Convolutional Neural Networks (CNNs)

MMAI 5500 – lecture 4
Fall 2024

Content

Network size & local minima

Neural network architectures

Overfitting & weight sharing

Convolutions

Convolutional neural network

- Architecture overview
- CNN standard layer types
    - Convolutional layer
    - Pooling layer

CNN continued

- Fancy CNN architectures

Why deep CNNs are better

Vanishing gradients

CNN in practice

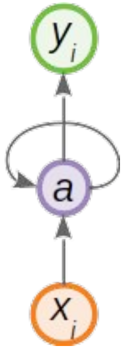
- Transfer learning

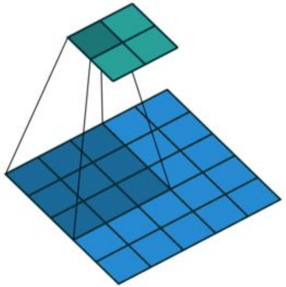
Tutorial: transfer learning

Overfitting
- Weight sharing

- Many & varied training exemplars (better estimate of the distribution)
- Regularization (penalize weight growth)
- Drop out (randomly remove neurons)
- **Decrease network size (fewer weights)**
- **Weight sharing** (few weights)

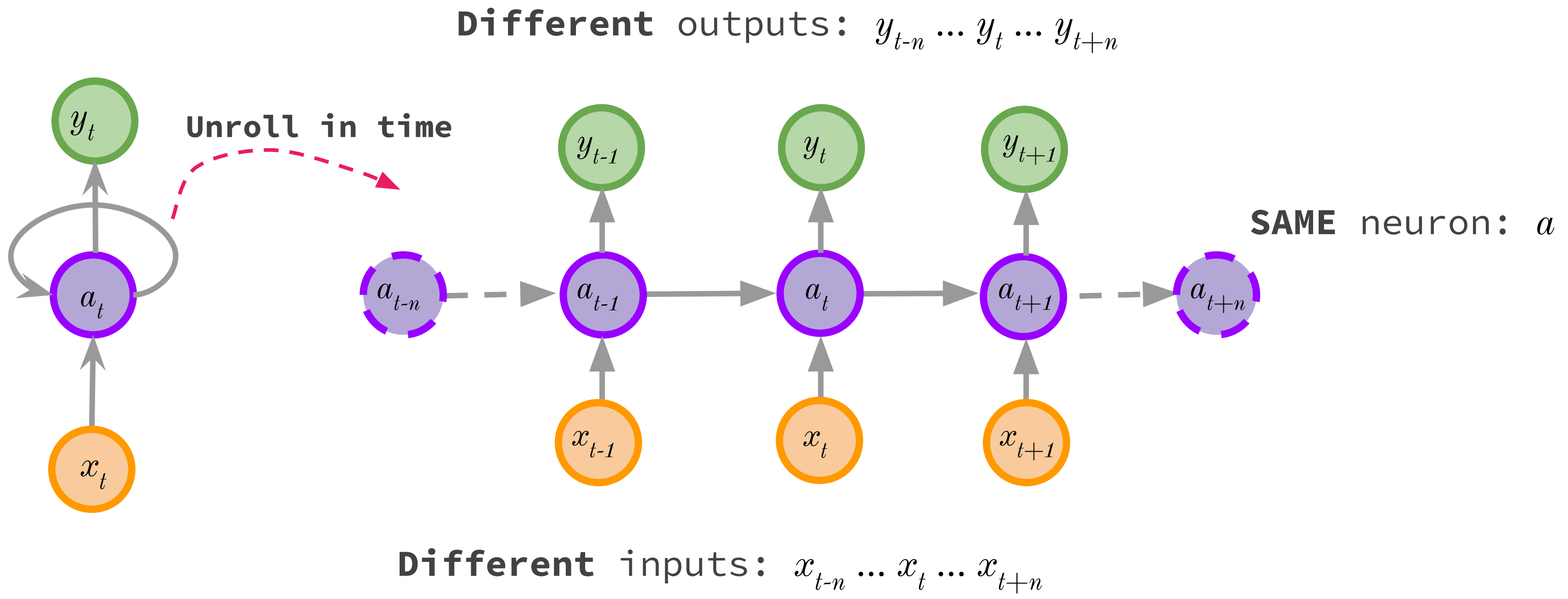
Weight sharing





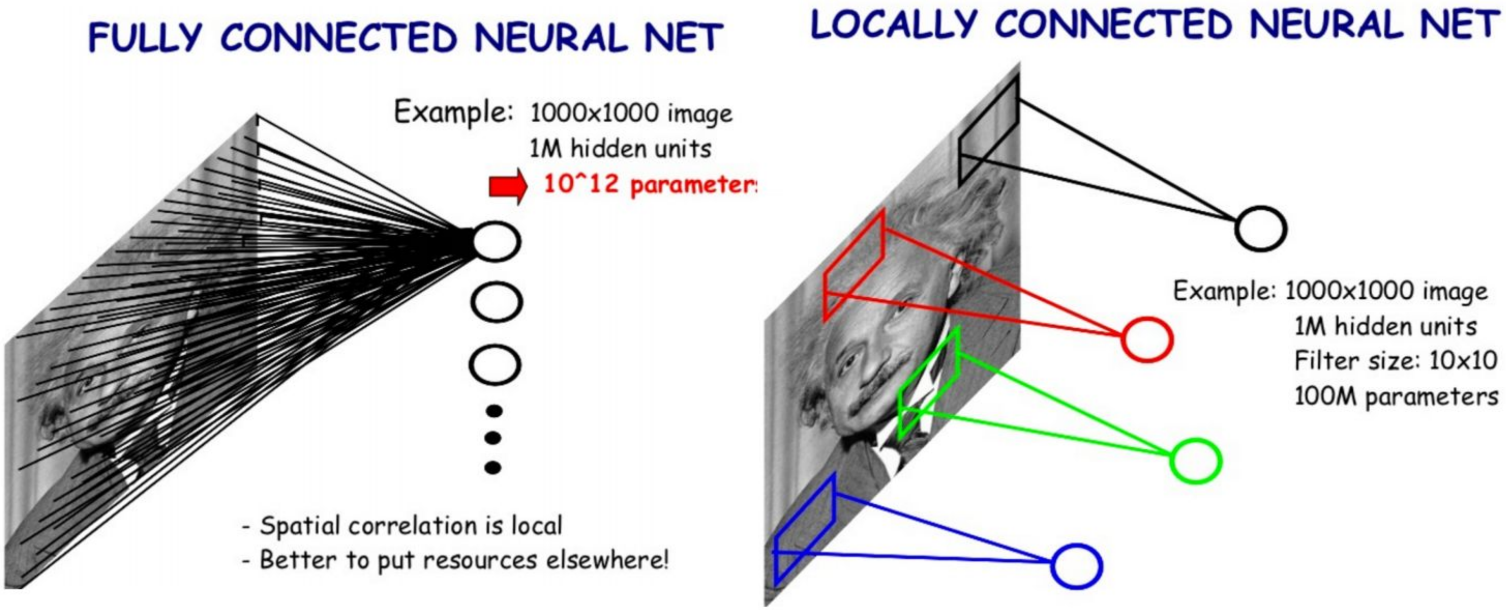
Weight sharing

- Recurrent Neural Network (RNN)



Fully vs locally connected

- Decreasing network size independent of capacity



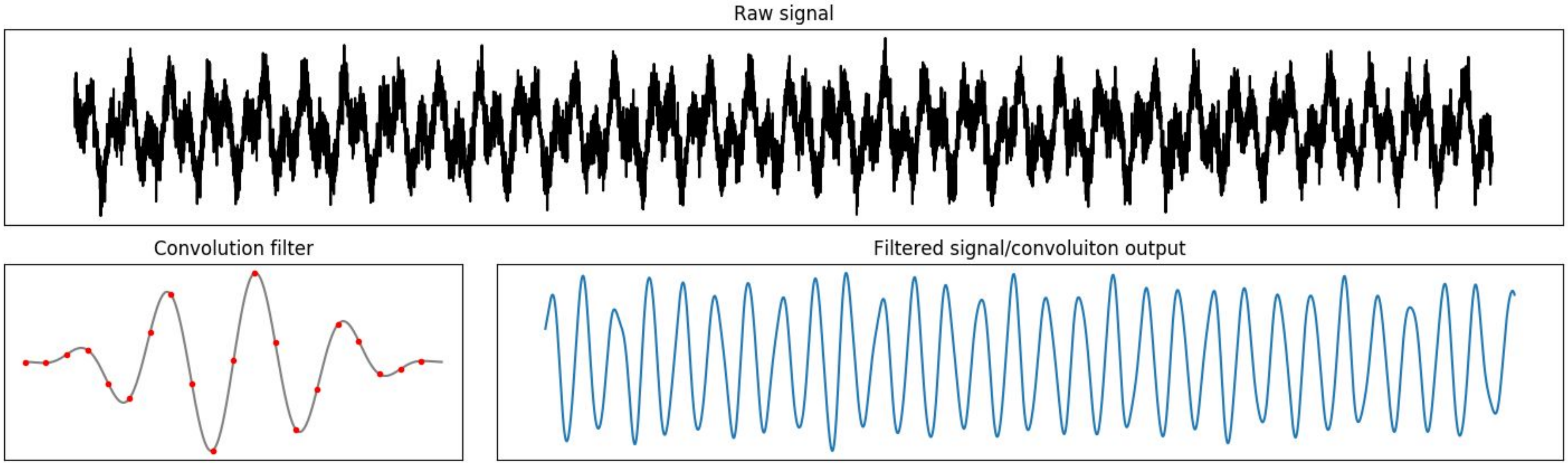
Convolutions

- In math & neural networks

Convolutions

- In mathematics

What is a **convolution** in mathematics?



Convolutions

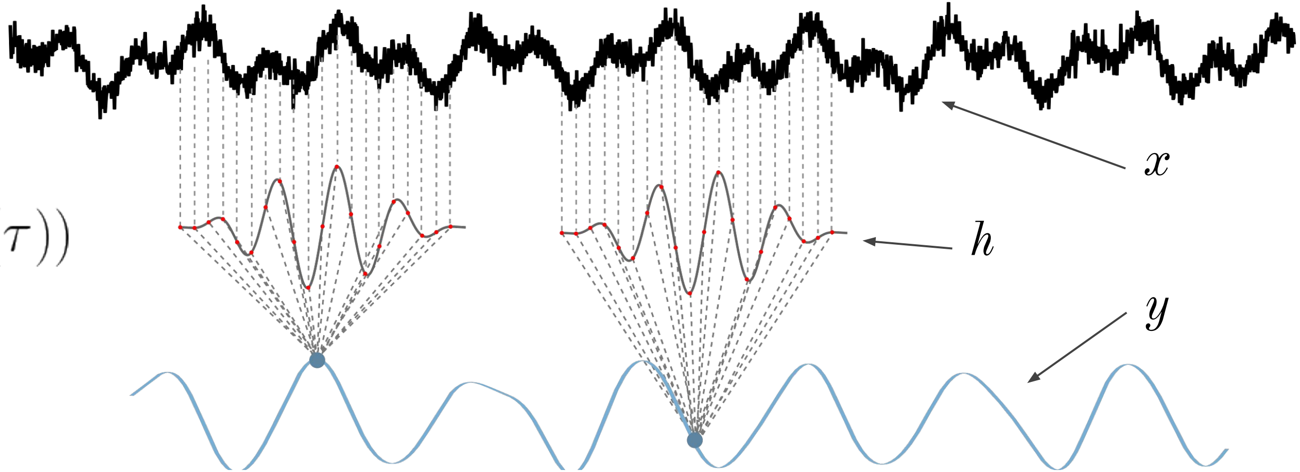
- In mathematics

What is a **convolution** in mathematics?

Similar to

- Cross-correlation
- Autocorrelation

$$y(t) = \sum_{\tau=-\infty}^{\infty} x_{t-\tau} * h(\tau))$$



Convolutions in 2D

- In neural networks

Question: What are x, h & y?
Question: How many weights does the filter have?

Input: blue

Output: cyan

Filter: vertical lines

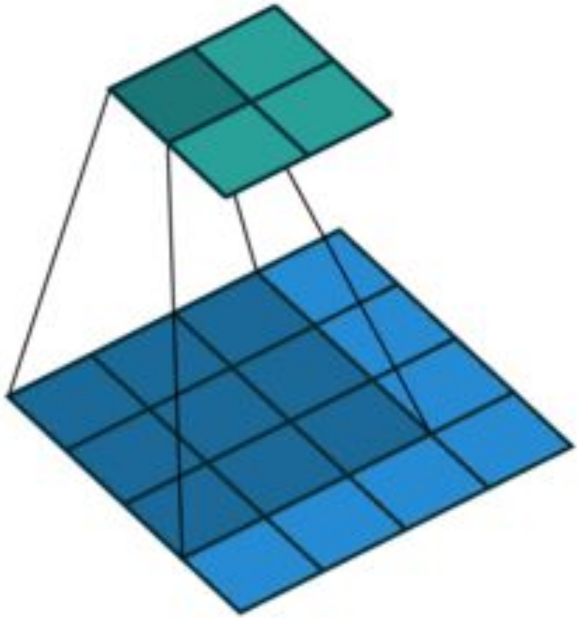


Image credits: Vdumoulin; published under MIT licence; https://github.com/vdumoulin/conv_arithmetic

2D convolutions

- In neural networks

Variations

Question: What is kernel_size?

???

Keras

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    None,
    ne,
```

???

PyTorch

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

2D convolution with *padding*

- In neural networks

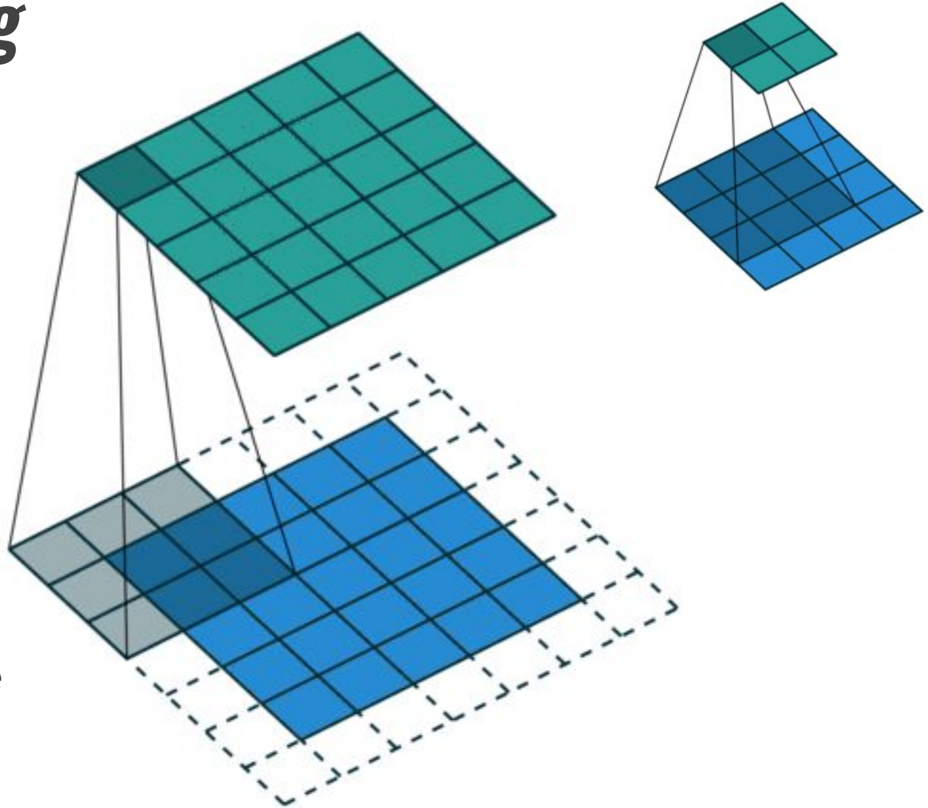
Padding

Input: blue

Output: cyan

Filter: vertical lines

Padding: *same* (input & output has same size)



Padding: valid

Image credits: Vdumoulin; published under MIT licence; https://github.com/vdumoulin/conv_arithmetic

Question: What is the result of using strides?

2D convolution with *stride*

- In neural networks

Stride

Input: blue

Output: cyan

Filter: vertical lines

Stride: 2 (moves 2 steps)

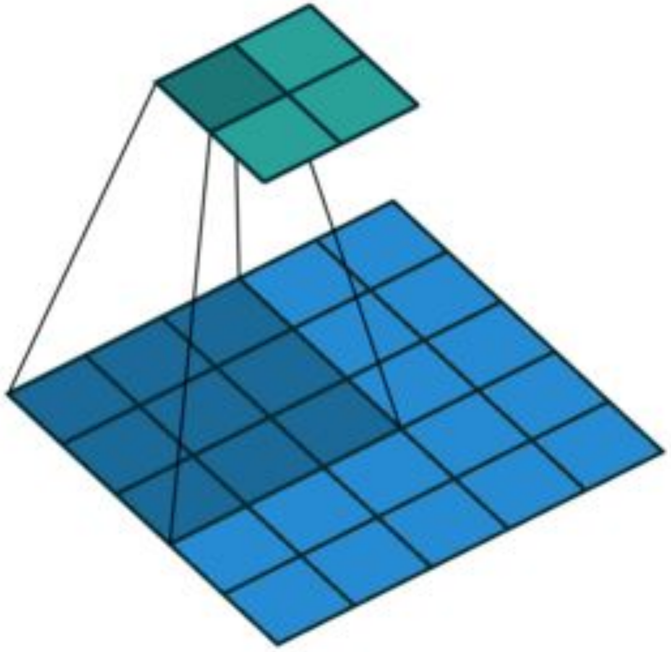


Image credits: Vdumoulin; published under MIT licence; https://github.com/vdumoulin/conv_arithmetic

2D convolution with *stride* & *padding*

- In neural networks

Input: blue

Output: cyan

Filter: vertical lines

Padding: 1 (not same)

Stride: 2 (moves 2 steps)

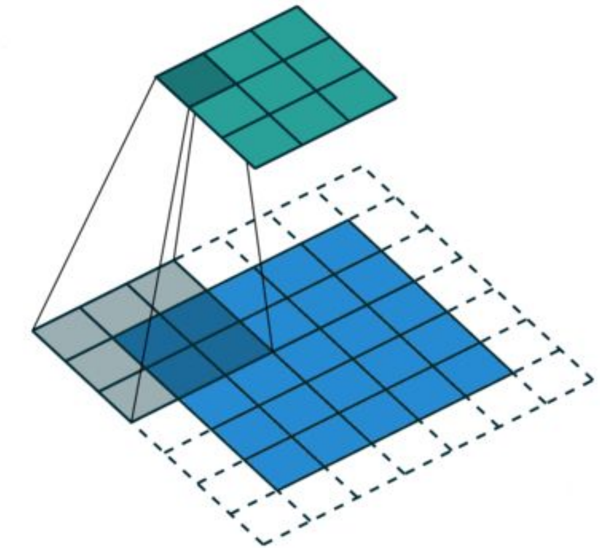


Image credits: Vdumoulin; published under MIT licence; https://github.com/vdumoulin/conv_arithmetic

Question: What is the result of using dilation?

2D convolution with *dilation*

- In neural networks

Dilation

Input: blue

Output: cyan

Filter: vertical lines

Dilation: 1

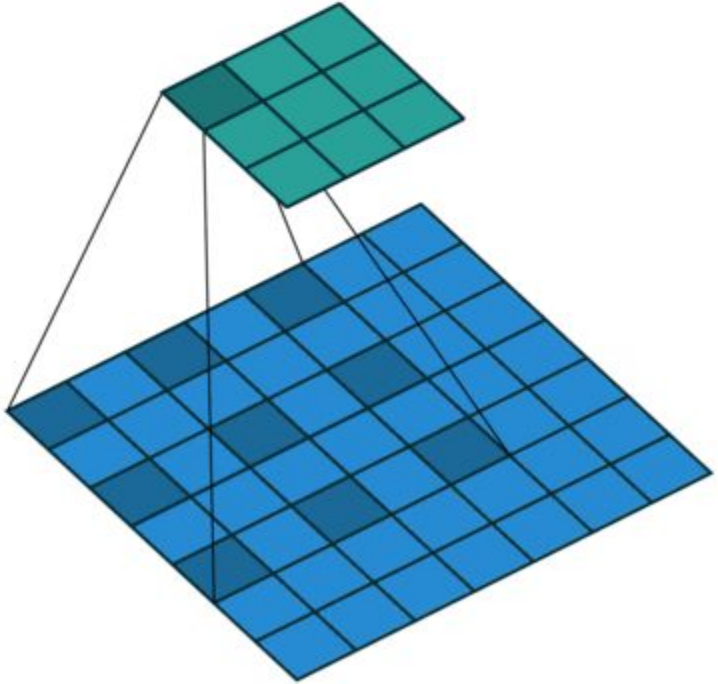


Image credits: Vdumoulin; published under MIT licence; https://github.com/vdumoulin/conv_arithmetic

Convolutional Neural Network

- Architecture

Convolutional Neural Networks

Similar to "normal" NNs (MLPs, week 2)

- Feed-forward networks
- Made up of neurons & learnable (filter) weights
- Weights are learned by backpropagation

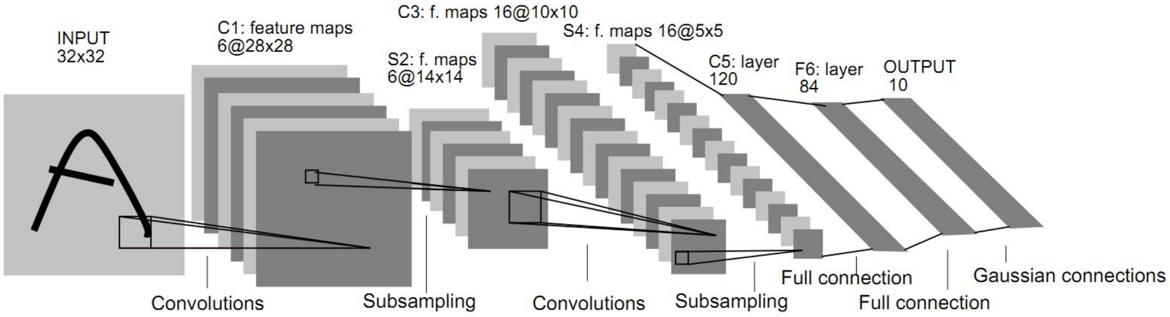


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Image credits: LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Proceedings of the IEEE

CNN

- Architecture motivation

Question: What does “fully connected” mean?
Question: What is is another term for “fully connected”?

Standard NNs (fully connected) don’t scale well to normal size images

- E.g. with a $500 \times 500 \times 3$ pixel image, *a single neuron* in the first hidden layer would have $750{,}000$ *weights*
- The huge number of weights leads to overfitting & high computational requirements

Like a standard NN, a CNN is made up of layers, but:

Hidden layers in CNNs have neurons arranged in 3D cubes

- Width x Height x Depth

Neurons in the same layer & depth *share the same filter* (i.e. weight sharing)

- E.g. for a $3 \times 3 \times 3$ filter, all neurons in a given layer & depth have 28 parameters ($3 \times 3 \times 3{+}1$)

CNN

Standard layer types

Convolutional layer

- The same filter is input to several neurons, i.e. they ***share*** weights
- Computes the output of neurons connected to local regions in the input
- Each neuron computes a dot product between weights & a small region in the input volume
- Often ReLU activation
- *Trainable*

Pooling layer

- Performs downsampling along the spatial dimensions (width & height)
- *Not trainable*

Fully connected layer

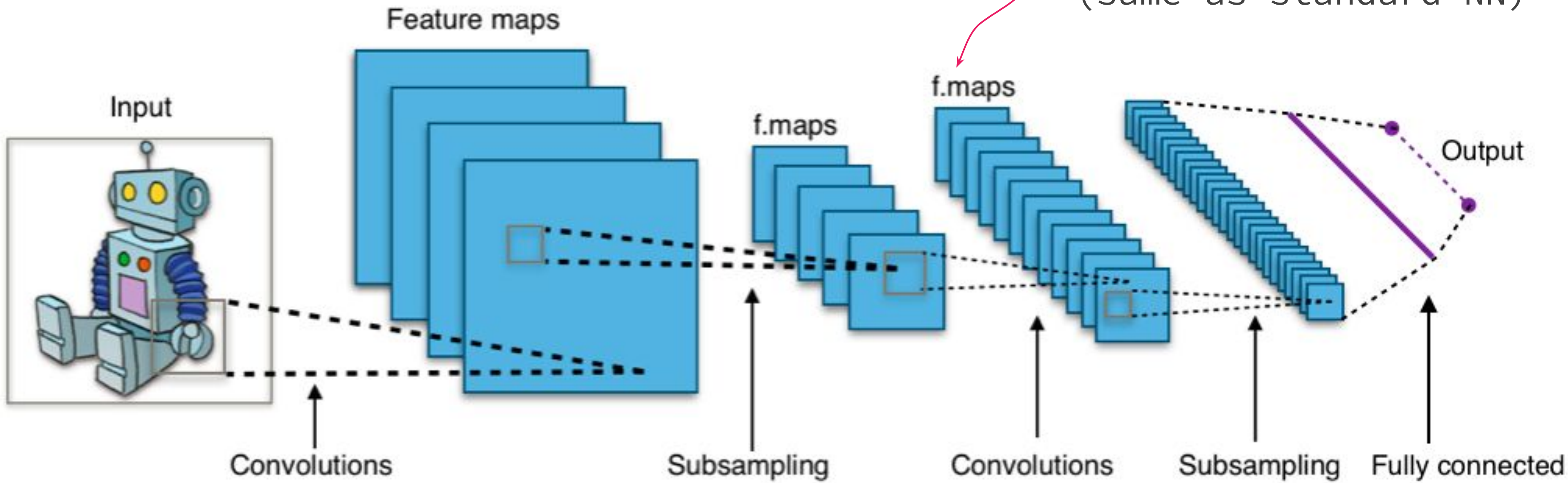
- Computes class scores
- Often softmax activation (if multi-class)
- *Trainable*

CNN

- Types of layers

Question: What does “f.maps” mean?

Convolutional layer **Pooling layer** **Fully connected layer** (same as standard NN)



The convolutional layer

A convolutional layer with 5 filters

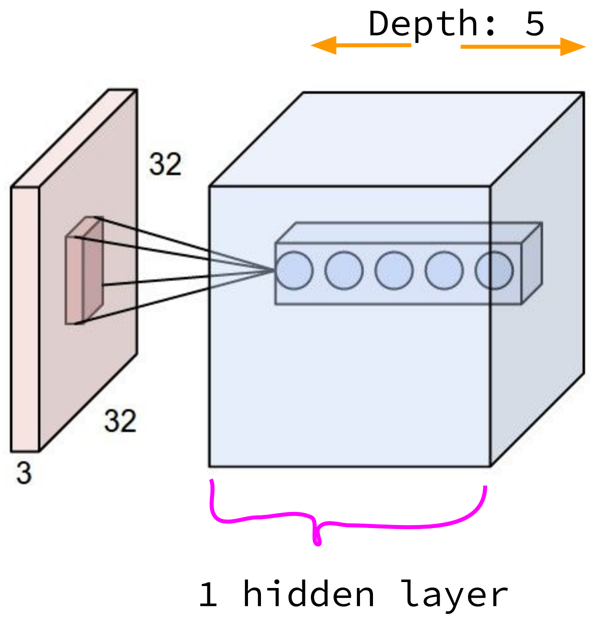
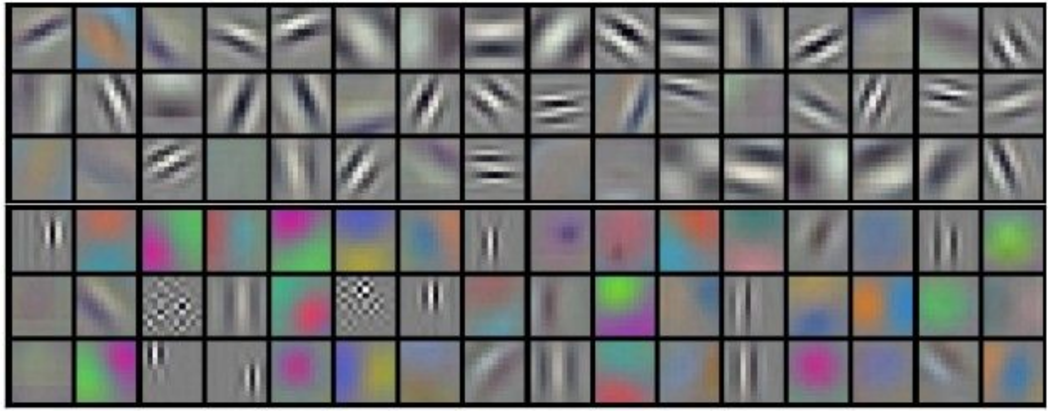


Image credits:
Conv layer by Aphex34; CC BY-SA 4.0, https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Conv_layer.png
Filers by Krizhevsky, Sutskever & Hinton (2012)

Examples of filter weights from AlexNet (1^{st} hidden layer)

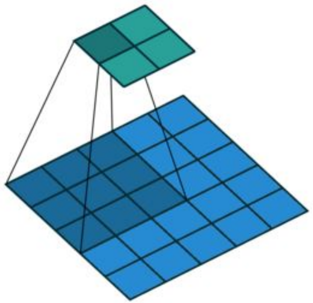


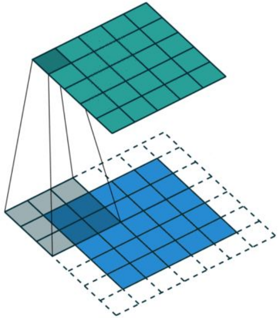
The convolutional layer

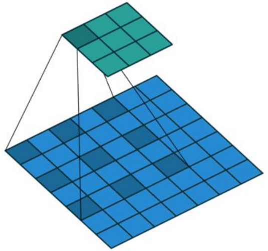
The number of filters in a layer is set by the depth.

The number of neurons in a layer depends on:

- *Depth (depth == # filters)*
    - Greater depth -> more neurons
- *Stride*
    - Longer stride (e.g. 2) -> less neurons
- *Padding*
    - More padding -> more neurons
- *Dilation*
    - More dilation -> less neurons







Question: What is gained by decreasing the number of parameters?

The convolutional layer

Parameter/weight sharing

Example:

Without weight sharing

- Assume $55 \times 55 \times 96 = 290{,}400$ neurons in the first convolutional layer & each has $11 \times 11 \times 3 = 363$ weights & 1 bias (locally **NOT** *fully* connected, similar to slide 9).
  - $290{,}400 \times 364 = \mathbf{105{,}705{,}600}$ **parameters** in the first layer alone.

With weight sharing

- As above, but every neuron at the same depth use the same weights.
  - 96 unique *sets* of parameters, i.e. $364 \times 96 = \mathbf{34{,}848}$ **parameters**.

Improvement by a factor of ~3,000!

Pooling layer

Reduces the spatial size (width, height) of the hidden layers.

→ less parameters & computation necessary

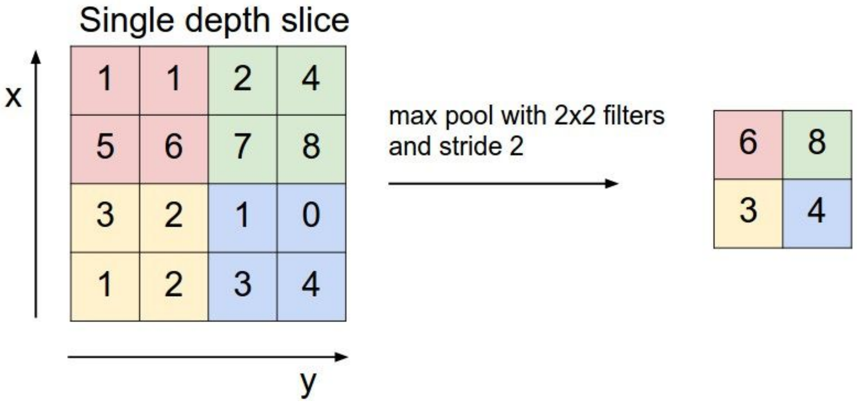
→ controls overfitting

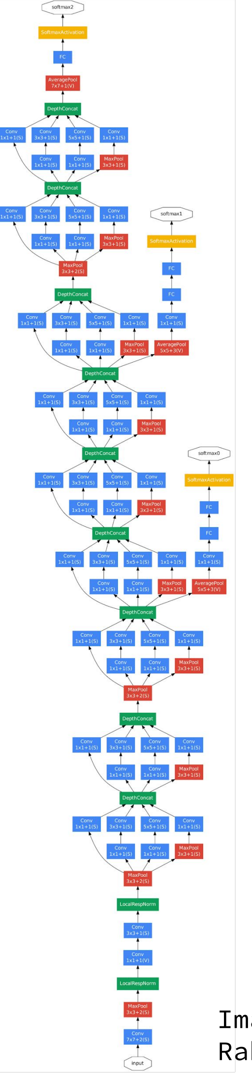
Operates independently on every depth slice using the MAX or AVERAGE operation.

Most common is pooling layer with filters of size 2×2 & a stride of 2.

→ downsamples every depth slice by a factor 2 along both width & height, discarding 75% of the activations.

Example of 2×2 pooling with a stride of 2



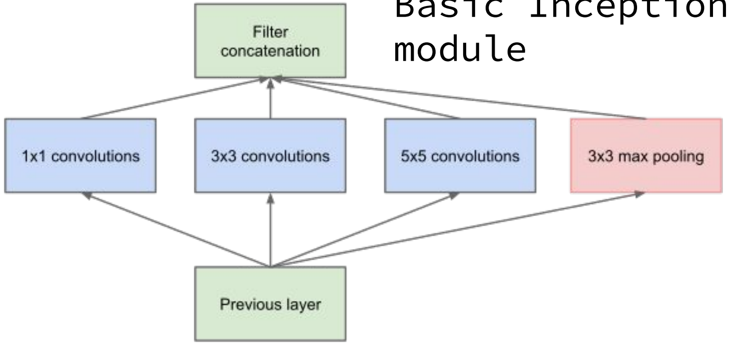


Fancy CNN architectures

Inception network (v1, 2014)

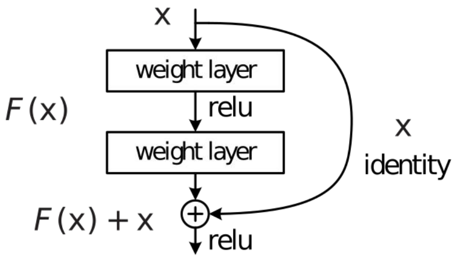
- Different filter sizes for different size objects

Basic Inception module



ResNet (2015)

- Skip connections
- No fully connected layer
- Batch normalization



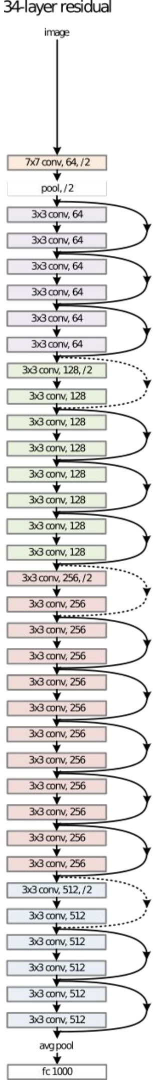
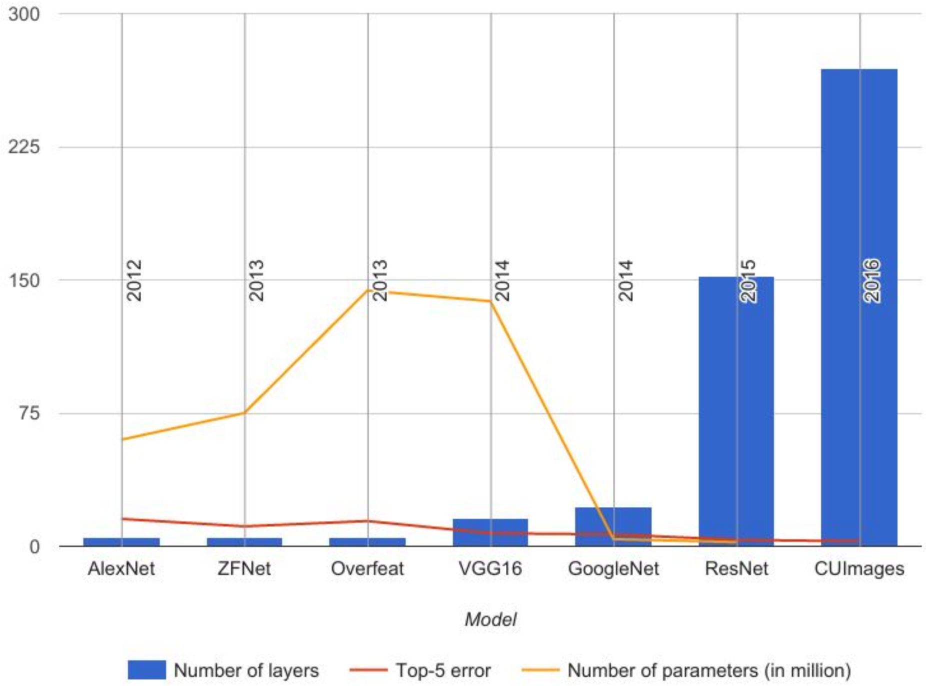


Image credits: Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A (2014) Arxiv; He K, Zhang X, Ren S, Sun J (2015) Arxiv

Why so deep?

Question: What does the *y*-axis show?



Increasing depth of the ImageNet winners

Why so deep?

- Stacking small filters

  → increasingly large receptive fields

  → filters of increasing complexity

- Large ($width \times depth$) is good (high capacity), but depth scales better than width (to some extent)

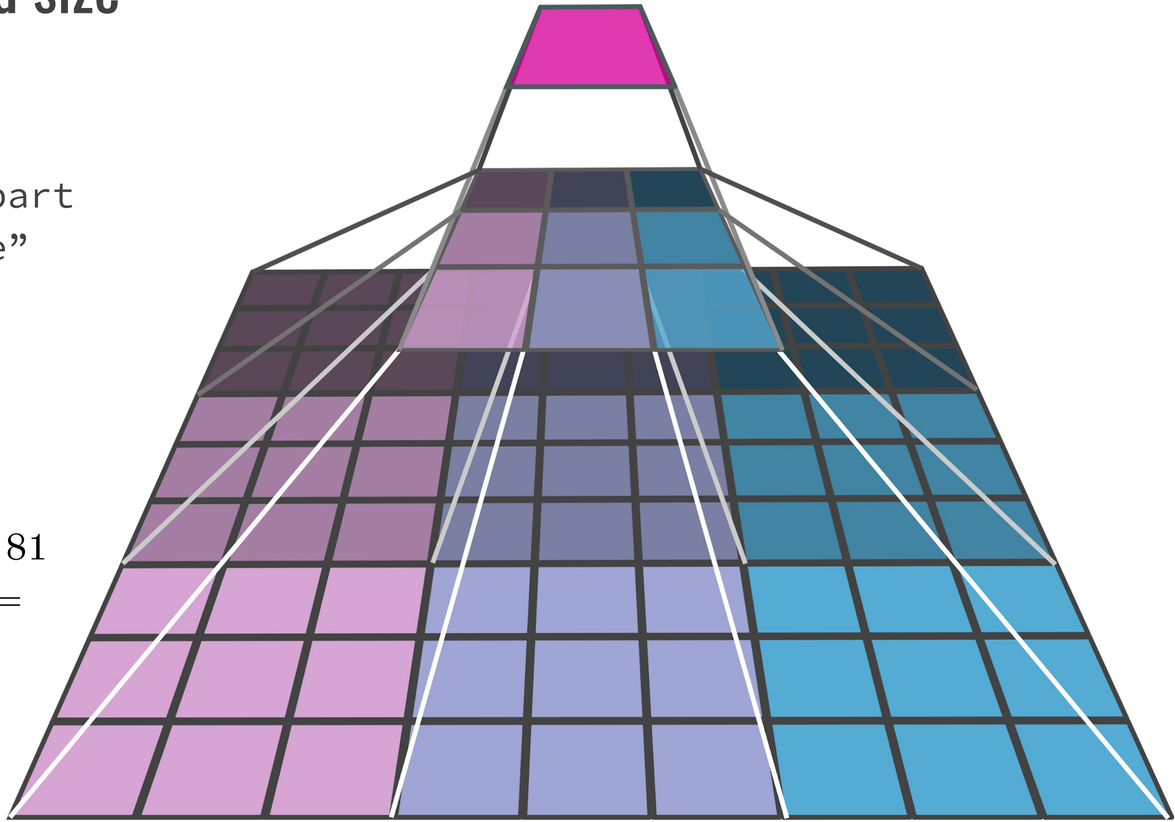
Why so deep?

- Increasing receptive field size

- *Receptive field* (rf): the part of the input space “visible” to a neuron

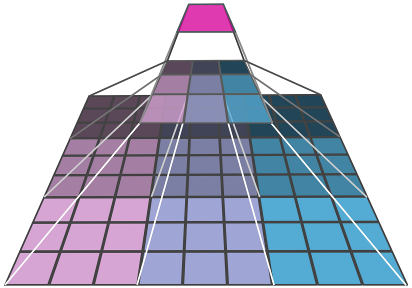
Example 3x3 filter:

- 1st layer rf size: $\mathbf{3}\times\mathbf{3} = 9$
- 2nd layer: $3\times3\times3\times3 = \mathbf{9}\times\mathbf{9} = 81$
- 3rd layer (not shown): $3^{3\times2} = \mathbf{27}\times\mathbf{27} = 729$



Why so deep?

- Increasing receptive field size



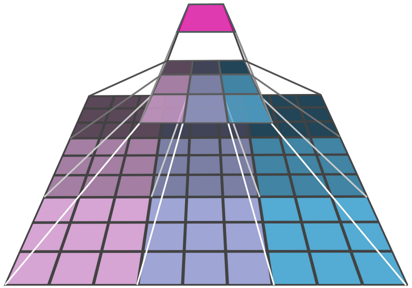
Example face detection:

3×3 filters & 720×960 photo

Assume that some neuron needs an rf big enough to cover the entire face.

Question: What species of snake?

Why so deep?
- Increasing receptive field size



Example face detection:

3×3 filters & 720×960 photo

Assume that some neuron needs an rf big enough to cover the entire face

- Rf size approx. 40% of 720 → 288×288 px
- Max rf with 5 layers: $3^{5\times2} = 243\times243$ px
- Max rf with 6 layers: $3^{6\times2} = 729\times729$ px

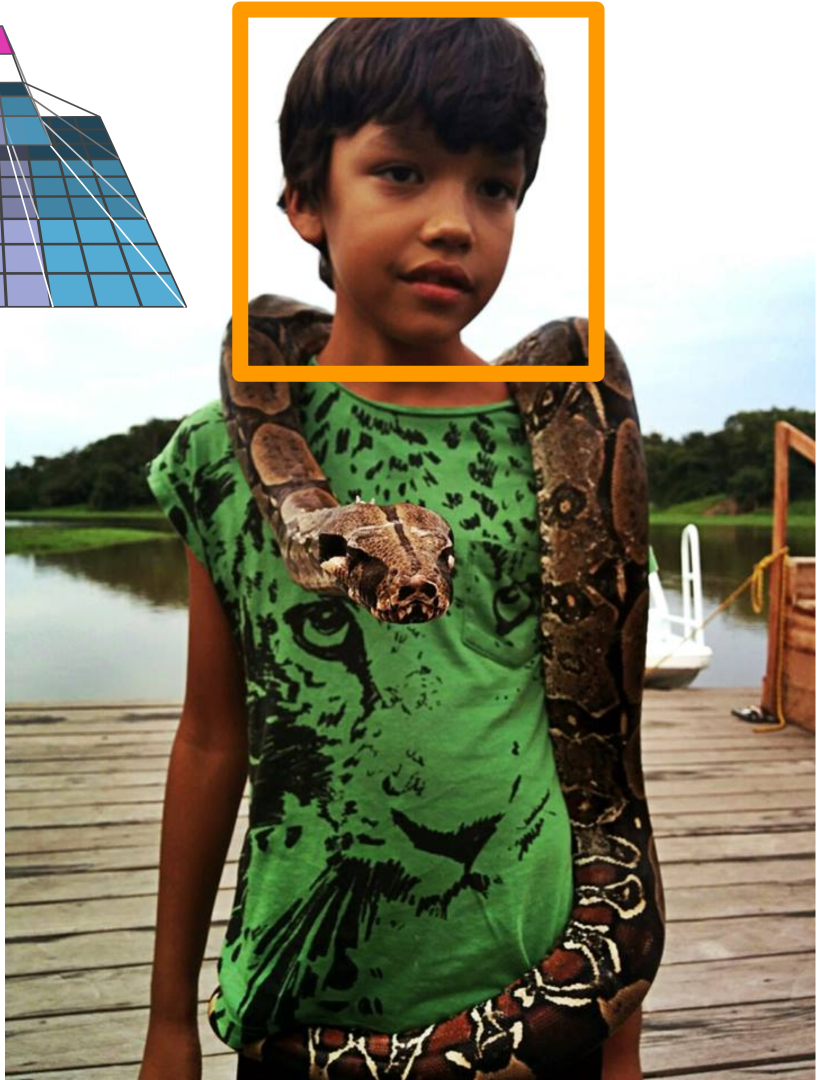


Image credits: Lee H, Grosse R, Ranganath R, Ng A (2011) ACM

Why so deep? - stacking results in increasingly complex filters

The problem (example): classify a 1000×1000 px grey-scale image into 2 classes, where each pixel can take one of 256 values

→ $\mathbf{256^{1,000,000}}$ **possible images**

→ I.e. an arbitrary function would be defined by a list of $\boldsymbol{256^{1,000,0000}}$ ***probabilities****

*compare to about 10^{78} atoms in our universe

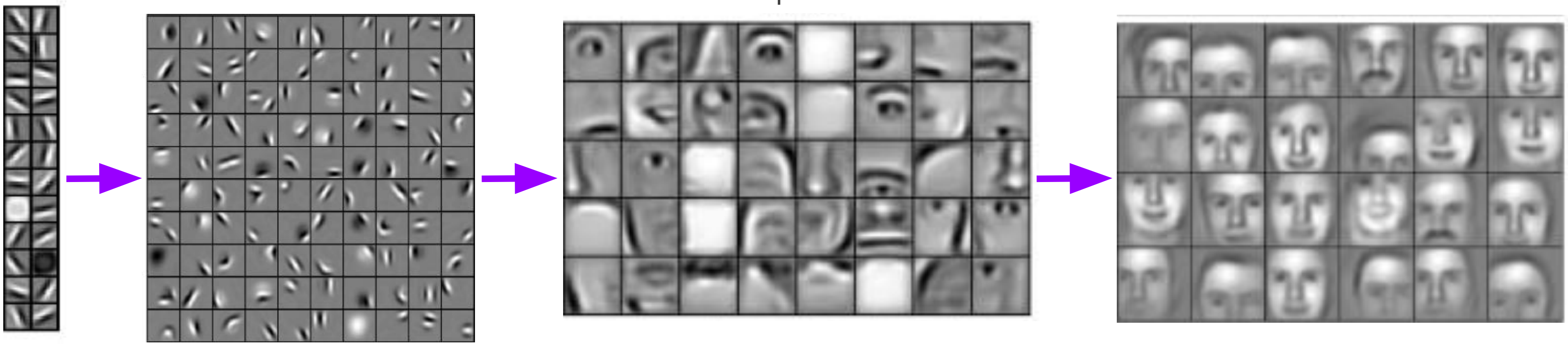


Image credits: Lee H, Grosse R, Ranganath R, Ng A (2011) ACM

Why so deep? - stacking results in increasingly complex filters

The problem (example): classify a 1000×1000 px grey-scale image into 2 classes, where each pixel can take one of 256 values

How can a list of $\mathbf{256^{1,00,0000}}$ ***probabilities*** *be learned with only millions of parameters?*

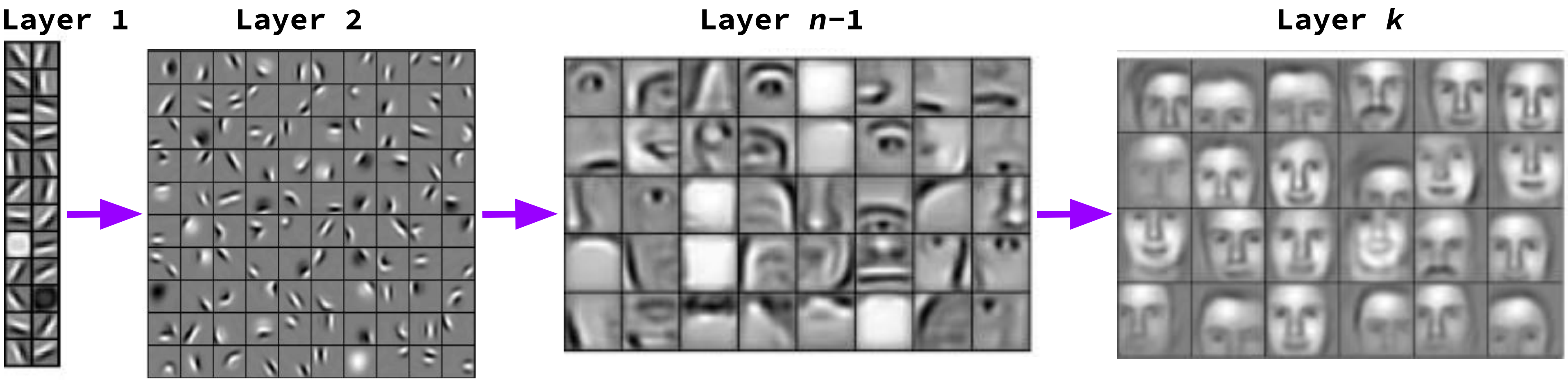


Image credits: Lee H, Grosse R, Ranganath R, Ng A (2011) ACM

Why so deep? - stacking results in increasingly complex filters

The physical world has a hierarchical structure

For example spatially, an **object hierarchy**: elementary particles → atoms → molecules → cells → organisms → planets → solar systems → galaxies → ...

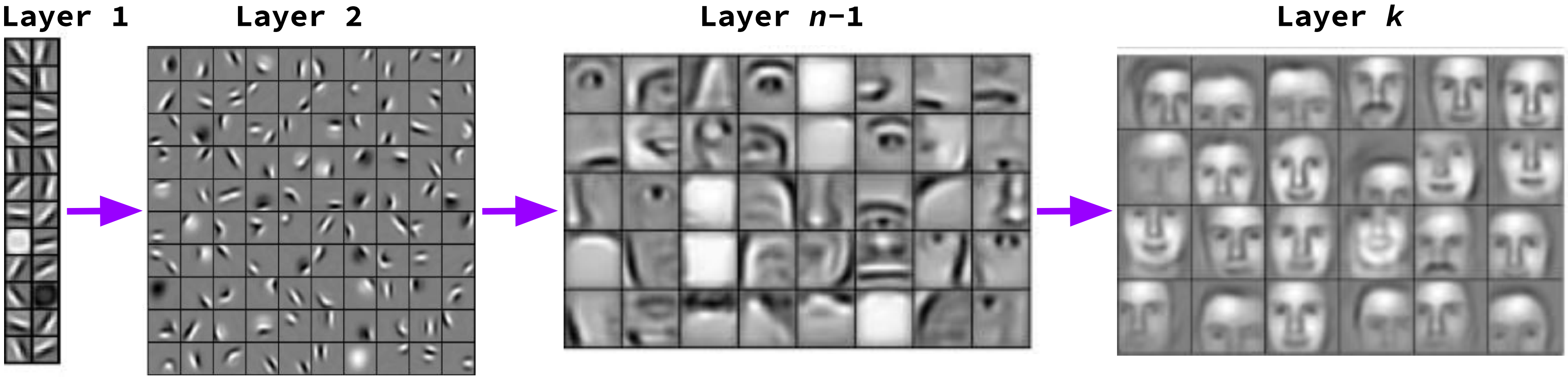


Image credits: Lee H, Grosse R, Ranganath R, Ng A (2011) ACM

Why so deep? - stacking results in increasingly complex filters

The physical world has a hierarchical structure

Instead of learning an **arbitrary function**, learn the **generative function**.

I.e. **hierarchical composition of simple features** (e.g. edges) into increasingly complex features (eyes) & objects (faces).

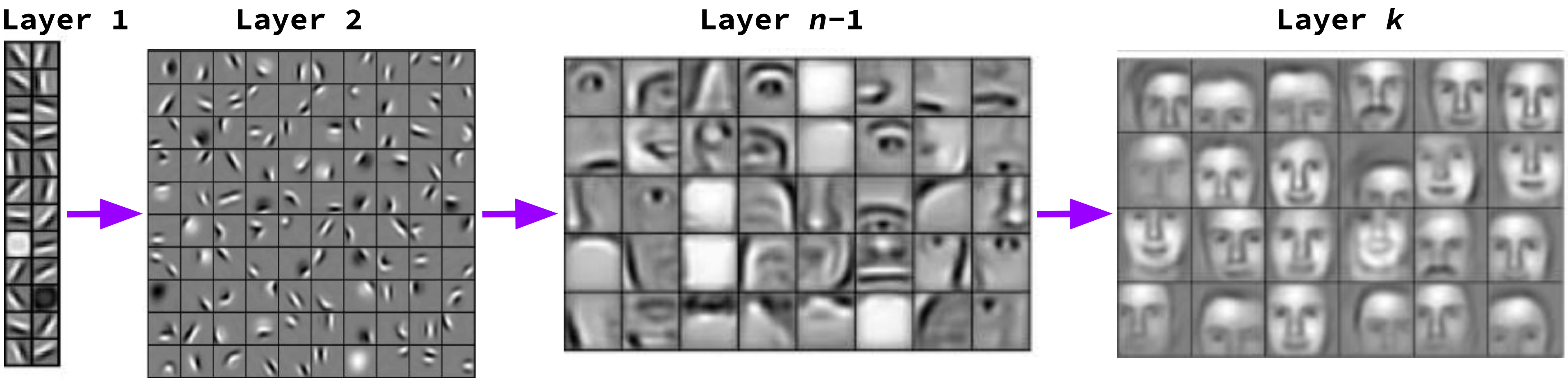
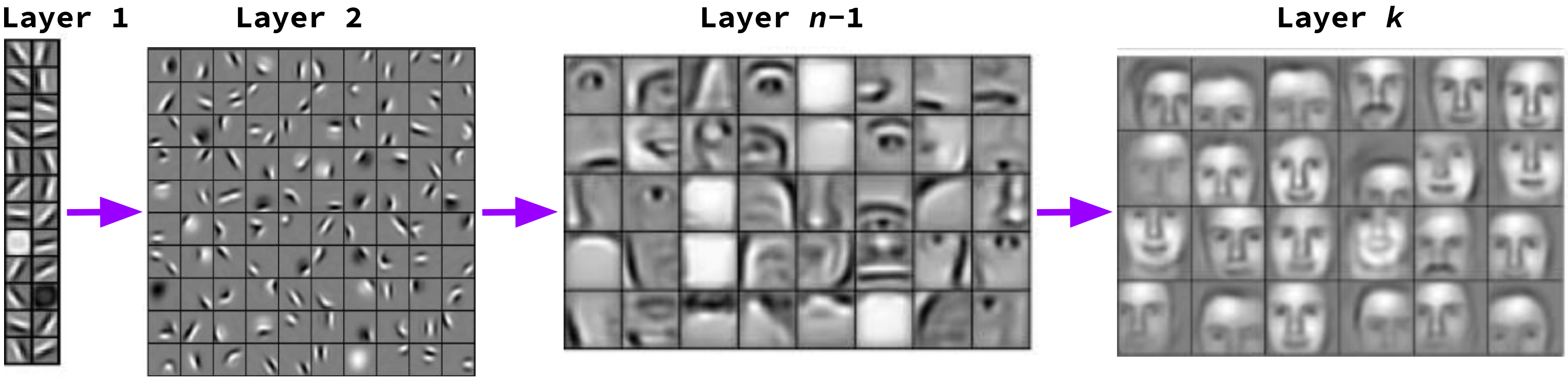


Image credits: Lee H, Grosse R, Ranganath R, Ng A (2011) ACM

Why so deep? - stacking results in increasingly complex filters

Multi-layer CNNs provide an architecture for learning this type of hierarchical composition of simple functions.

See “*Why does deep & cheap learning work so well?*” By Henry W. Lin and Max Tegmark (2016) https://arxiv.org/abs/1608.08225



Why so deep? - depth scales better than width

A k-1 layer architecture might require an exponential number of elements compared to a k layer architecture [according to theoretical proofs from circuit design].

→ deep & narrow can approximate a complicated function with **fewer parameters** than a shallow & wide architecture

For more see:

- Chapter 5 in "*Neural Networks & Deep Learning*" by Michael Nielsen (2019) http://neuralnetworksanddeeplearning.com/chap5.html
- "*Learning Deep Architectures for AI*" by Yoshua Bengio (2009) https://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf

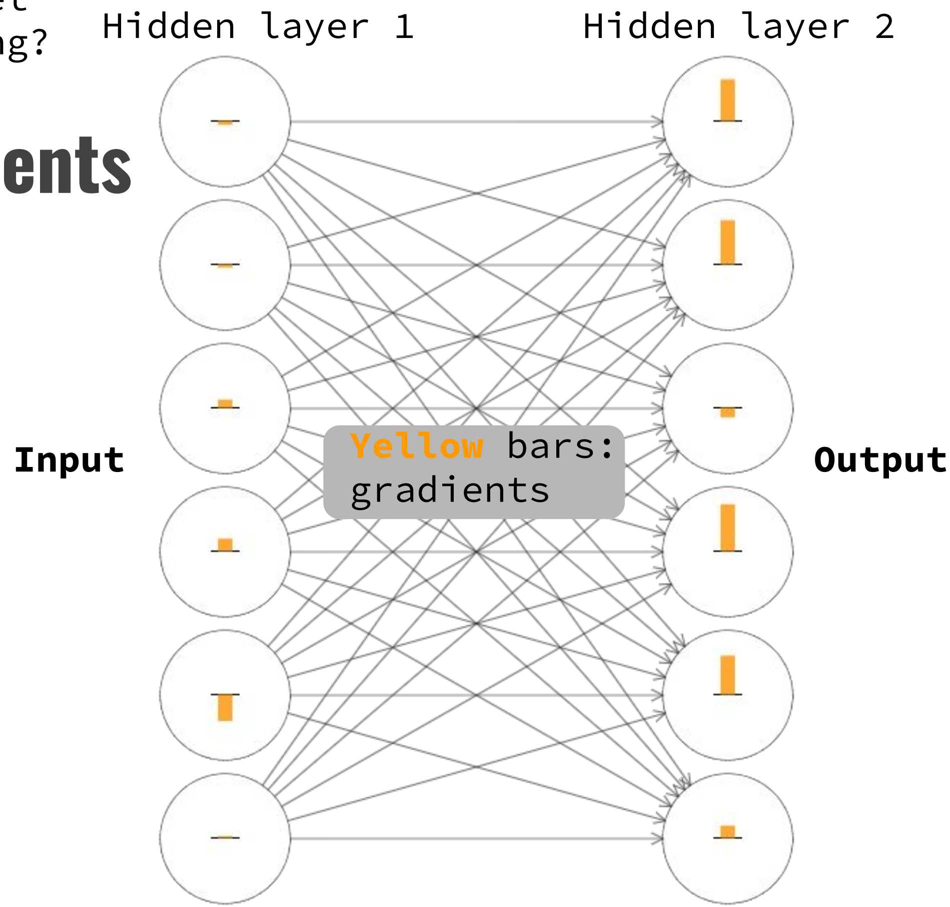
Vanishing Gradients

- The problem with depth

Question: What layer will (on average) get the smallest weight changes while learning?

Vanishing & exploding gradients

The layer closest to the output tends to learn fastest (i.e. biggest gradients).



Question: What layer is closest to the output: Hidden layer 1 or 4?

Vanishing & exploding gradients

The layer closest to the output tends to learn fastest (i.e. biggest gradients).

||A||: size of A

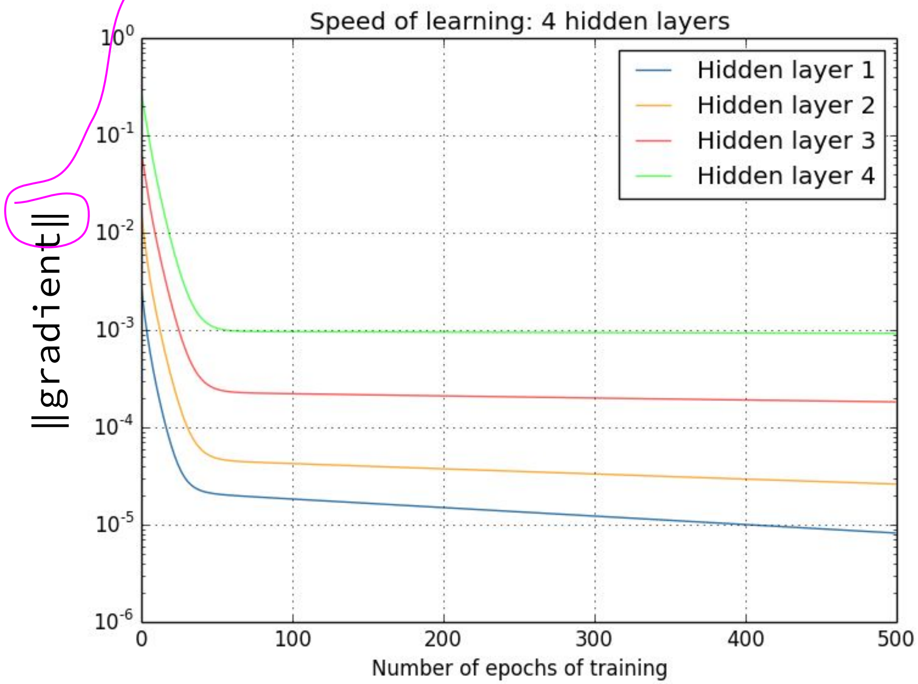
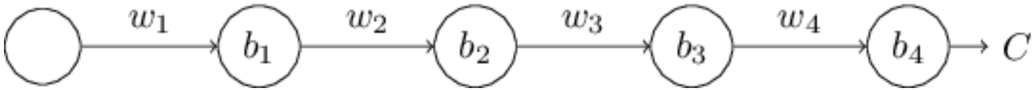


Image credit: Michael A. Nielsen, "*Neural Networks and Deep Learning*", Determination Press, 2015; licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License.

Vanishing & exploding gradients
- example

Consider a super simple example:

- 4 layers deep, 1 neuron wide & sigmoid activation functions.



The layer closest to the output tends to learn fastest (i.e. biggest gradients).

WHY?

Vanishing & exploding gradients
- example

The cause of vanishing gradients

- A long series of multiplication with values close to 0.

Backpropagation of the cost

- Cost: C
- Sigmoid activation: σ
- Derivative of activation func: σ'

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

Image credit: Michael A. Nielsen, "*Neural Networks & Deep Learning*", Determination Press, 2015; licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License.

Vanishing & exploding gradients - example

Derivative of sigmoid:

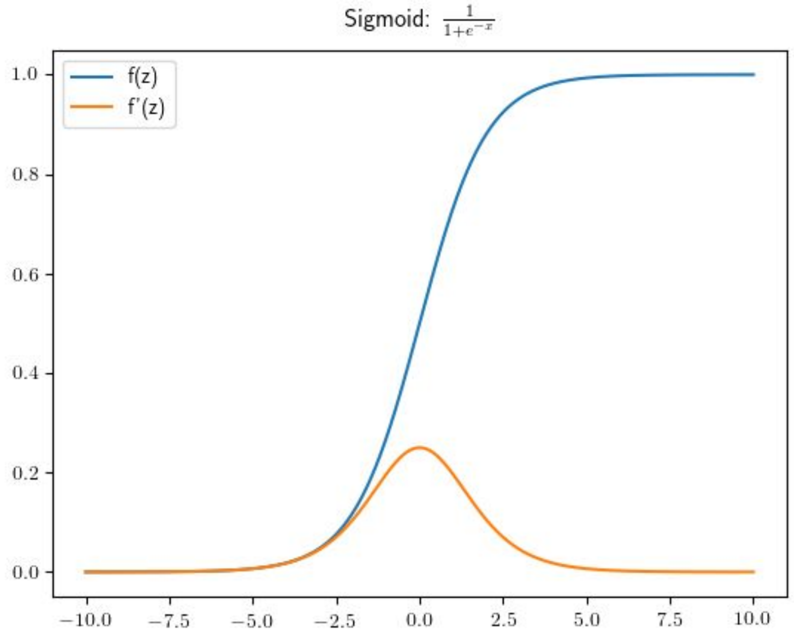
- Max 0.25

Weight initialization:

- Normal distribution (mean: 0 & standard deviation: 1)
- I.e. weights tends to be < 1.

Approximate effect on gradients in the 1st layer:

- $0.25^4 = 0.0039$

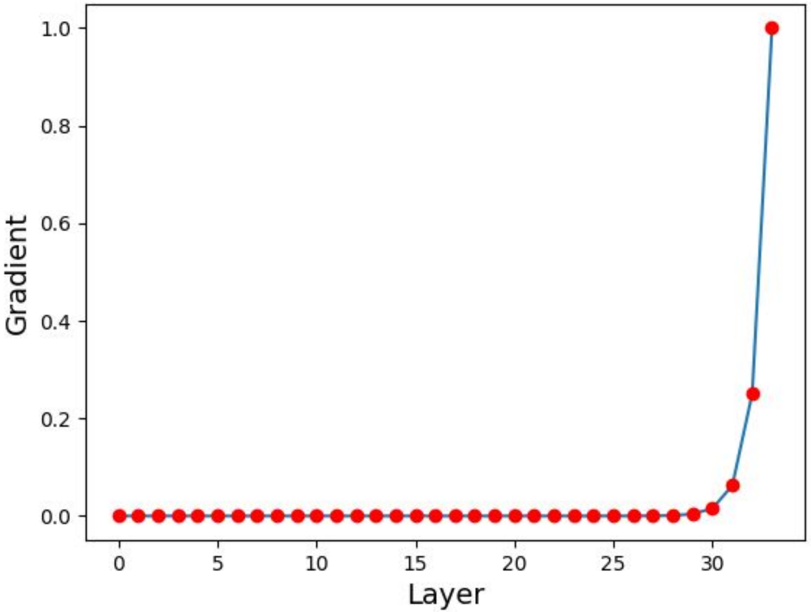


$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

Vanishing & exploding gradients

Gradients can be exponentially decreasing with distance from output layer.

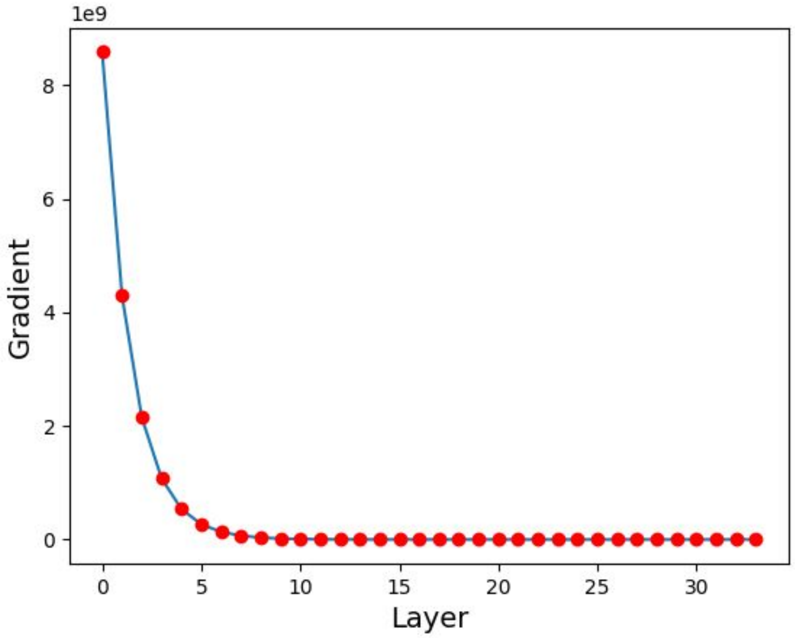
No gradient → No learning



Vanishing & *exploding* gradients

Exploding gradients:

Repeated multiplication of gradients greater than 1.



Vanishing & exploding gradients

- Mitigating the issues

Question: What is the derivative of ReLu?

Exploding gradients:

- Clip the gradient → maintains the direction of the gradient.

Vanishing gradients:

- ReLu activation
- Careful weight initialization (check Keras Initializers: https://keras.io/initializers/)
- Skip-connections (e.g. ResNet & Transformer)
- Auxiliary classifiers (e.g. Inception network)

Vanishing gradients in Recurrent Neural Networks (RNNs):

- Gating (lecture 9)

CNN – practice

Don't be a hero

Use whatever works best on ImageNet

In the vast majority of applications you don't have to develop your own architecture or *train it from scratch*.

Instead:

1. Find whatever architecture currently works best on ImageNet.
2. Download a pretrained model.
3. Finetune it on your data.

You should rarely ever have to design or train a CNN from scratch.

Question: What does *train from scratch* mean?

Transfer learning

Build good models quick by **standing on the shoulder of giants!**

- Use pretrained models, i.e. models trained on a big benchmark dataset (e.g. ImageNet).
- Not only for CNNs & images.

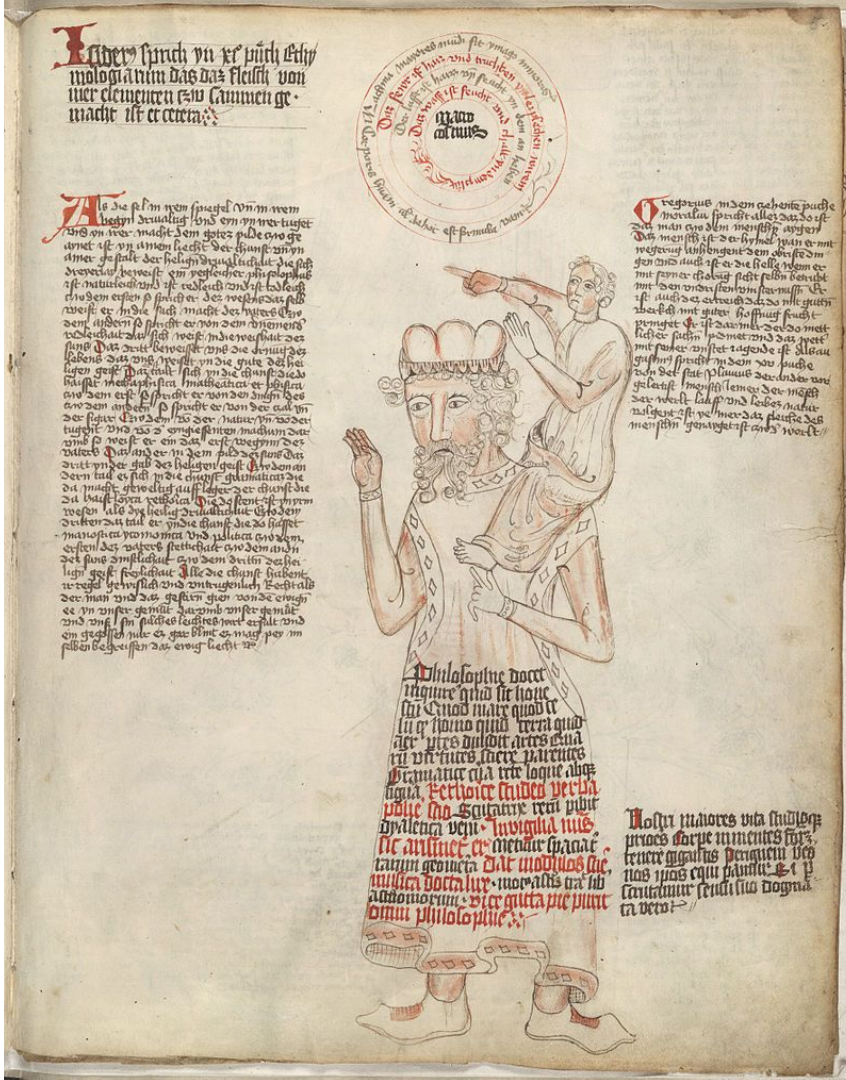
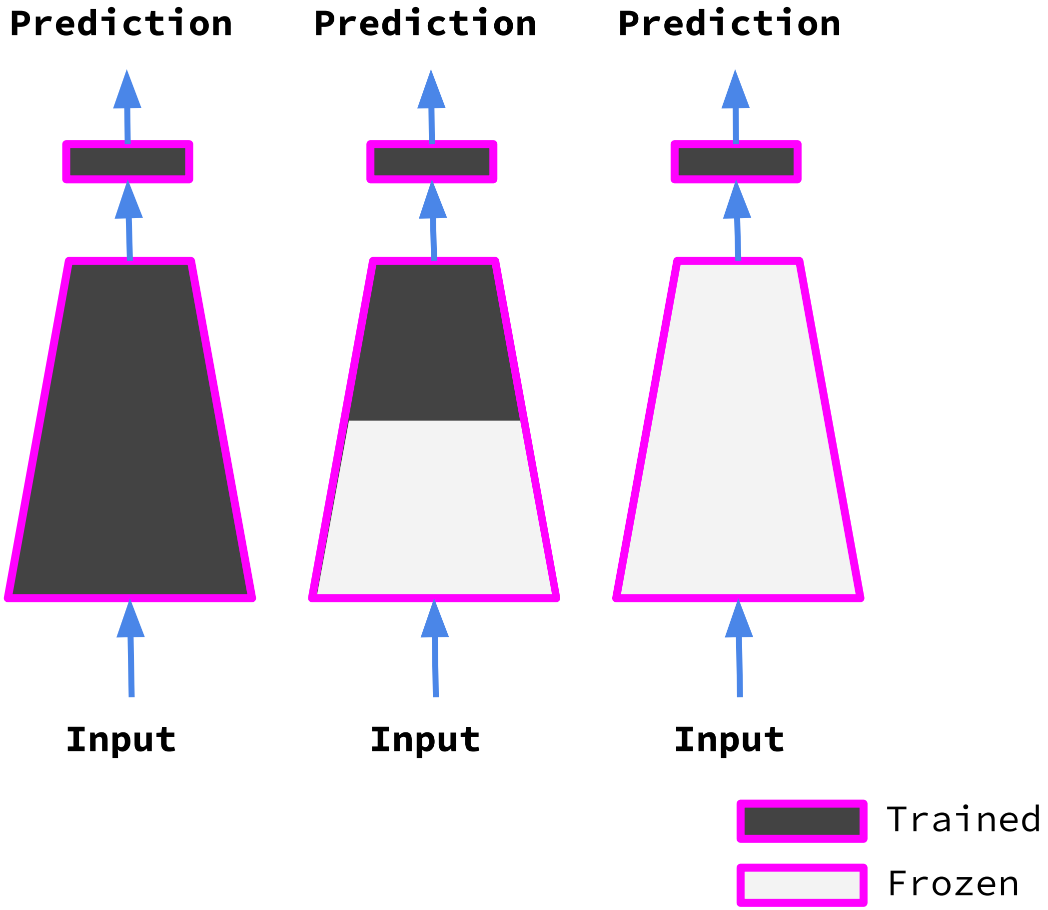


Image credits: Library of Congress, Rosenwald 4, Bl. 5r; Public domain. The picture is derived from Greek mythology: the blind giant Orion carried his servant Cedalion on his shoulders to act as the giant's eyes.

Transfer learning

- strategies

1. **Train the entire model**
    - Careful with the learning rate
2. **Train some layers & leave the rest frozen**
    - Careful with the learning rate
3. **Freeze the convolutional base**



Transfer learning
- Pre-trained models

Pre-trained models for Keras:

https://keras.io/applications/

» Keras API reference / Keras Applications

Keras Applications

Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

Weights are downloaded automatically when instantiating a model. They are stored at `~/.keras/models/`.

Upon instantiation, the models will be built according to the image data format set in your Keras configuration file at `~/.keras/keras.json`. For instance, if you have set `image_data_format=channels_last`, then any model loaded from this repository will get built according to the TensorFlow data format convention, "Height-Width-Depth".

Available models

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4

Keras Applications
- Available models
- Usage examples for image classification models
  - Classify ImageNet classes with ResNet50
  - Extract features with VGG16
  - Extract features from an arbitrary intermediate layer with VGG19
  - Fine-tune InceptionV3 on a new set of classes
  - Build InceptionV3 over a custom input tensor

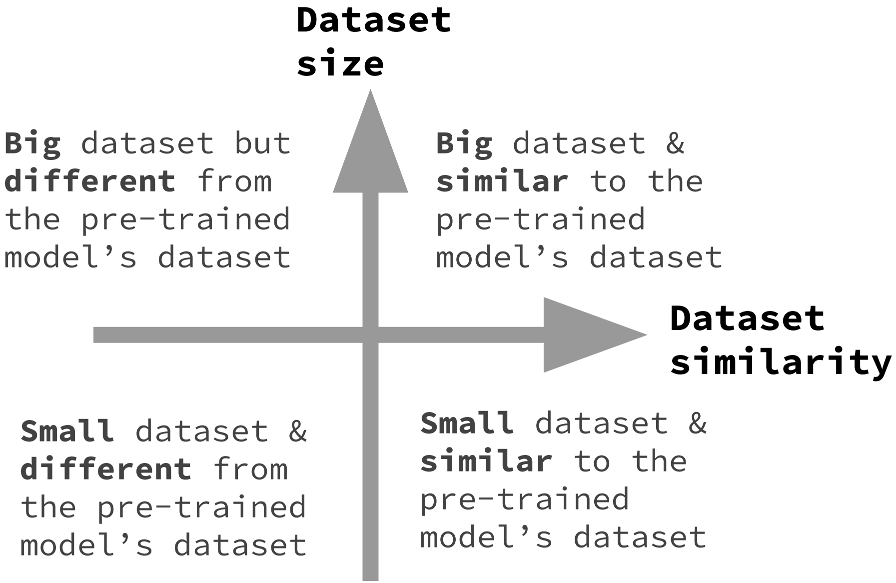
Usage examples for image classification models

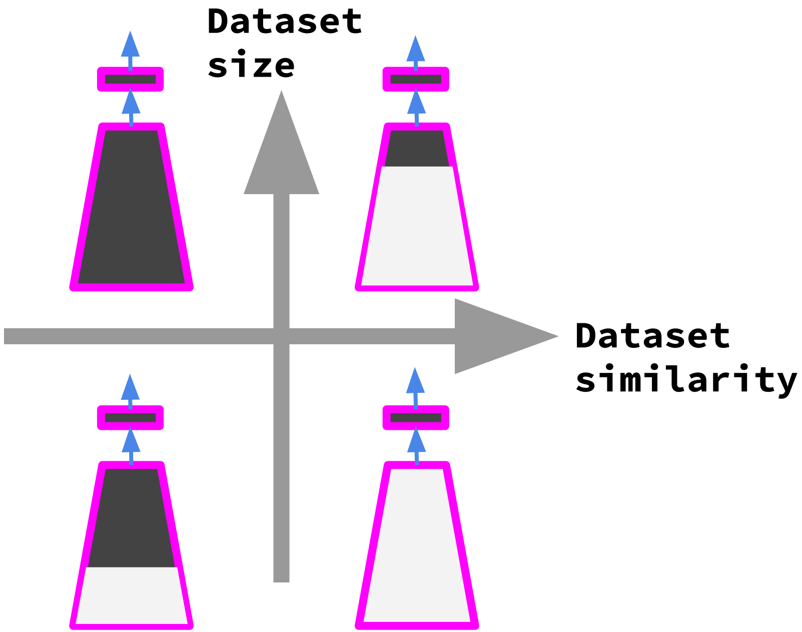
Classify ImageNet classes with ResNet50

```
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np

model = ResNet50(weights='imagenet')

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265', u't
```

Transfer learning

- When to do what





Tutorial: transfer learning

Upload the notebook MMAI5500_class04_transfer.ipynb & data.zip to Google Colab.