Convolutional Architectures For Image Classification II

Advanced Computer Vision

Niels Landwehr

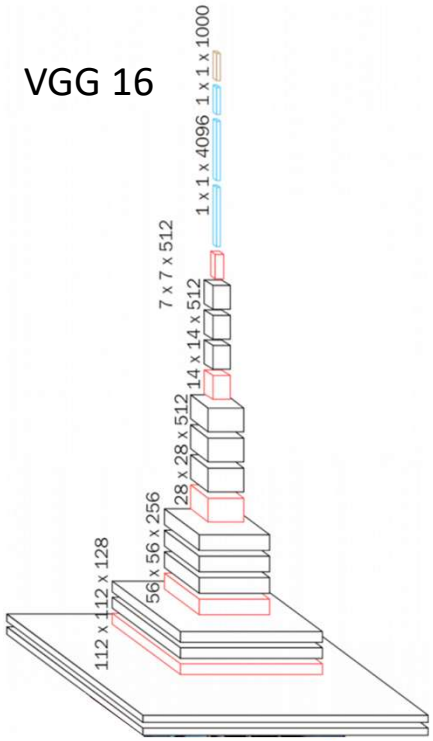
Overview

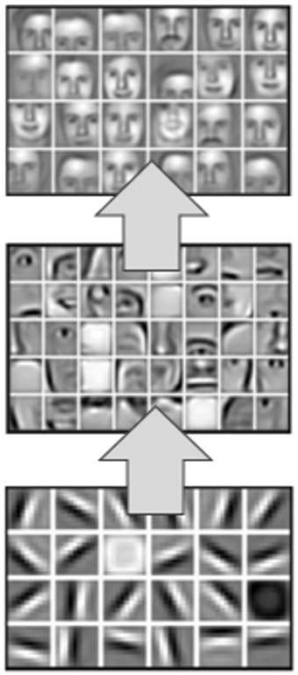
- Introduction: Computer Vision
- Data, Models, Optimization
- Neural Networks and Automatic Differentiation
- Convolutional Architectures For Image Classification I
- Convolutional Architectures For Image Classification II

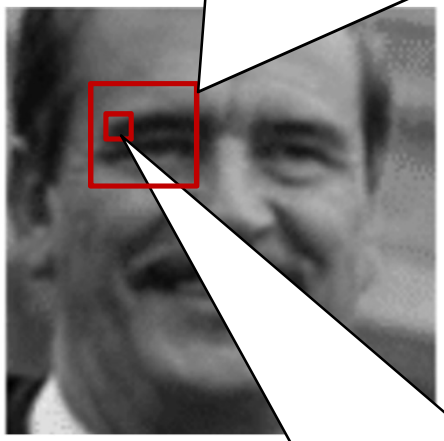
Recap: Convolutional Neural Networks

- Recap: convolutional neural networks to learn hierarchies of increasingly abstract visual features

Example: Face recognition with deep convolutional neural network







High-level layers should extract more complex and abstract features. Spatially, these need to cover larger parts of the image, and need to be built up from the simpler features.

Low level layers extract simple features such as edges with convolution filters from input

Advanced Convnet Architectures

- The relatively simple architecture concept of networks such as VGG has been refined and advanced in many ways
- The basic principle is still to stack multiple convolution layers with pooling or strided convolutions in between for reducing spatial dimensions, but certain extensions allow to construct more powerful and efficient networks

Increased Network Depth	**Power/efficiency Tradeoff**	**Speed and Robustness of Optimization**
How to train deep (> 20 layers) convnet architectures?	How to optimize computational efficiency of architectures?	How to improve convergence speed and robustness of optimization?
Residual connections	*Inception, depthwise separable convolutions, ...*	*Batch normalization layers*

Advanced Convnet Architectures

- The relatively simple architecture concept of networks such as VGG has been refined and advanced in many ways
- The basic principle is still to stack multiple convolution layers with pooling or strided convolutions in between for reducing spatial dimensions, but certain extensions allow to construct more powerful and efficient networks

Increased Network Depth	**Power/efficiency Tradeoff**	**Speed and Robustness of Optimization**
How to train deep (> 20 layers) convnet architectures?	How to optimize computational efficiency of architectures?	How to improve convergence speed and robustness of optimization?
Residual connections	*Inception, depthwise separable convolutions, ...*	*Batch normalization layers*

Deep versus Shallow Neural Networks

- **Empirical observation 1**: given large enough data sets, deeper networks improve accuracy over shallower networks in many cases.
  - Not surprising: more powerful non-linear function representation
  - Transition from shallow neural networks used in the past (>10 years ago) to deeper architectures coined the term „deep learning“

Comparison of VGG networks of different depth

11 layers

19 layers

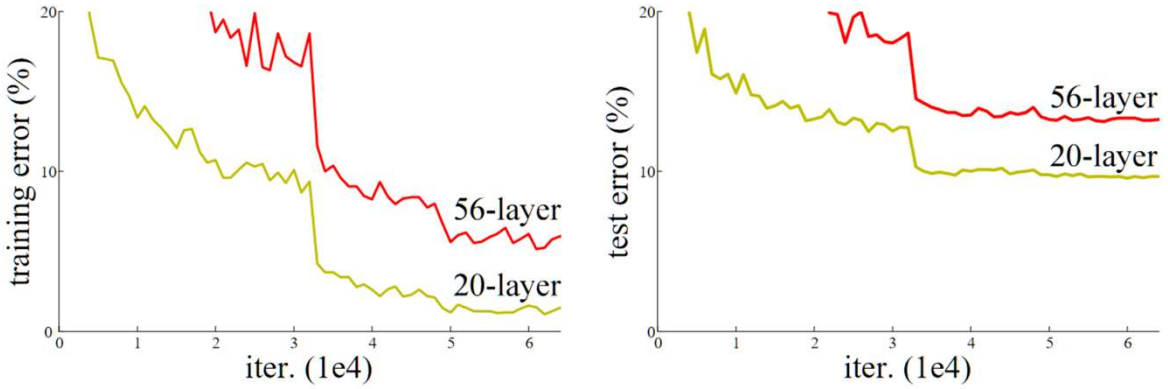
ConvNet config. (Table 1)	smallest image side		top-1 val. error (%)	top-5 val. error (%)
	train (S)	test (Q)		
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	[256;512]	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	[256;512]	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	[256;512]	384	**25.5**	**8.0**

Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

Deep versus Shallow Neural Networks

- **Empirical observation 2**: For depths larger than approx. 20 layers, accuracy starts to decline again for VGG-style architectures
  - Surprisingly, **training accuracy** declines: not a problem of overfitting
  - Adding more layers to a network increases representational power: the deeper network can represent any function the shallower network can represent
  - Points to a problem with learning: optimizer cannot find good weights

Training and test accuracies of VGG-style networks of depth 20 and 56 layers

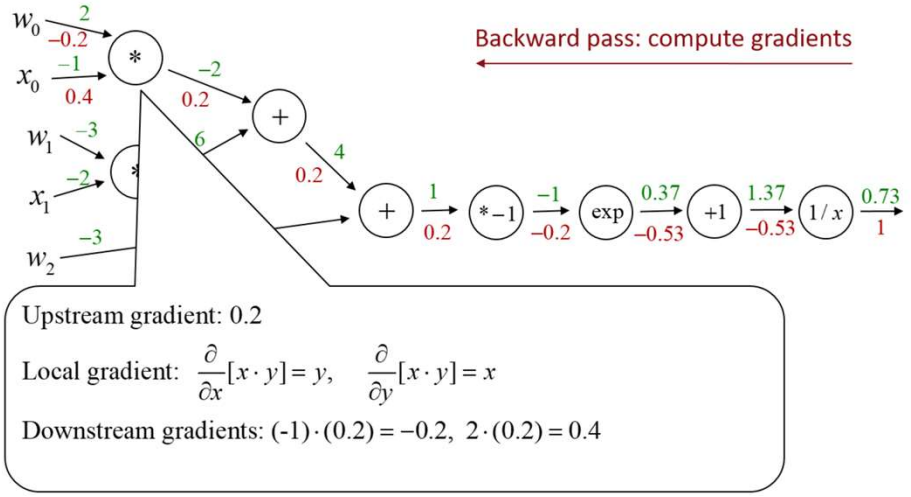


He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

Deep Models are Difficult to Optimize

- **Why does optimization fail to find good network weights for very deep networks?**
- One aspect is the so-called vanishing/exploding gradient problem:
  - At each layer, backpropagation multiplies upstream gradient with local gradient to compute downstream gradient
  - For deep networks, gradient at lower layers is computed by multiplying many local gradients. Depending on current weights this can lead to very small (local gradients < 1) or very large (local gradients > 1) gradient.

Backpropagation: multiplication of many local gradients



Deep Models are Difficult to Optimize

- Vanishing gradients result in very small updates to lower layers in the model: model takes extremely long to train or stops training altogether
- Exploding gradients result in unstable updates to lower layers and prevent convergence
- Intermediate normalization layers (see below) can help against vanishing/exploding gradients, but very deep models still do not perform well
- Good parameter initialization strategies can also help, but do not completely solve the problem in general
- Activation functions also matter: non-saturating activations such as ReLU pose less problems than saturating activation functions such as tanh, but even with ReLUs things gets difficult beyond approx. 20 layers

- **Overall, very deep models with gradient computations „chained“ across many layers result in difficult optimization problems**

Residual Architectures

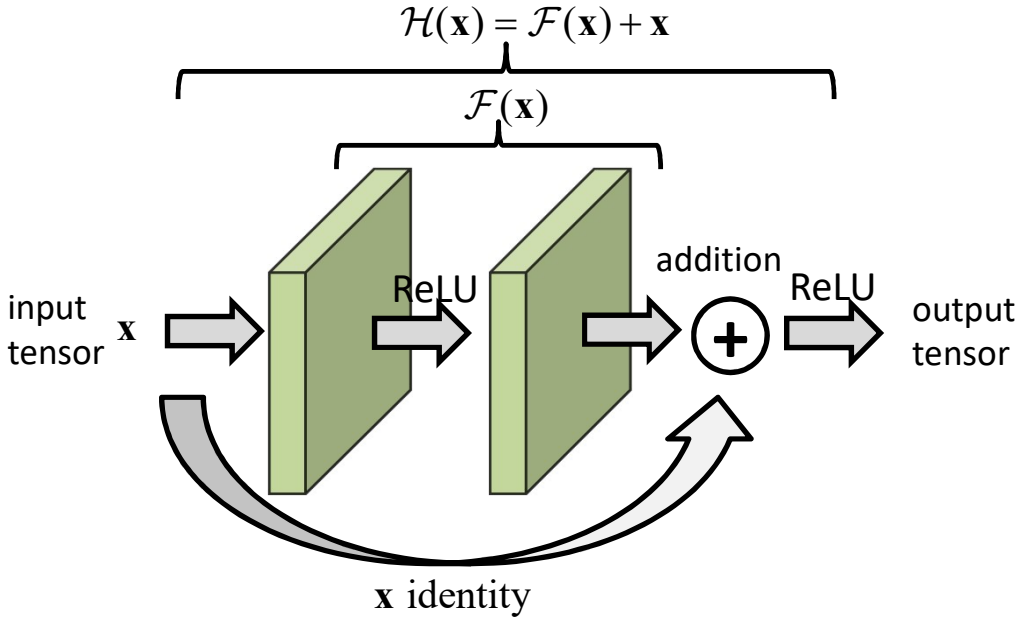
- **Residual neural networks [He et al., 2015] are architectures that contain „skip“ connections to improve gradient flow during optimization**
- Idea: Instead of using a stack of convolution layers to directly fit a desired function, fit the residual of the function

Residual building block:

Let $\mathcal{H}(\mathbf{x})$ denote the function we want to fit (with a stack of layers within the network).

Instead of fitting $\mathcal{H}(\mathbf{x})$ directly, we fit the residual $\mathcal{F}(\mathbf{x}) = \mathcal{H}(\mathbf{x}) - \mathbf{x}$.

This does not change the representational power of the stack of layers, but might make optimization easier



Assume dimension of input = dimension of output

Residual Architectures

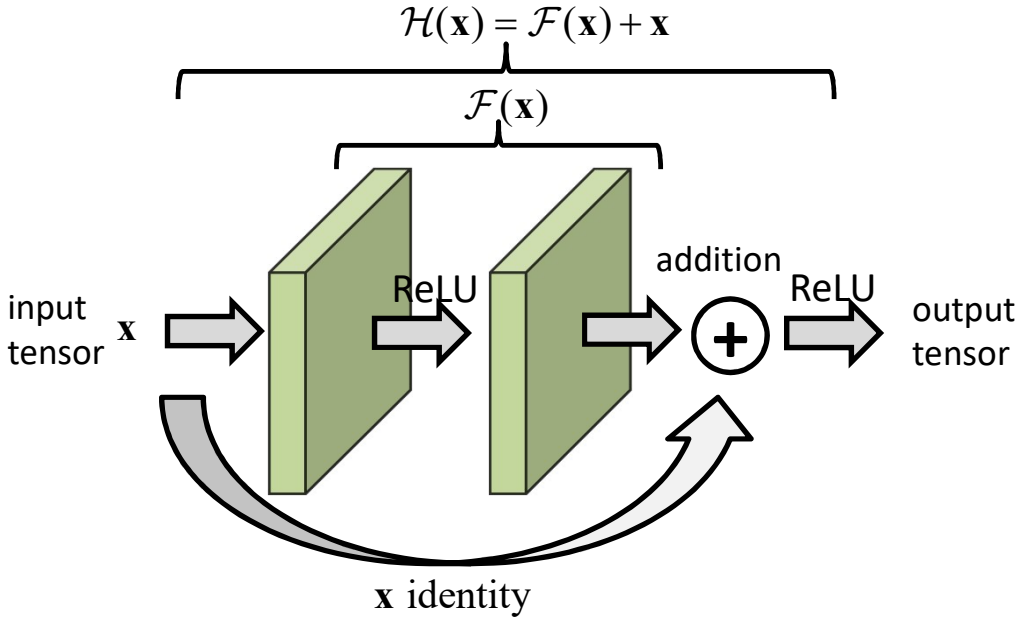
- **Residual neural networks [He et al., 2015] are architectures that contain „skip“ connections to improve gradient flow during optimization**
- Idea: Instead of using a stack of convolution layers to directly fit a desired function, fit the residual of the function

Why residual blocks?

Idea: deeper networks could represent shallower networks by learning the identity function in some layers (but optimizers seem to fail to do this).

In a residual block, learning identity is easy: just set weights to zero

Small weight initializations will lead to near-identity mappings initially, making the network approximately shallow

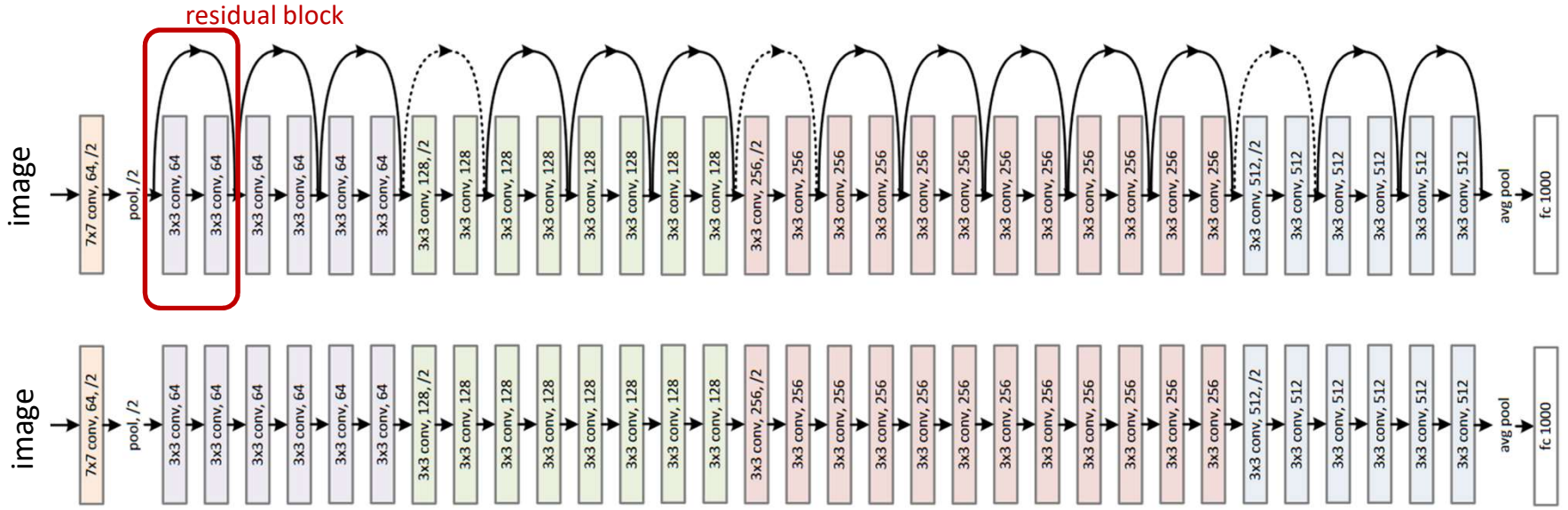


Assume dimension of input = dimension of output

Example: 34-Layer ResNet

- „**ResNets**“: Build up complete convolutional neural network architectures from residual building blocks

Plain VGG-style versus ResNet architecture (34 layers deep)

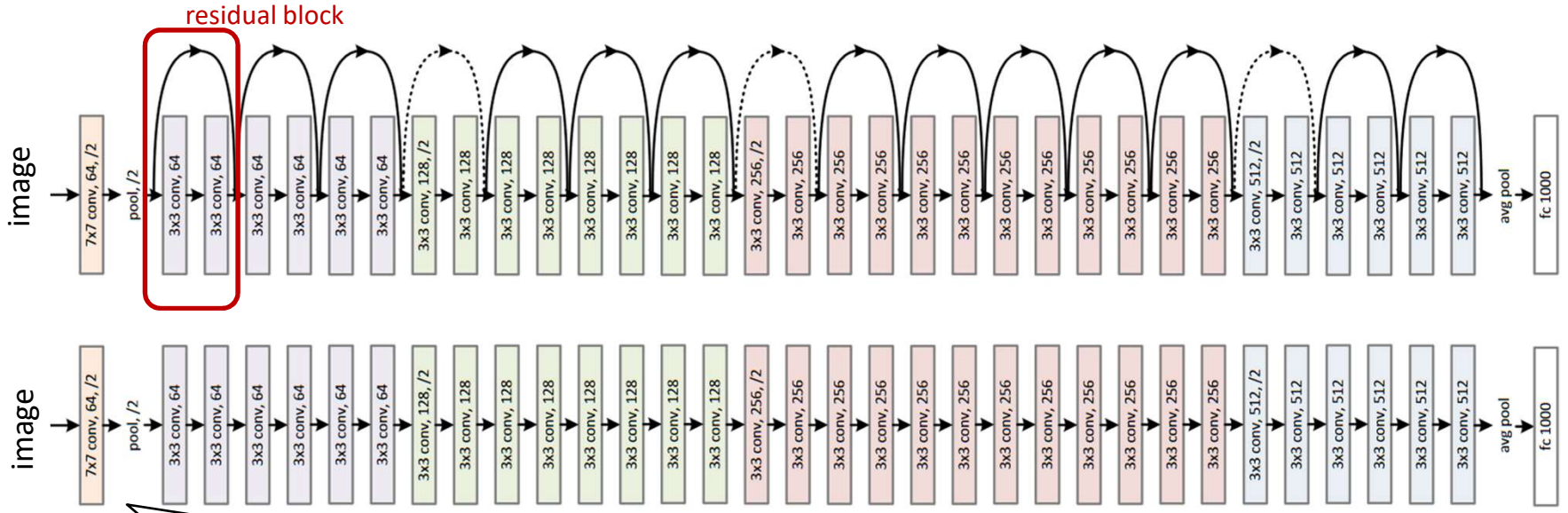


He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

Example: 34-Layer ResNet

- „**ResNets**“: Build up complete convolutional neural network architectures from residual building blocks

Plain VGG-style versus ResNet architecture (34 layers deep)

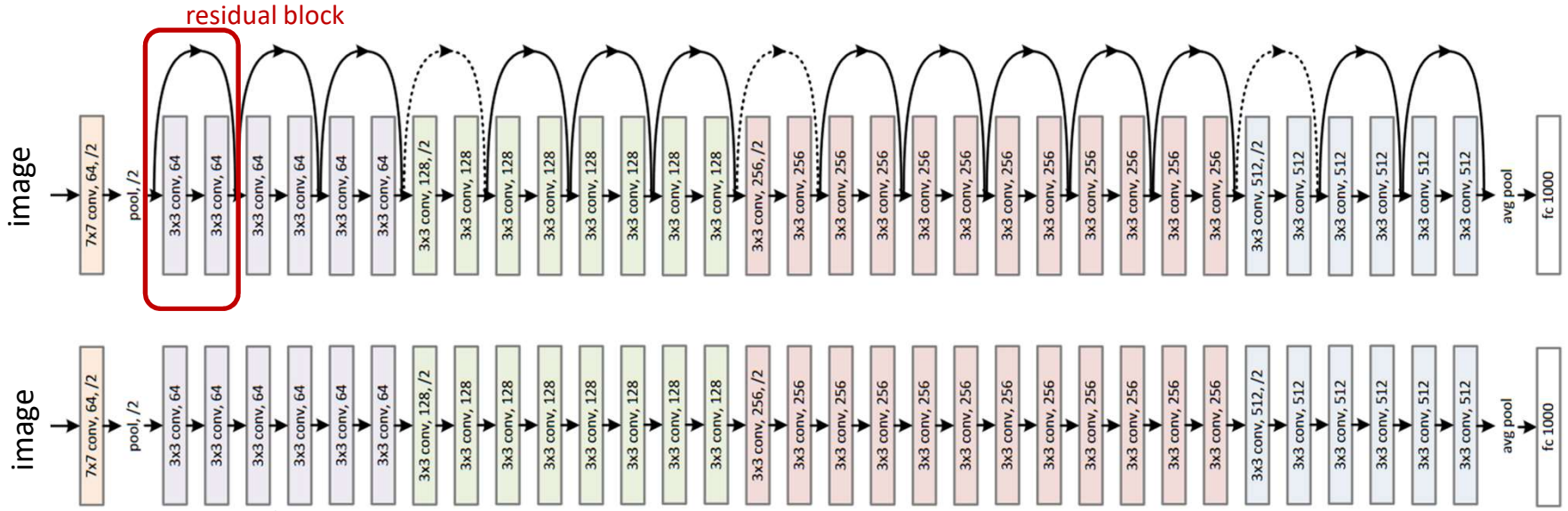


- Image input: 224x224x3
- Initial convolution layer with 7x7 kernel, 64 filters and stride two => 112x112x64
- Pooling layer with 2x2 kernel and stride 2 => 56x56x64

Example: 34-Layer ResNet

- „**ResNets**“: Build up complete convolutional neural network architectures from residual building blocks

Plain VGG-style versus ResNet architecture (34 layers deep)



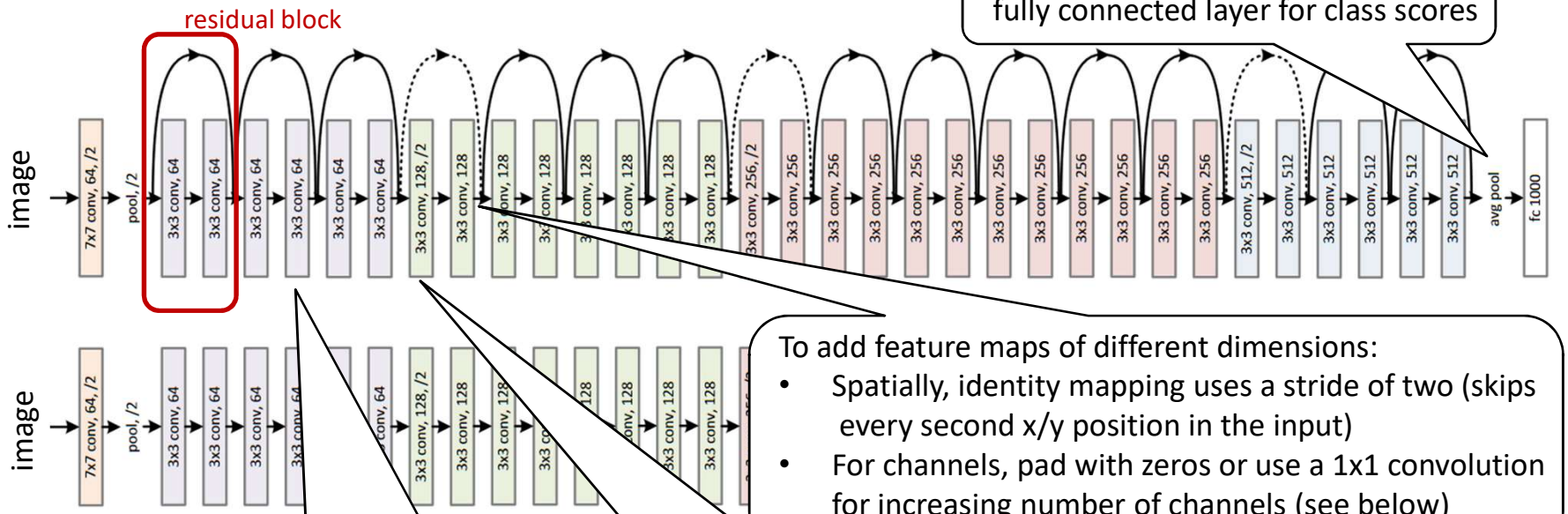
Stacked 3x3 convolution layers, 64 filters each

At three points in the network, strided convolution layer reduces spatial dimension (color change in figure). At the same time, number of filters is increased

Example: 34-Layer ResNet

- „**ResNets**“: Build up complete convolutional neural network architectures from residual building blocks

Plain VGG-style versus ResNet architecture (34 layers



Global average pooling before final fully connected layer for class scores

To add feature maps of different dimensions:
- Spatially, identity mapping uses a stride of two (skips every second x/y position in the input)
- For channels, pad with zeros or use a 1x1 convolution for increasing number of channels (see below)

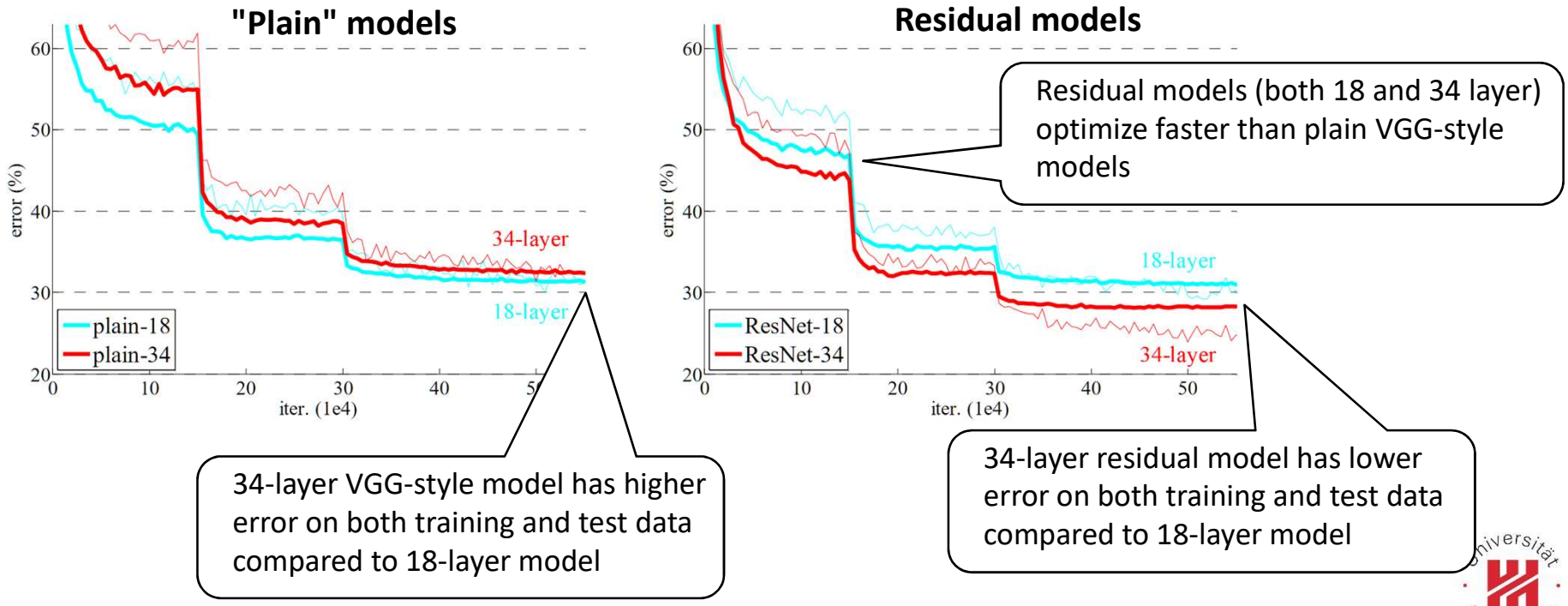
- Generally, residual blocks of same dimension are stacked.
- If weights are small, these blocks are close to identity function

Strided convolution with change of spatial dimensions and number of filters/channels.

Empirically, ResNets Outperform Plain Models

- Empirical comparison shows that residual architectures outperform VGG-style plain networks in terms of predictive accuracy
- Example: 34 and 18 layer VGG-style and residual models on ImageNet
- Many variants and improvements of ResNets have been developed

Error as a function of training iterations. Thin lines: training error, thick lines: test error



Advanced Convnet Architectures

- The relatively simple architecture concept of networks such as VGG has been refined and advanced in many ways
- The basic principle is still to stack multiple convolution layers with pooling or strided convolutions in between for reducing spatial dimensions, but certain extensions allow to construct more powerful and efficient networks

Increased Network Depth	**Power/efficiency Tradeoff**	Speed and Robustness of Optimization
How to train deep (> 20 layers) convnet architectures?	How to optimize computational efficiency of architectures?	How to improve convergence speed and robustness of optimization?
Residual connections	*Inception, depthwise separable convolutions, ...*	*Batch normalization layers*

Measuring Network Size and Complexity

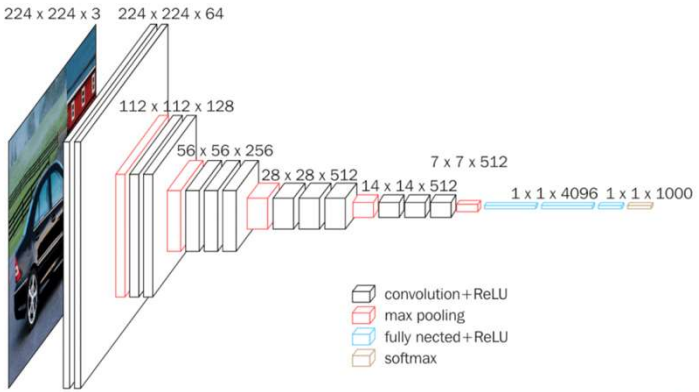
- Deep neural networks implement complex mathematical functions by stacking standard building blocks (convolutions, pooling, dense layers, ...)
- Complexity of resulting function can be characterized in terms of
  - the number of compute operations for making a single prediction. Measured in multiply-add operations

    Example: $z = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$ has four multiply-adds

  - the number of learnable parameters, which determines size of the model when stored on disk or in memory

Example: VGG-16 architecture

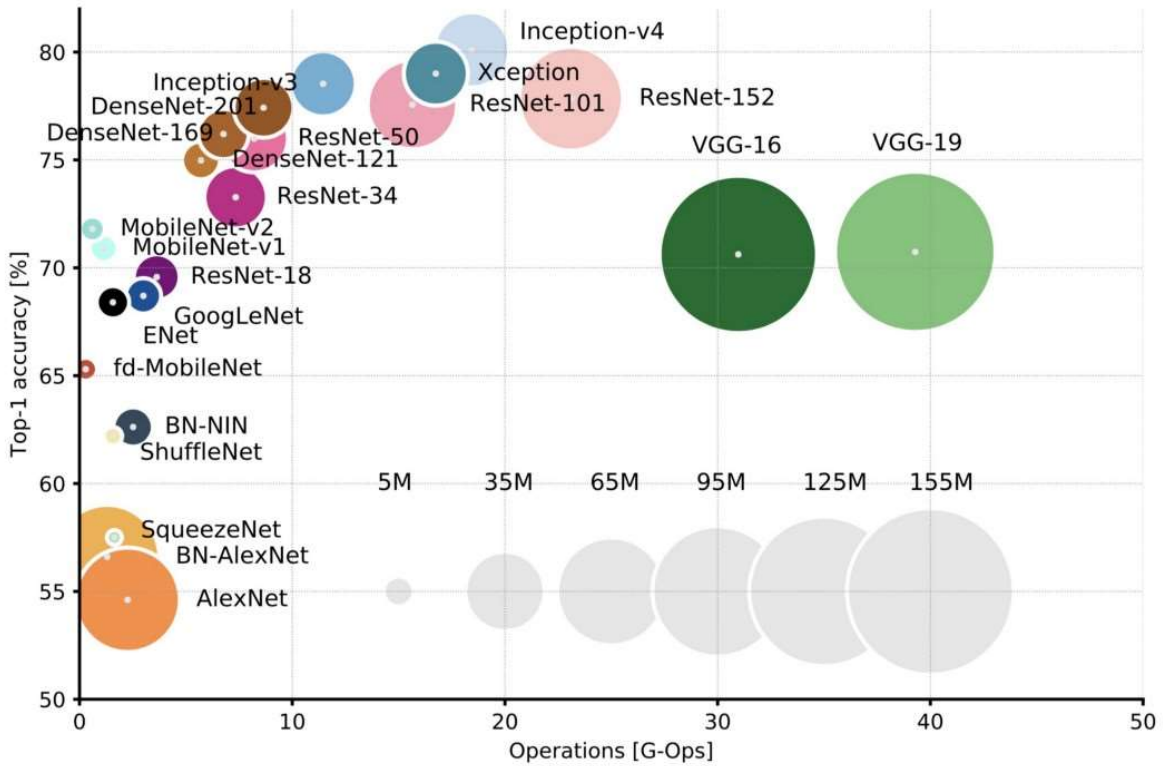
- Approx. 132.000.000 parameters
- Approx. 16.000.000.000 multiply-add operations for single prediction



Network Size versus Accuracy

- Empirical observation: Larger (and therefore computationally more expensive networks) generally outperform smaller networks

Top-1 accuracy on ImageNet as function of multiply-add operations



Canziani, Alfredo, Adam Paszke, and Eugenio Culurciello. "An analysis of deep neural network models for practical applications." *arXiv preprint arXiv:1605.07678* (2016).

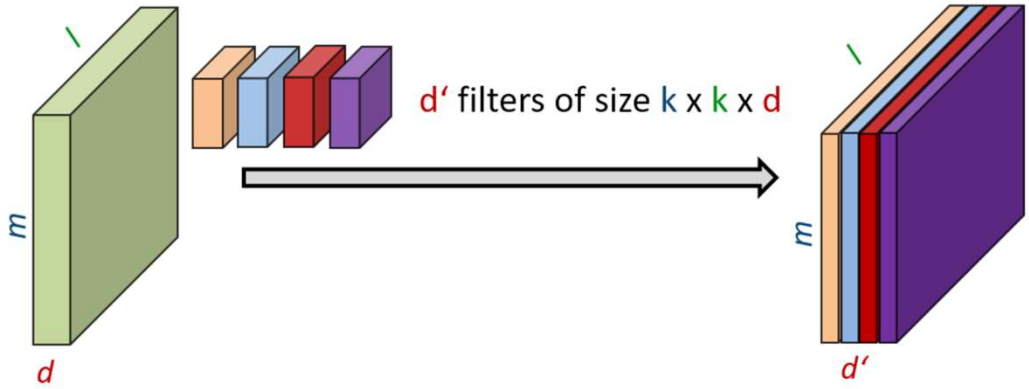
Reminder: Flops and Parameters per Layer

- Reminder: Number of multiply-add operations for (stride-one) convolution layer is $m \cdot l \cdot d'(k^2 d + 1)$.

Linear in both
spatial dimensions *m, l*

Linear in number of input
channels *d* and output filters *d'*

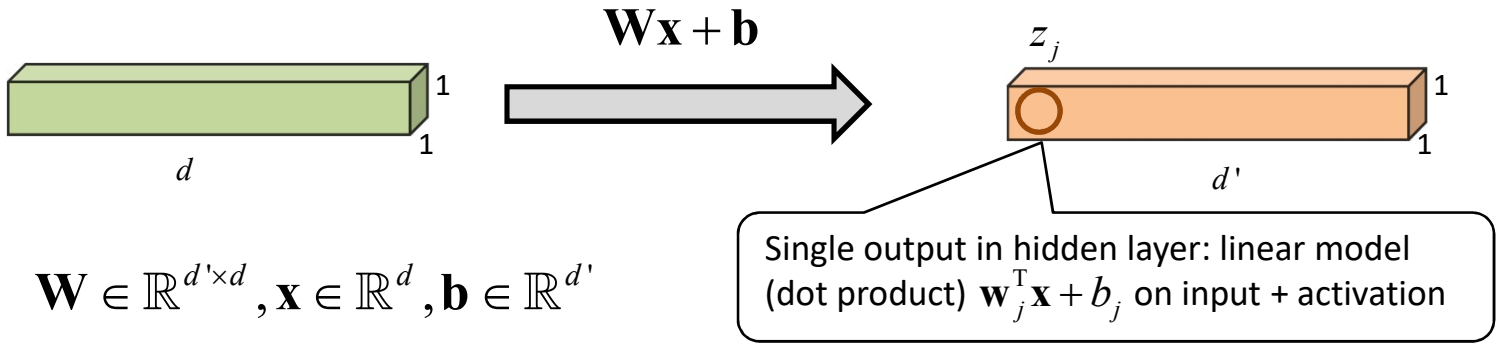
Quadratic in kernel size *k* (small)



- Number of parameters of convolution layer is $d' \cdot (k^2 d + 1)$

Reminder: Flops and Parameters per Layer

- For a fully connected (dense) layer, let d denote the number of input neurons and d' denote the number of output neurons

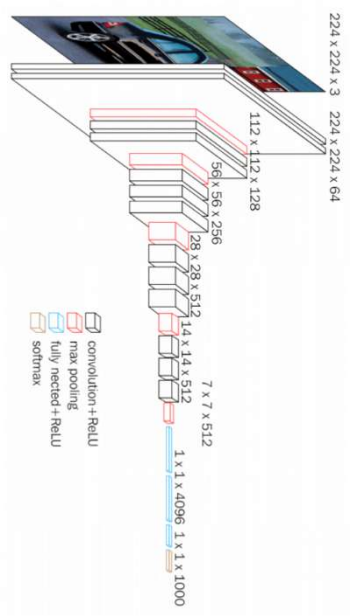


$$\mathbf{W} \in \mathbb{R}^{d' \times d}, \mathbf{x} \in \mathbb{R}^{d}, \mathbf{b} \in \mathbb{R}^{d'}$$

- Number of multiply-add operations in fully connected layer is $d' \cdot (d+1)$
- Number of parameters is $d' \cdot (d+1)$

Computational Effort and Parameters in Network

- For most convolutional neural network architectures, the majority of the computational effort is spent in convolution layers

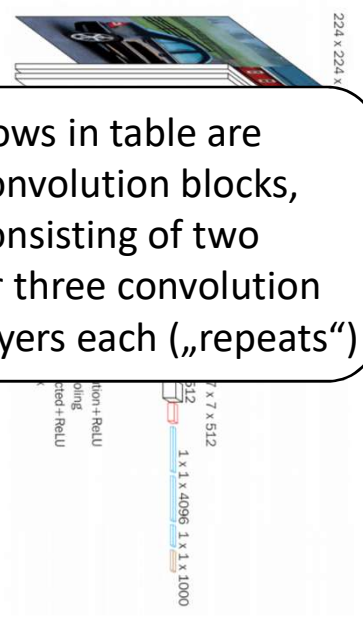


Layer	H	W	kernel H	kernel W	depth	repeats	FLOP/s
input	224	224	1	1	3	1	0.00E+00
conv1	224	224	3	3	64	2	1.94E+09
conv2	112	112	3	3	128	2	2.77E+09
conv3	56	56	3	3	256	3	4.62E+09
conv4	28	28	3	3	512	3	4.62E+09
conv5	14	14	3	3	512	3	1.39E+09
flatten	1	1	0	0	100352	1	0.00E+00
fc6	1	1	1	1	4096	1	4.11E+08
fc7	1	1	1	1	4096	1	1.68E+07
fc8	1	1	1	1	100	1	4.10E+05
							1.58E+10

most of compute spent here

Computational Effort and Parameters in Network

- For most convolutional neural network architectures, the majority of the computational effort is spent in convolution layers



Rows in table are convolution blocks, consisting of two or three convolution layers each („repeats“)

Layer	H	W	kernel H	kernel W	depth	repeats	FLOP/s
input	224	224	1	1	3	1	0.00E+00
conv1	224	224	3	3	64	2	1.94E+09
conv2	112	112	3	3	128	2	2.77E+09
conv3	56	56	3	3	256		4.62E+09
conv4	28	28	3	3	512		4.62E+09
conv5	14	14	3	3	512		1.39E+09
flatten	1	1	0	0	1003		0.00E+00
fc6	1	1	1	1			4.11E+08
fc7	1	1	1	1			1.68E+07
fc8	1	1	1				4.10E+05
							1.58E+10

most of compute spent here

Multiply-adds in first convolution layer: $224 \cdot 224 \cdot 64 \cdot (3^2 \cdot 3 + 1) = 89,915,392$
Multyply-adds in second convolution layer: $224 \cdot 224 \cdot 64 \cdot (3^2 \cdot 64 + 1) = 1,852,899,328$
Sum of the first two convolution layers: $1,942,814,720$

Designing Computationally Efficient Architectures

- Goal of architecture design: create a model that is as expressive as possible (many layers, high resolution, many filters) while keeping computational effort and memory requirements in check

- Architecture design can be manual or good architectures can be „discovered“ by automatically searching a large space of possible architectures, training and evaluating each architecture in the space

- Two frequently used ideas (next slides):
  - 1x1 convolutions that serve as „bottleneck“ layers to compress input that goes into a convolution layer and thereby reduce the computational effort
  - „depthwise separable convolutions“, a simplified and faster (although less expressive) variant of the standard convolution layer

1x1 Convolutions

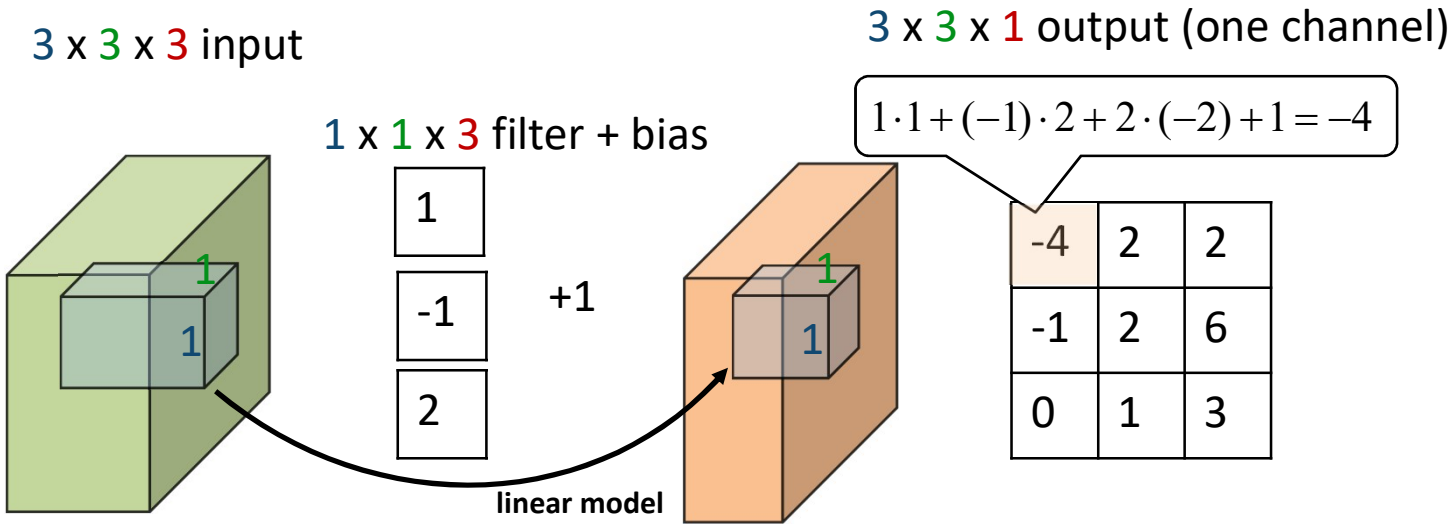
- A frequently used special case of the standard convolution operation is a convolution with a 1x1 kernel size (and stride 1)

Example with three input channels

1	-2	2
-1	0	1
0	2	1

2	-1	1
1	1	0
-1	0	1

-2	1	0
0	1	2
-1	-1	1

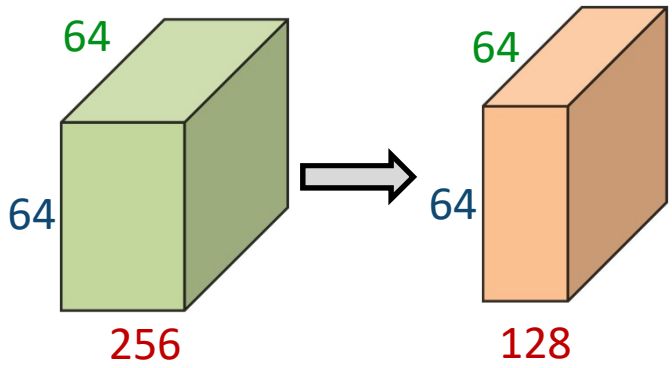


Spatial size of output is unchanged, number of channels can be increased or reduced.
Acts like a linear model in the channel dimension, applied independently to every x/y position

1x1 Convolutions as Bottleneck Layers

- A 1x1 convolution can be used to „compress“ the information in a layer by reducing the number of channels

For example, going from 64x64x256 to 64x64x128



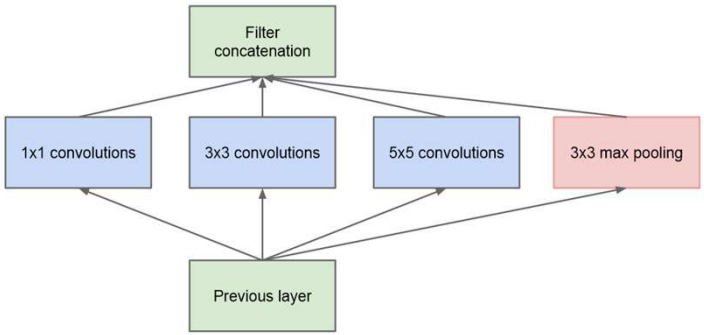
- Implements a linear mapping from the original 256-d space to a new 128-d space. Can think of this as similar to a principal component analysis
- Because the model is trained end-to-end, mapping will be learned that preserves as much (useful) information as possible from the 256-d representation
- If we put such a „bottleneck“ layer before a convolution layer, it will halve computational effort in the convolution layer

Inception Architecture

- Inception architecture [Szegedy et al, 2015]: multi-scale convolutions with bottleneck layers. Stacks so-called inception modules on top of each other

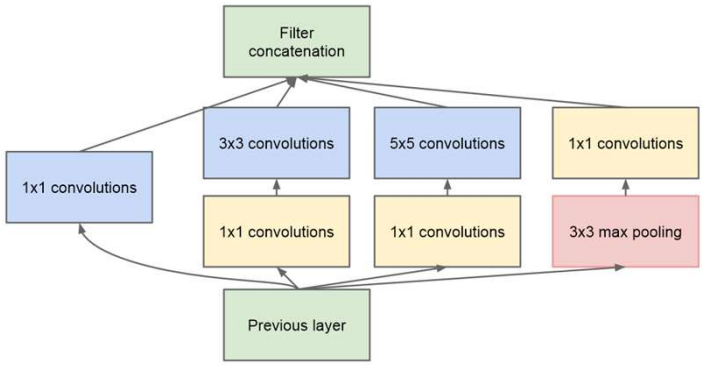
Inception module (naive):

- Process input to module simultaneously with convolutions of different kernel sizes and a stride-one max pooling layer
- Concatenate the output of all these layers in the channel dimension



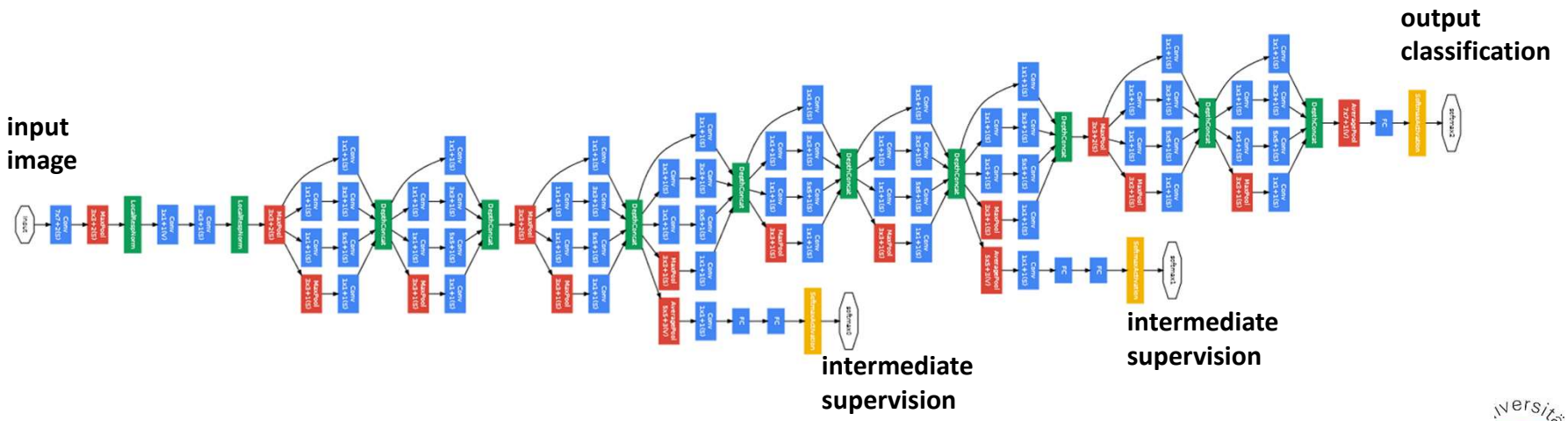
Inception module (with bottlenecks):

- Problem with naive inception is that number of channels gets very large in output, making subsequent convolution layers very expensive
- Solution: add 1x1 convolutions as bottleneck layers before convolution layers



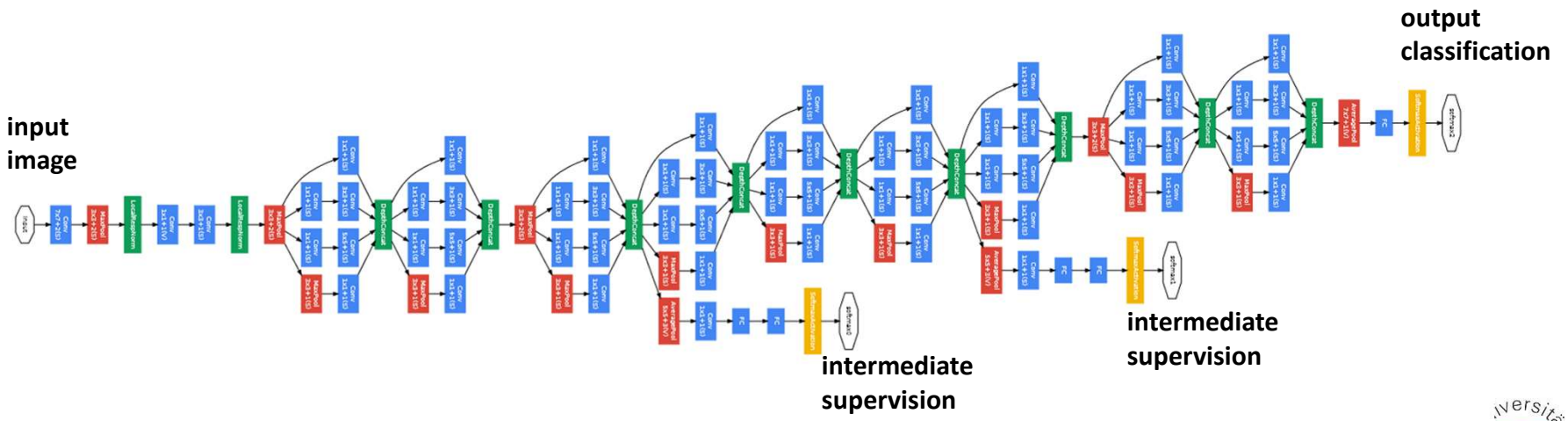
Inception Architecture

- GoogLeNet: model constructed by stacking several inception modules on top of each other
- Intermediate supervision is also added at two stages in network
  - Try to predict class based on representation generated by network so far
  - Aim is to allow more robust training of lower parts of model



Inception Architecture

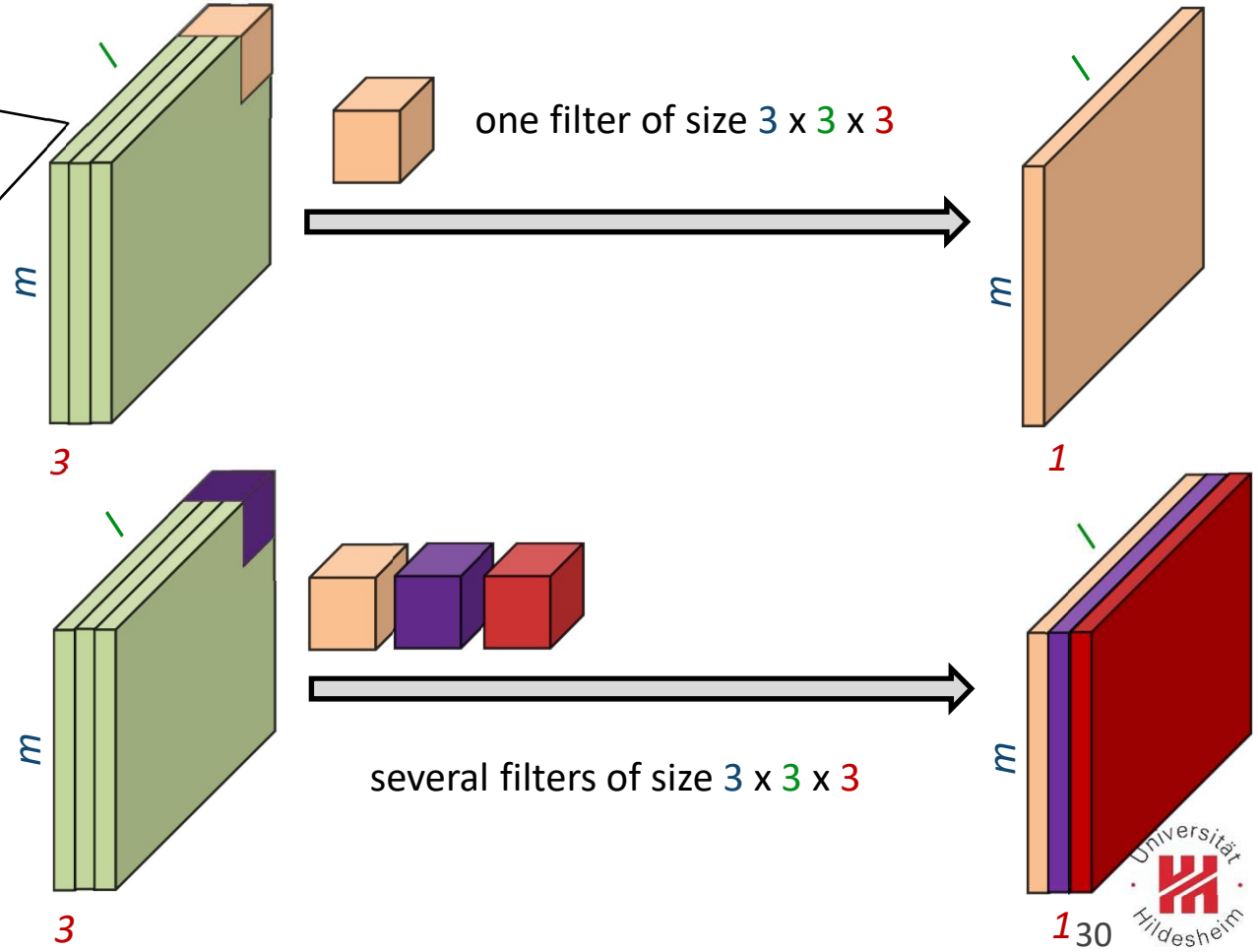
- Model has a cost of approximately 2,000,000,000 multiply-adds and approximately 13,000,000 parameters
- At the 2014 ImageNet competition, GoogleNet was competitive to VGG-16 despite having approximately an order of magnitude fewer parameters and computational cost
- Inception architecture has been further developed over the last years



Normal Convolution: Each Filter Operates On Entire Depth (Channels) in Input

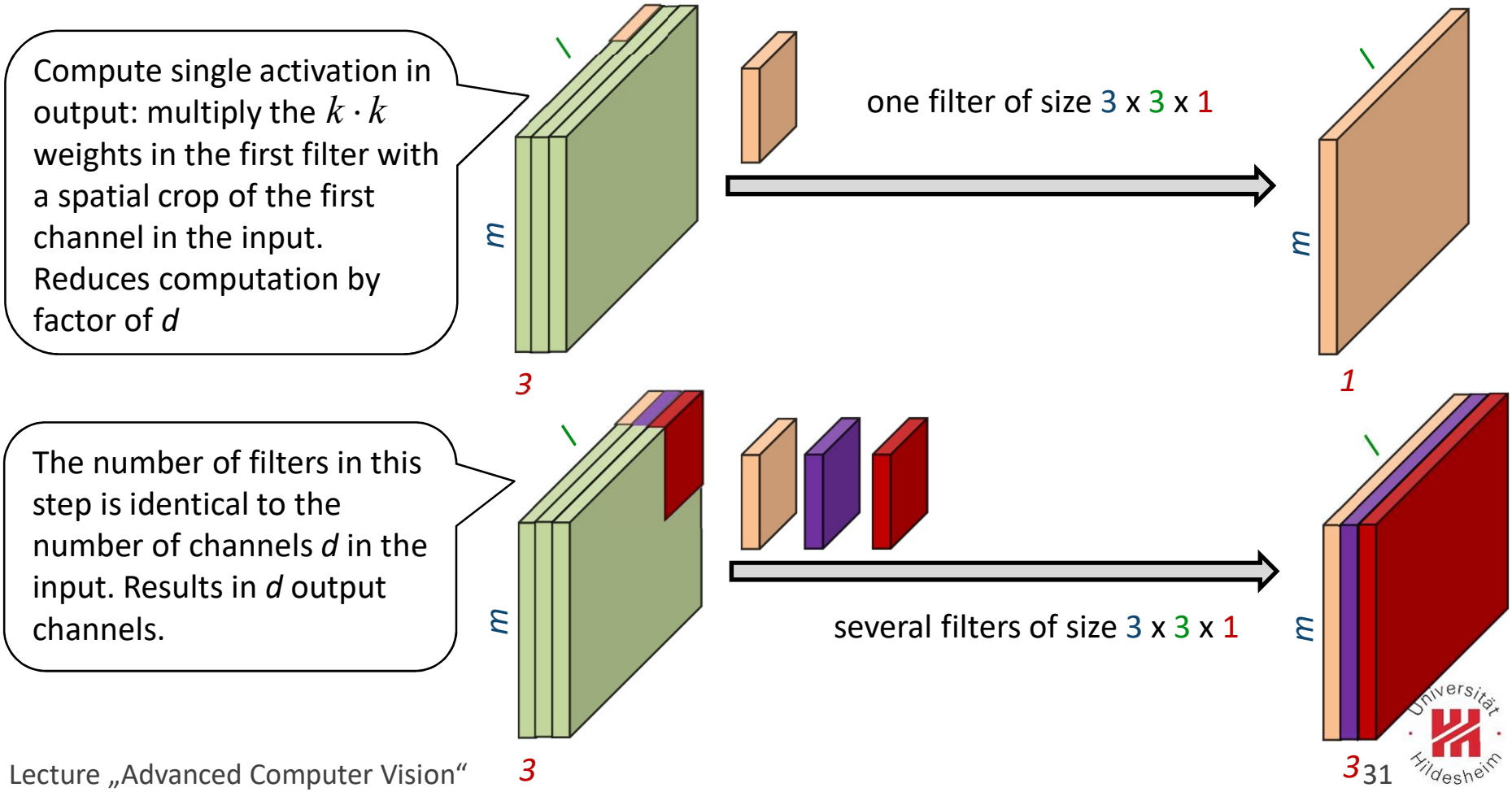
- Most of the computational effort is typically spend in the convolution layers
- Cost of computing a single activation in output: $k^2 d + 1$

Compute single activation in output: multiply the $k \cdot k \cdot d$ weights in the filter with a spatial crop in the input.
k is typically small (e.g. *k*=3)
d is 3 in example but gets large in later layers (e.g. *d*=1024)



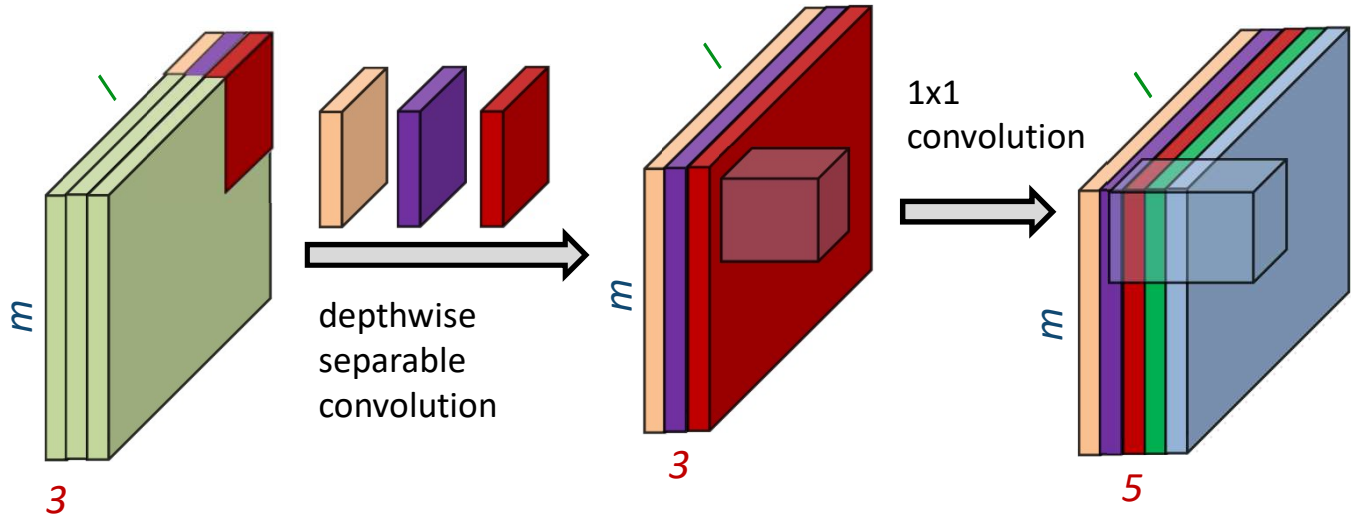
Depthwise Separable Convolutions (1. Step)

- **Depthwise separable convolution**: each filter only operates on a single input channel (first step)



Depthwise Separable Convolutions (2. Step)

- First step in depthwise separable convolution is usually followed by a 1x1 convolution as a second step to arrive at an arbitrary number of channels in the output

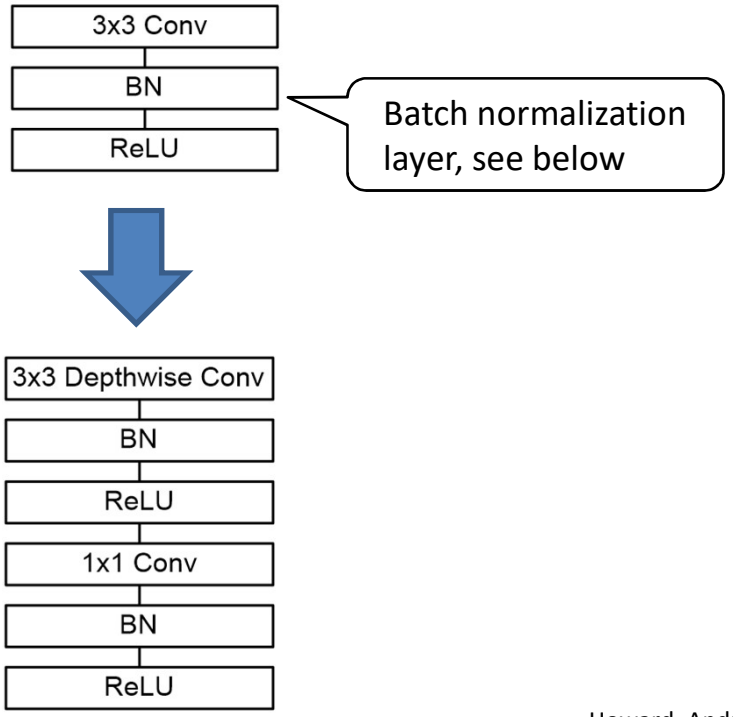


- The first step only operates spatially, for each channel independently
- The second step only operates in the channel dimension, for each spatial location independently

MobileNets

- **MobileNets**: efficient, widely used architecture for mobile/embedded vision based on depthwise separable convolutions

Replace standard 3x3 convolution with 3x3 depthwise separable convolution plus 1x1 convolution



Full architecture

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	$1 \times 1 \times 1000$

Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).

Advanced Convnet Architectures

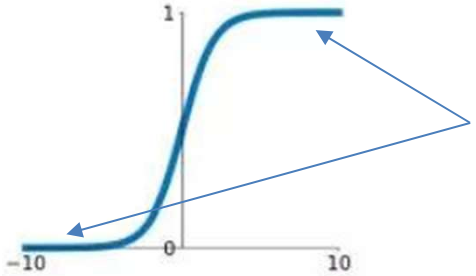
- The relatively simple architecture concept of networks such as VGG has been refined and advanced in many ways
- The basic principle is still to stack multiple convolution layers with pooling or strided convolutions in between for reducing spatial dimensions, but certain extensions allow to construct more powerful and efficient networks

Increased Network Depth	Power/efficiency Tradeoff	**Speed and Robustness of Optimization**
How to train deep (> 20 layers) convnet architectures?	How to optimize computational efficiency of architectures?	How to improve convergence speed and robustness of optimization?
Residual connections	*Inception, depthwise separable convolutions, ...*	*Batch normalization layers*

Deep Networks Train Slowly

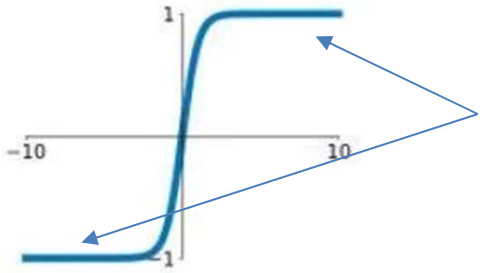
- **Empirical observation: deep networks often train very slowly**
- One possible cause is the vanishing gradient problem discussed above
  - local gradients might be small and the gradient at lower layers, which is a product of many local gradients, becomes very small
  - when using saturating activation functions, inputs can move into the saturated regime of the activation function, resulting in small gradients

$$\text{sigmoid: } \sigma(z) = \frac{1}{1+\exp(-z)}$$



saturated regime

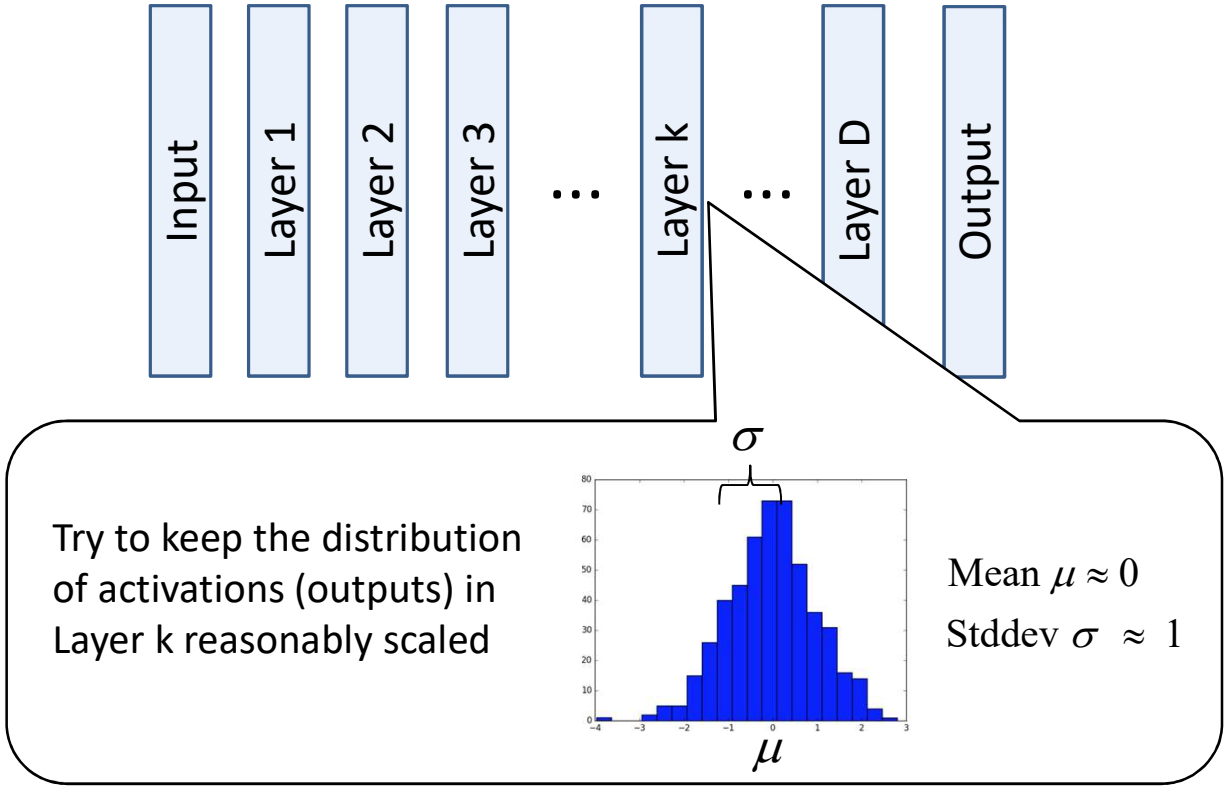
$$\tanh(z)$$



saturated regime

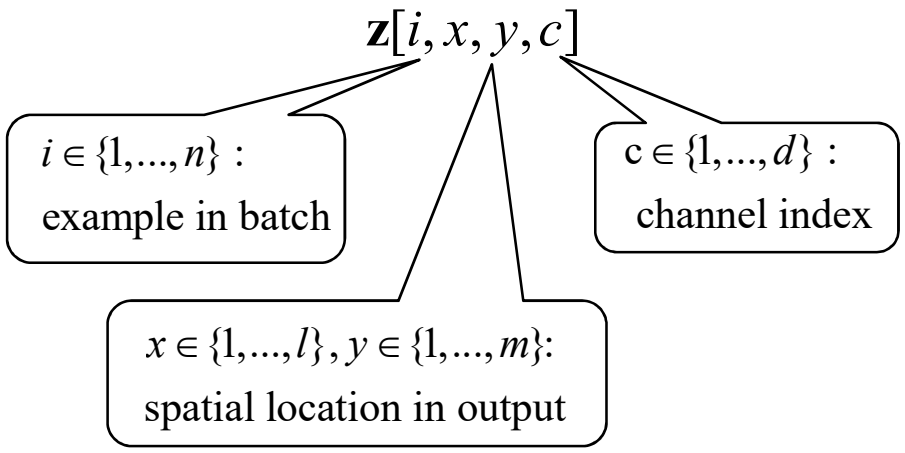
Controlling the Scaling of Outputs of Layers

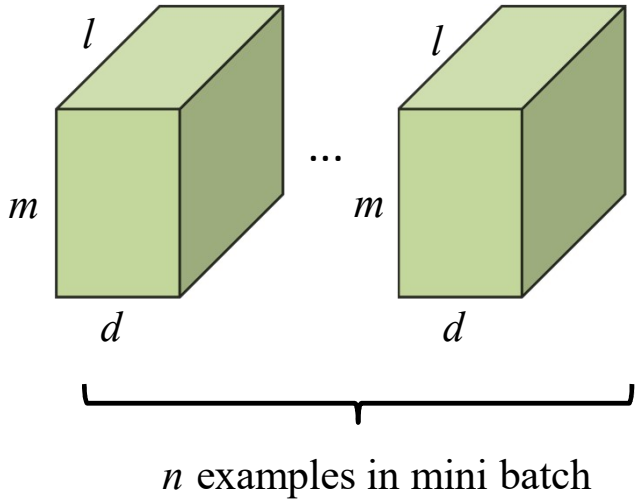
- **Idea:** can we control the scaling of the output at intermediate layers of the network?
- Keeping the output of intermediate layers reasonably scaled might help against vanishing/exploding gradients and saturation of activation functions



Batch Normalization Layer

- **Batch normalization layer:** renormalize the mean and standard deviation of outputs of a layer using minibatch statistics
- Batch output $\mathbf{z}$ of a convolution layer: 4D tensor of dimension $n \cdot m \cdot l \cdot d$





Batch Normalization Layer

- **Idea:** normalize to zero mean and unit standard deviation, per channel
  - Normalized output:

$$\hat{\mathbf{z}}[i,x,y,c] = \frac{\mathbf{z}[i,x,y,c] - \mu_c}{\sigma_c} \qquad (1)$$

  - Mean and standard deviation are estimated over the different instances in the mini batch, and the spatial locations

$$\mu_c = \frac{1}{nlm}\sum_{i,x,y} \mathbf{z}[i,x,y,c] \qquad \sigma_c = \sqrt{\frac{1}{nlm}\sum_{i,x,y}(\mathbf{z}[i,x,y,c] - \mu_c)^2}$$

- Note: operation (1) is fully differentiable, including the computation of μ_c, σ_c
- We can thus put it in a layer in the neural network and backpropagate through it to obtain the gradient, which is needed for training

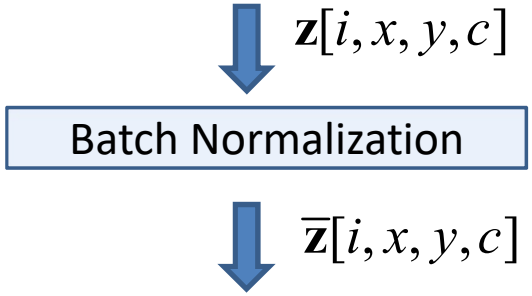
Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *International conference on machine learning*. PMLR, 2015.

Batch Normalization Layer

- But do we really want zero mean and unit variance? Maybe this reduces the representational power of the network too much
- Let us instead normalize to arbitrary mean β_c and standard deviation α_c :

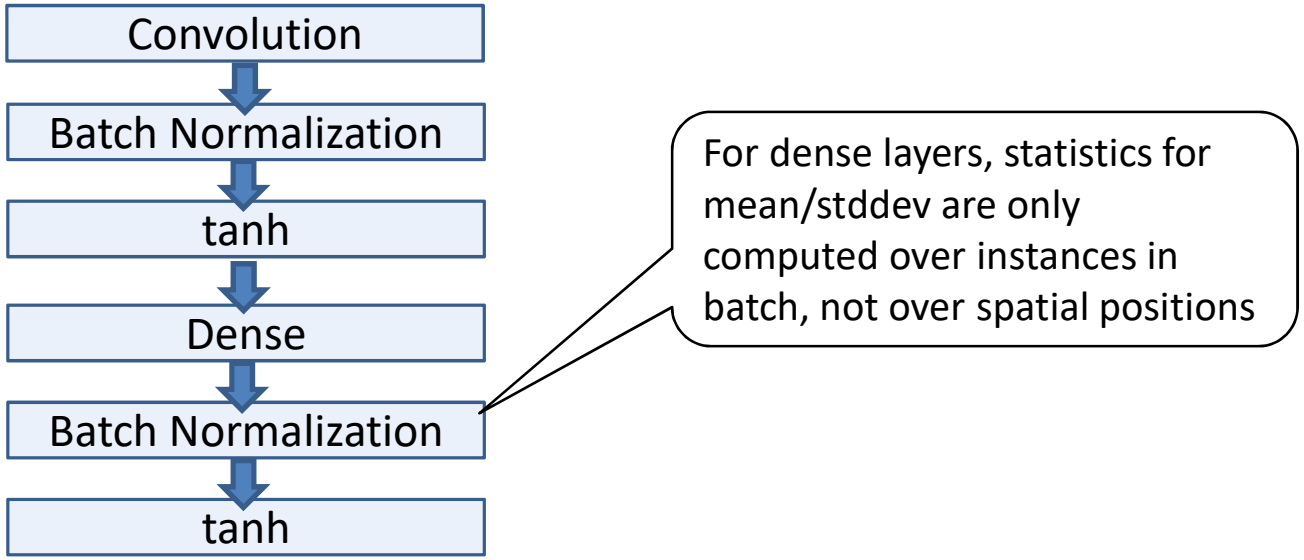
$$\overline{\mathbf{z}}[i,x,y,c] = \alpha_c \hat{\mathbf{z}}[i,x,y,c] + \beta_c$$

- The parameters α_c, β_c are treated as weights and learned by backpropagation
- The computation from $\mathbf{z}[i,x,y,c]$ to $\overline{\mathbf{z}}[i,x,y,c]$ is put into a neural network layer, which does not change the dimension of the feature map in any way, is fully differentiable, and contains learnable parameter α_c, β_c for all channels



Where To Put Batch Normalization Layer

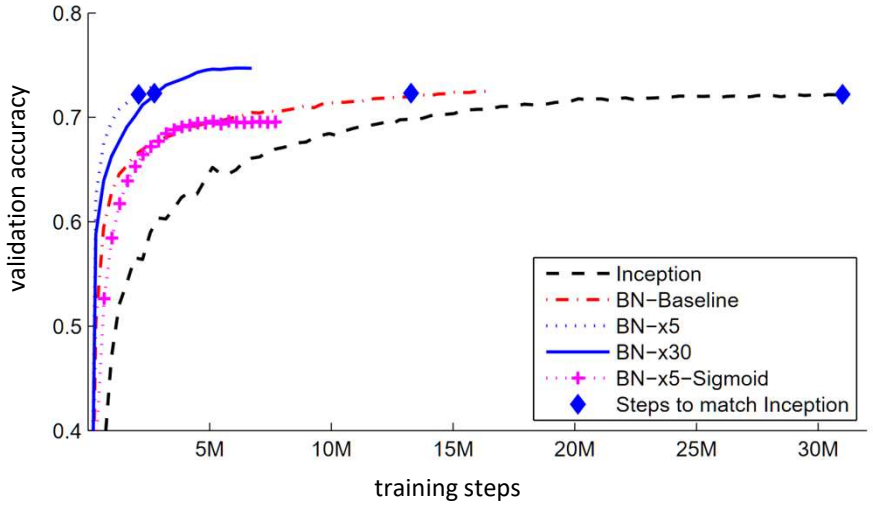
- Batch normalization layers are usually placed after convolutional layers or fully connected/dense layers, but before the nonlinear activation:



- At test time, batch normalization layers function slightly differently:
  - the mean/std are not computed based on the batch, but instead a single fixed empirical mean of activations obtained during training is used
  - can be estimated during training e.g. with running averages

Empirical Results Batch Normalization

- Empirically, batch normalization usually leads to
  - improved gradient flow through the network
  - faster training of deep models, mostly by allowing higher learning rates without making the optimization diverge
  - more robust training that does not depend as much on good parameter initializations
  - can also act as a form of regularization (this is not yet well understood)



Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *International conference on machine learning*. PMLR, 2015.

Summary: Convolutional Neural Network Architectures

- Convolutional neural network architectures pass 3D-Tensors through network that respect the spatial arrangement of pixels
- Convolution and max pooling layers also specifically operate on these spatially arranged tensors
- Through layers, spatial dimension is reduced while the number of channels is increased to learn more and more abstract features
- Modern architectural elements such as residual connections, bottleneck layers, batch normalization (and others we did not yet talk about) improve accuracy and efficiency of models