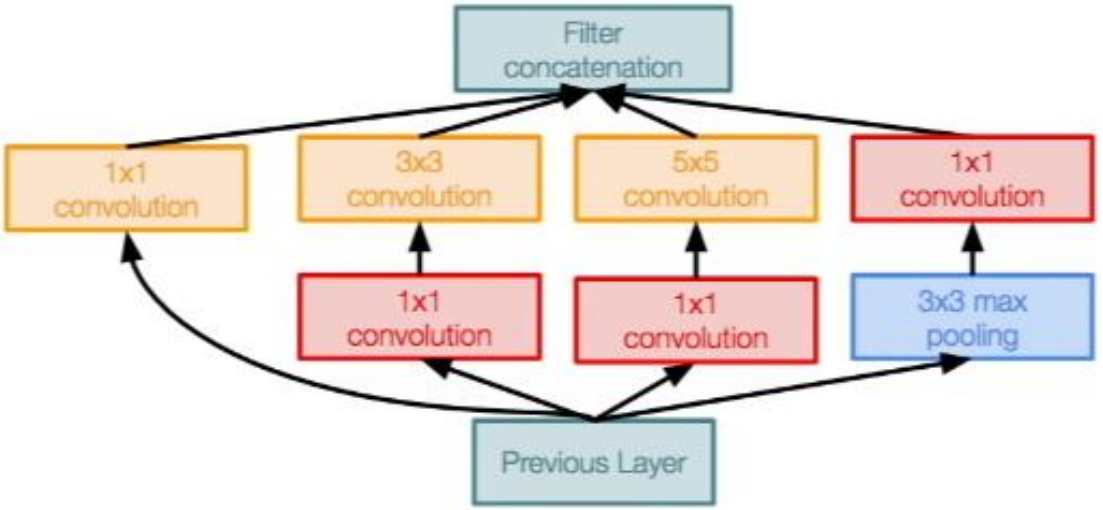
Inception module with dimension reduction

Computationally efficient inception modules



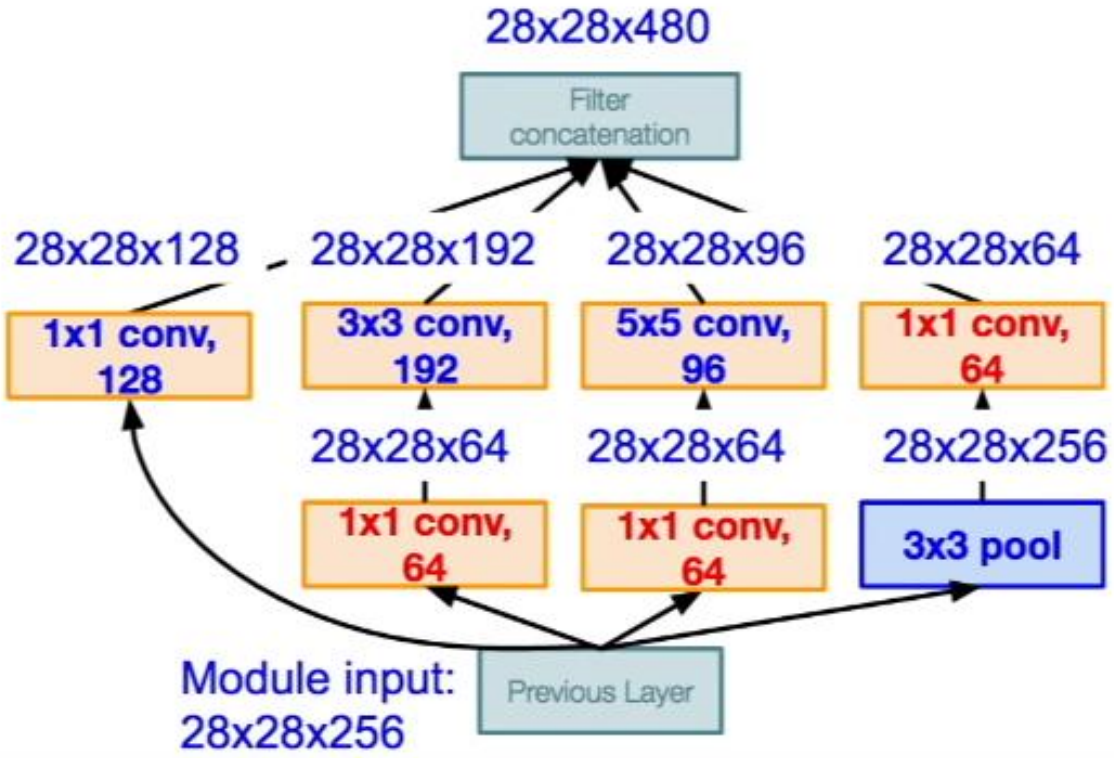
Naive Inception module

1x1 conv "bottleneck" layers



Inception module with dimension reduction

Computationally efficient inception modules
The improvements



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

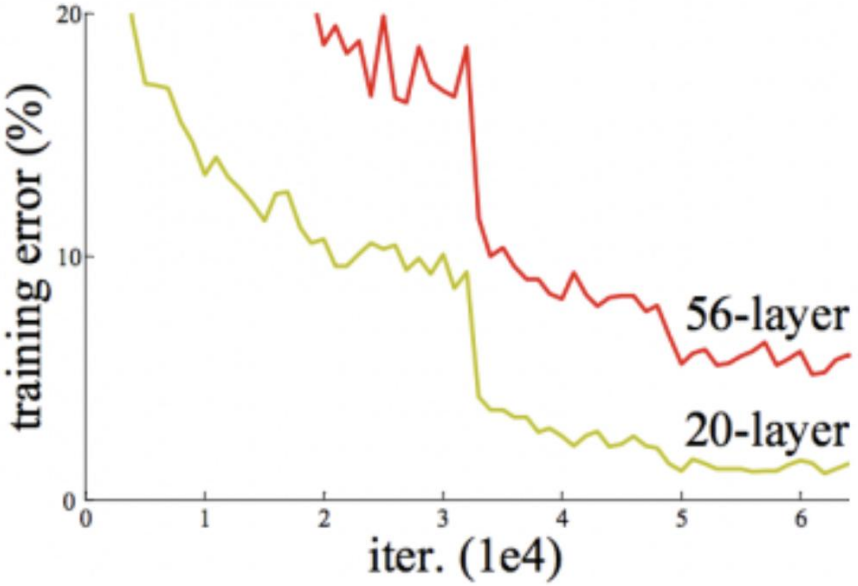
Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

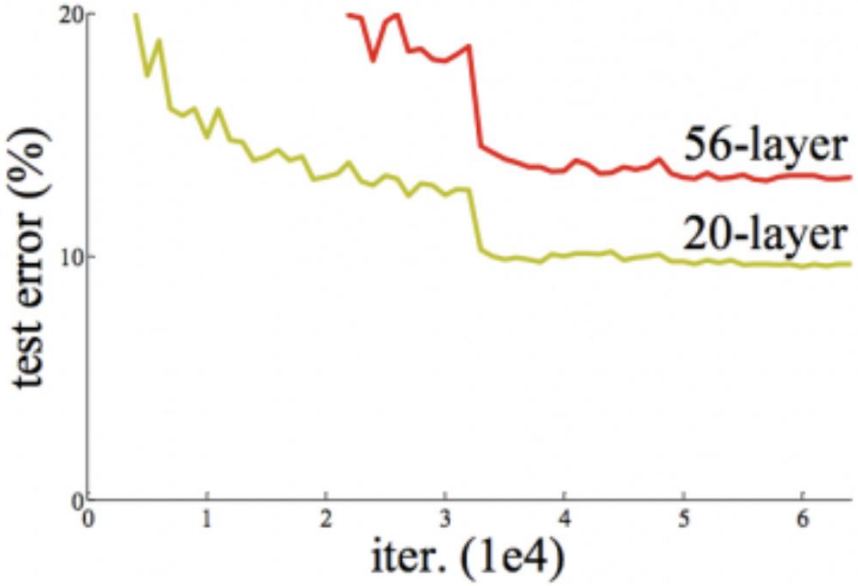
ResNet

- ResNet exposes a problem that has been restricting the training of very deep networks
- During the training of deep networks, the accuracy saturates to a certain limit and thereafter degrades rapidly
- This phenomenon has been restricting the accuracy to a certain threshold, no matter how deep the architecture goes
- ResNet was introduced by Microsoft research in a paper called, *Deep Residual Learning for Image Recognition*
- This is the first network architecture that has successfully trained a network with a depth of more than 100 layers
- What made this impossible possible?
  - Better weight initializations
  - New normalization layers
  - The skip-connections

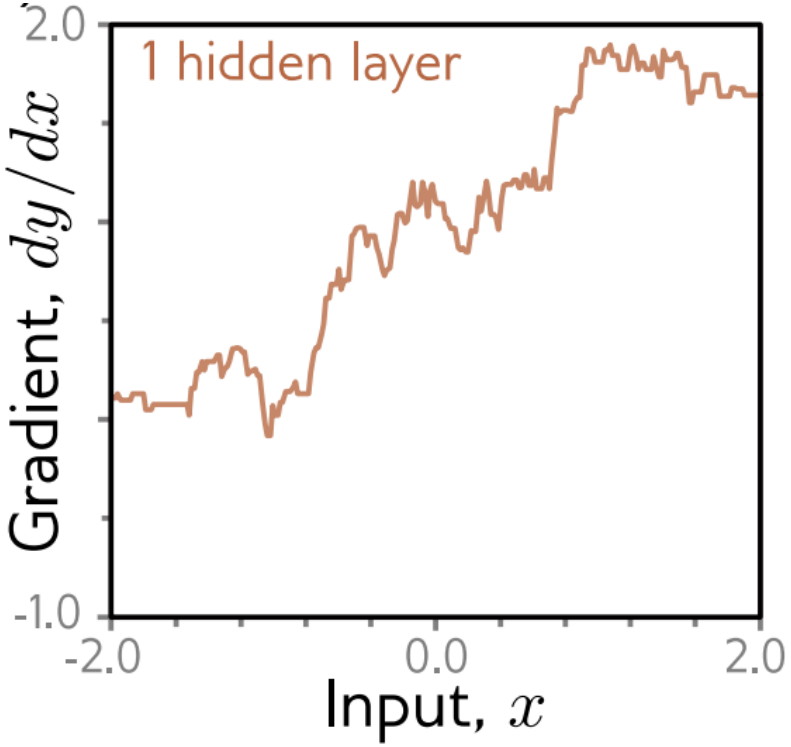
Why ResNet? 56/20 Layers

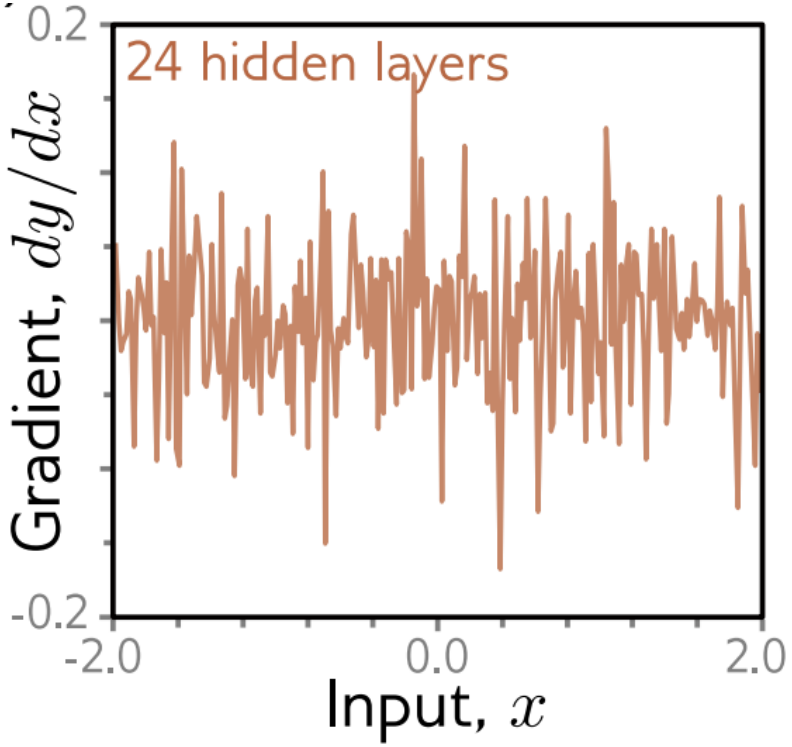
- A group at Microsoft research were going deeper and deeper and found the following graph
- Training & test error are lower (better) for a 20 layer network than a 56 layer network!!!
- Overfitting is the problem
- There was no good way to optimize the weights

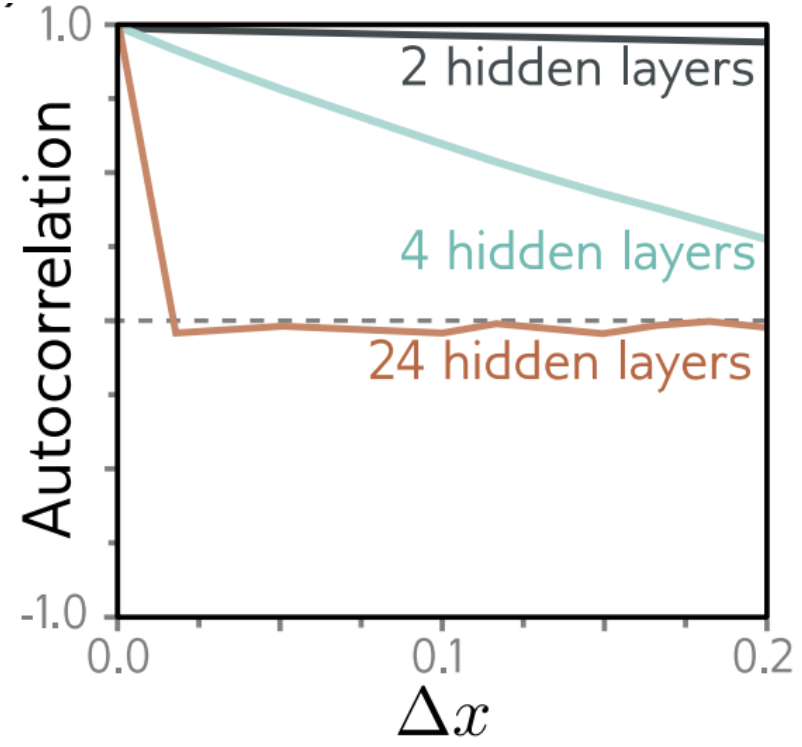




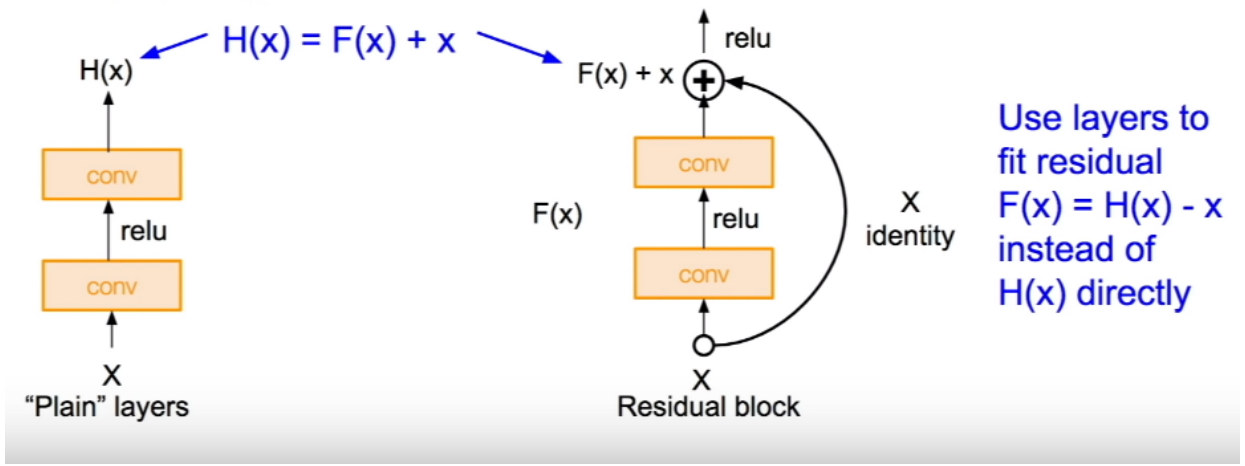
Finding the cause: shattered gradient





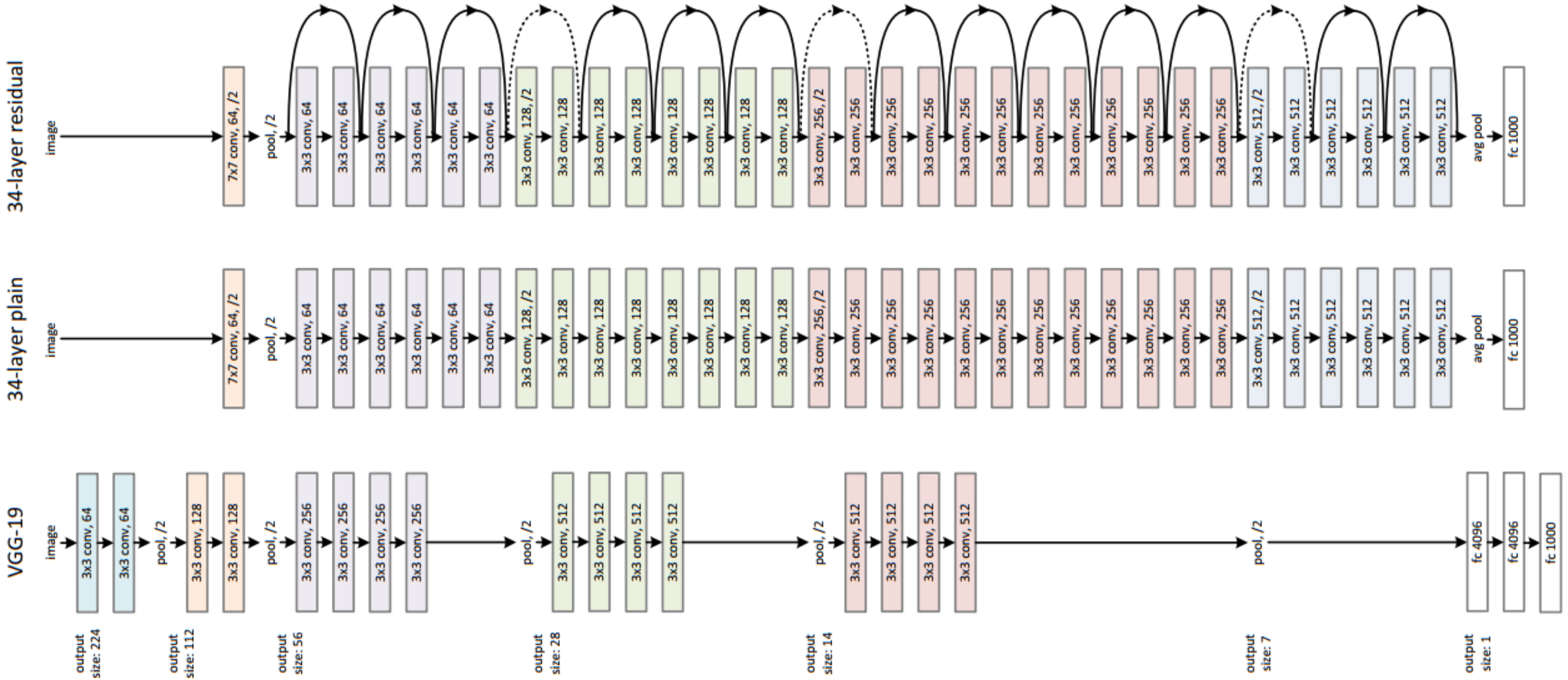


ResNet Concepts



- Each layer computes an additive change to the current representation instead of transforming it directly
- This allows deeper networks to be trained but causes an exponential increase in the activation magnitudes at initialization
- Residual blocks use batch normalization to compensate for this, which re-centers and rescales the activations at each layer
- This kind of creates a connection from the input to the output of the layer, which is called the **residual connection** or **skip connection**

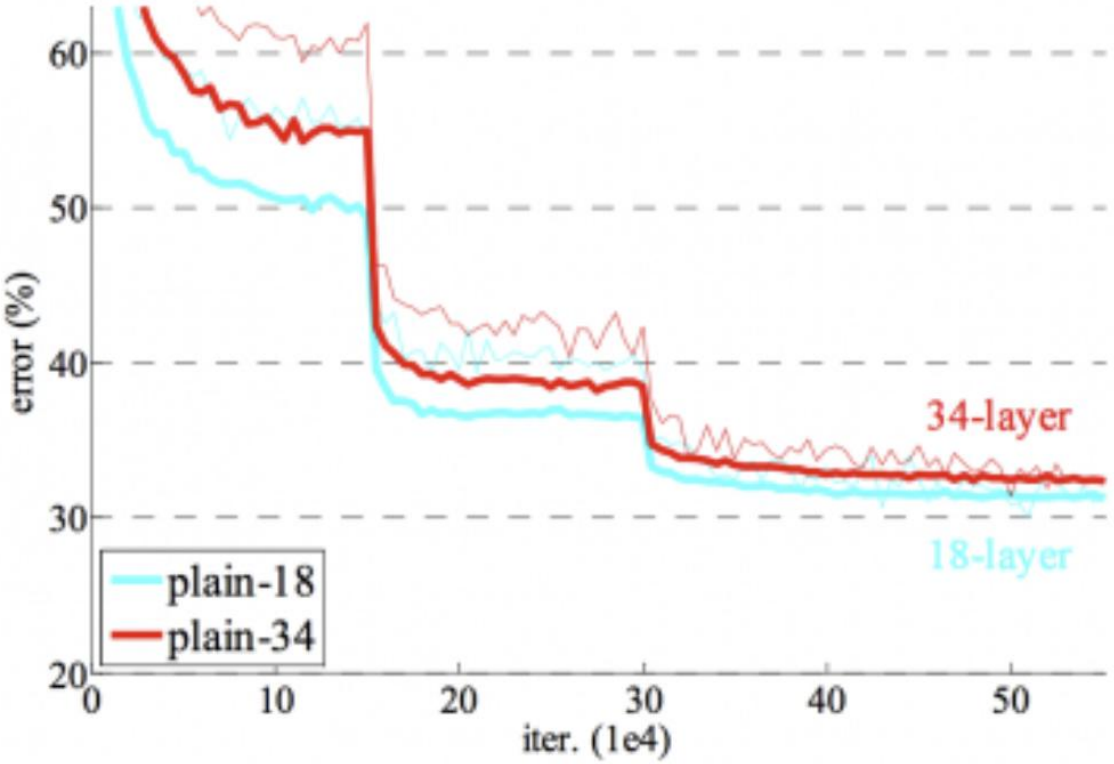
ResNet Actual Implementation

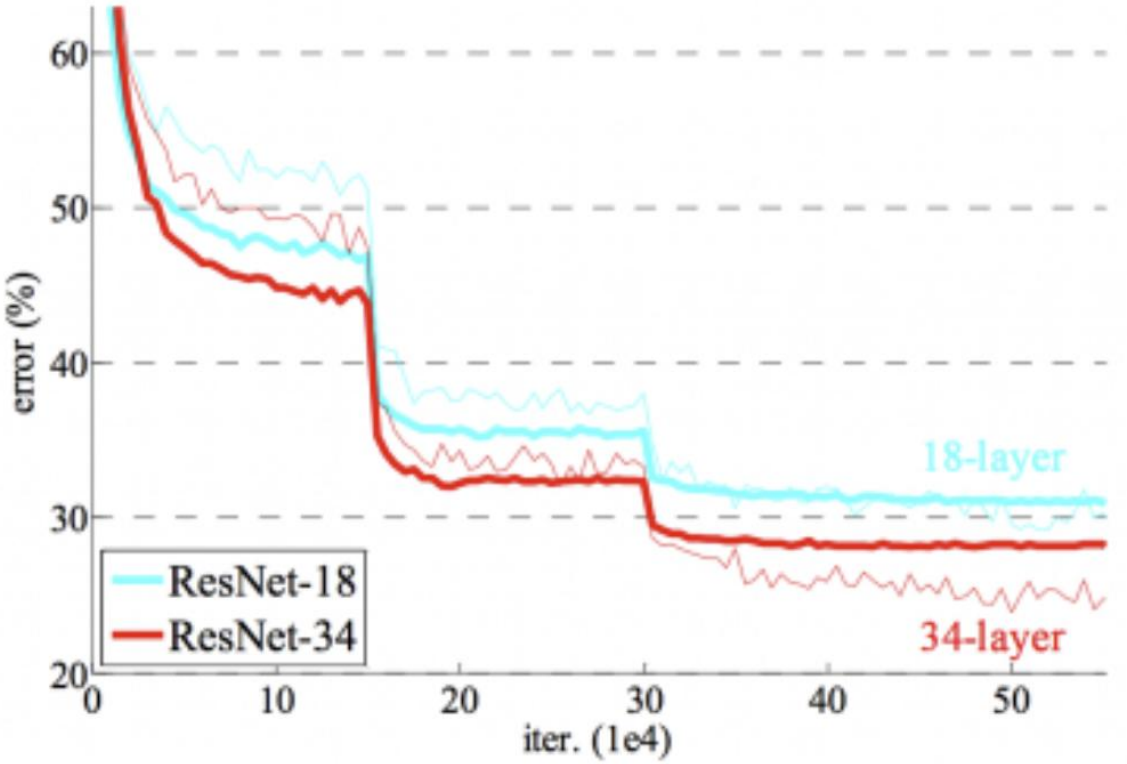


Why ResNet?

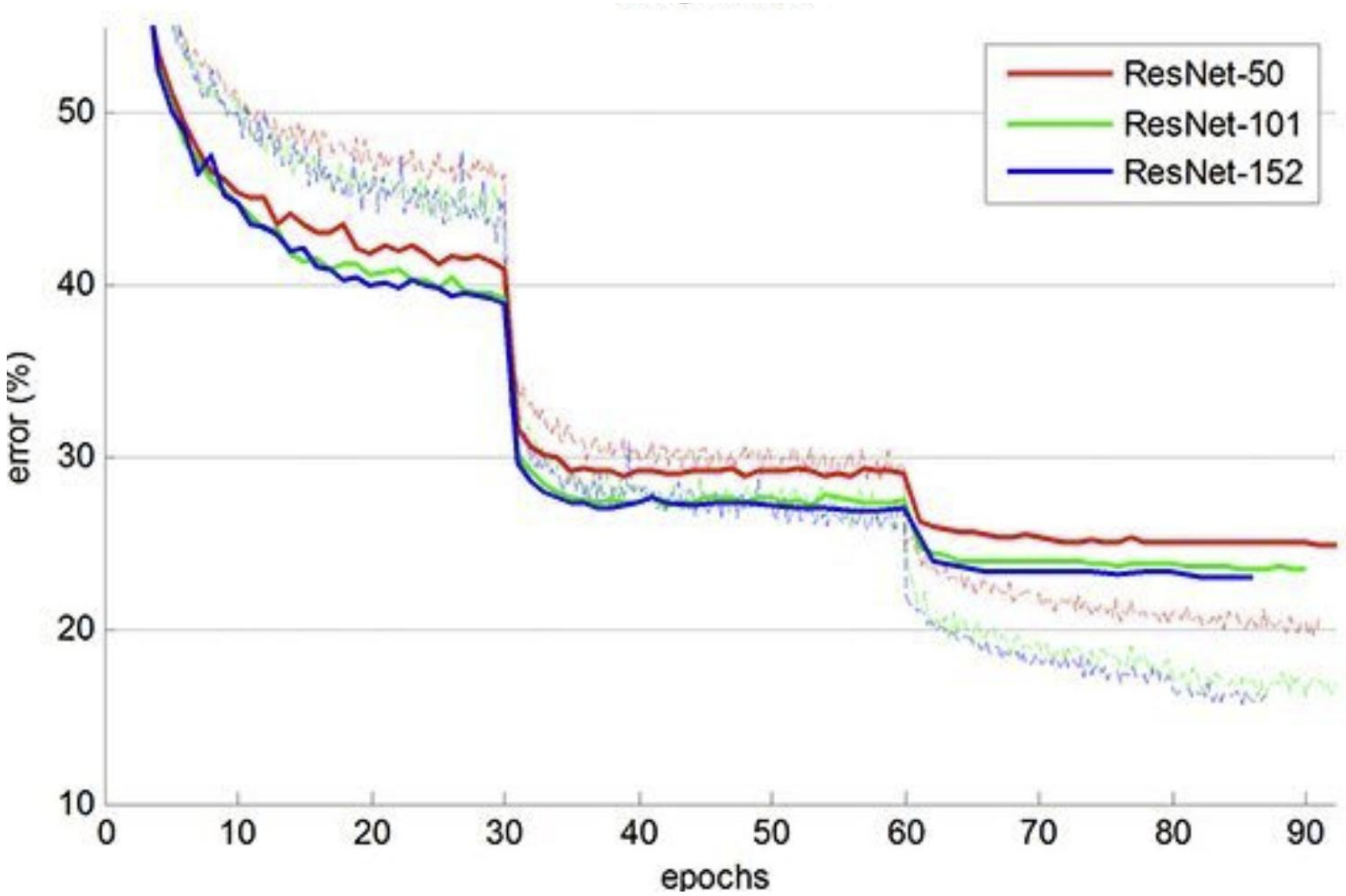
- When the deeper network starts to converge, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly
- Such degradation is not caused by overfitting not by more layers
  - But because of the failure of the traditional optimization methods
- Therefore, we add workarounds to propagate the information needed to optimize better
- So, does ResNet solve the problem?
  - The next slide shows the improvements

ResNet


error (%)
60
50
40
30
20
0
10
20
30
40
50
iter. (1e4)
plain-18
plain-34
34-layer
18-layer

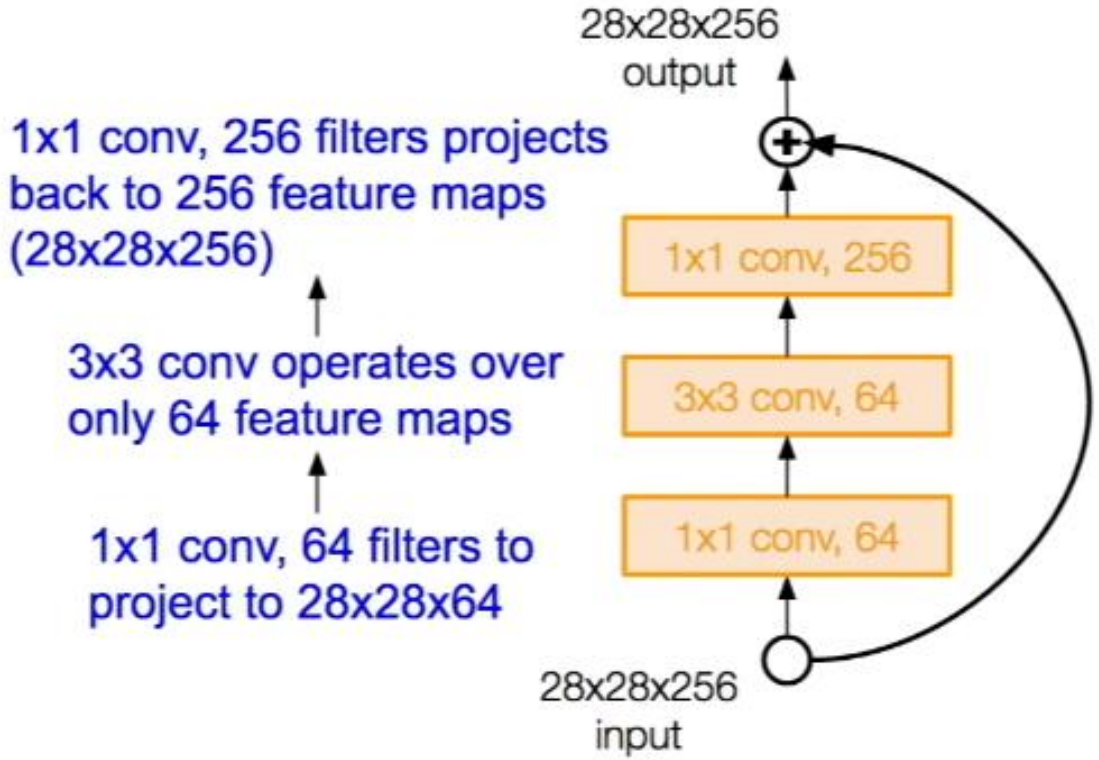

error (%)
60
50
40
30
20
0
10
20
30
40
50
iter. (1e4)
ResNet-18
ResNet-34
18-layer
34-layer

ResNet


ResNet-50
ResNet-101
ResNet-152
error (%)
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90
epochs

More optimizations

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



ResNet in practice

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