

Deep Learning

TSM-DeLearn

9. Deep Architectures

Jean Hennebert
Martin Melchior

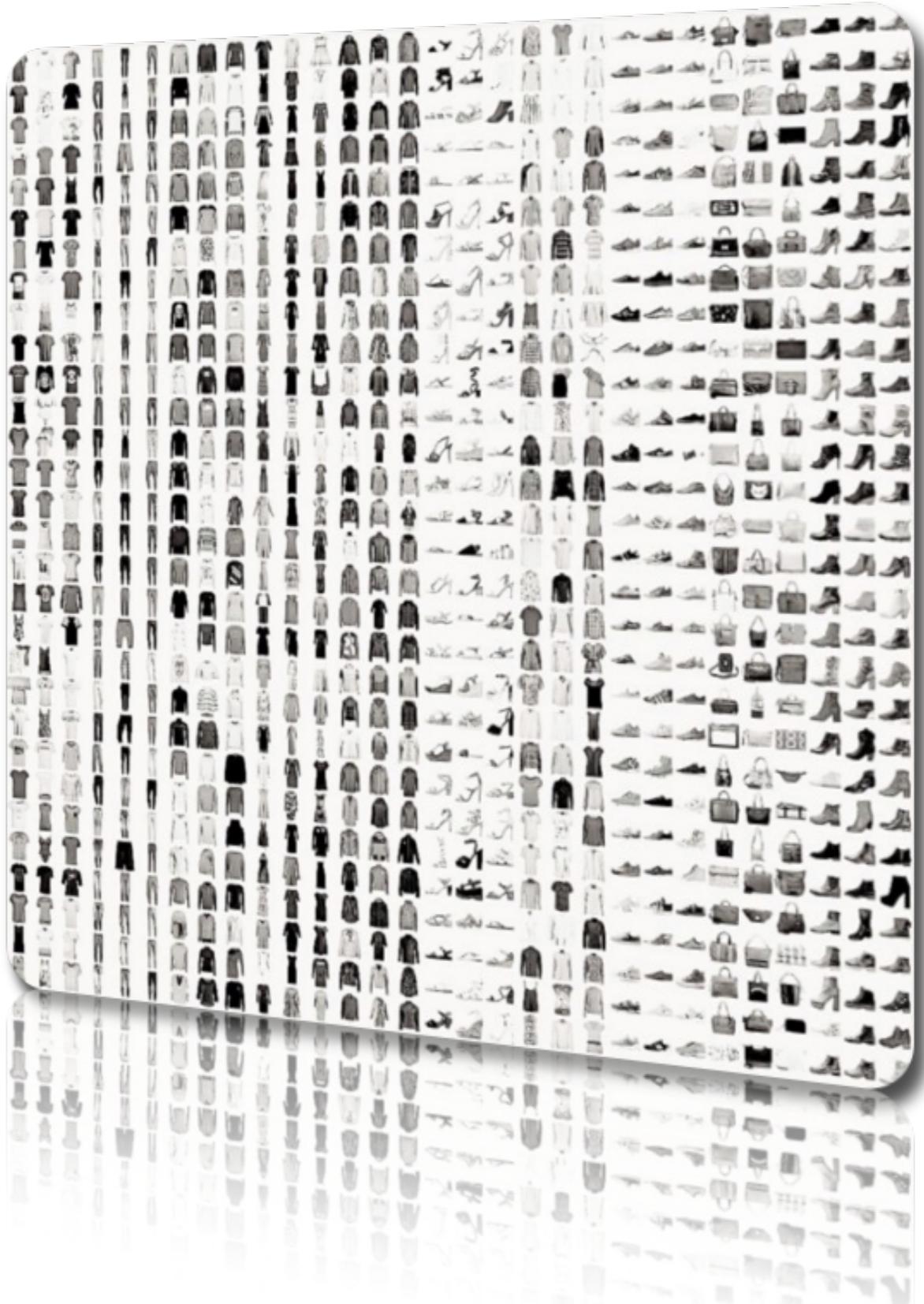


Plan

1. CNN recaps - example on Fashion MNIST
2. CNN - hierarchical features / visualisation inside network
3. Data augmentation
4. Deep CNN Architectures
5. Exam question examples
6. PW

CNN recaps

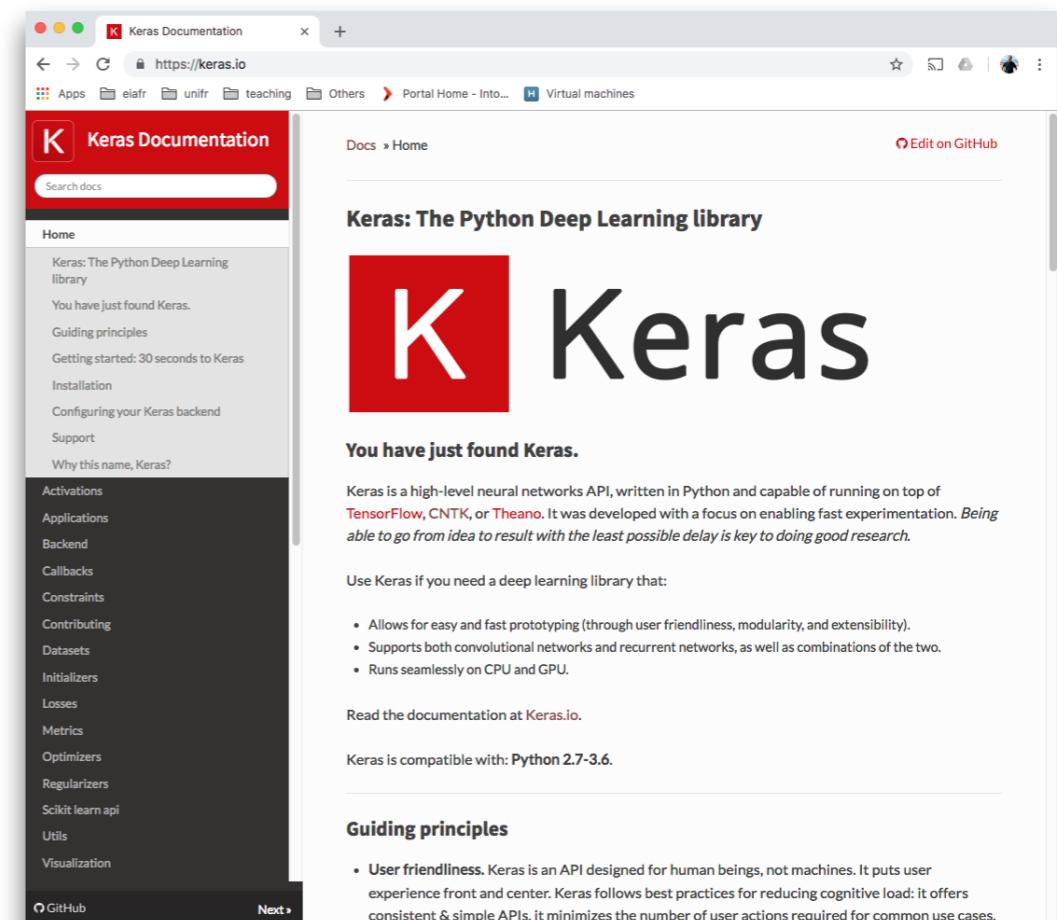
Recaps through
FashionMNIST
CIFAR10 leaderboard
CIFAR10 online demo



RECAP 1 - Keras - what is it?

Keras is a high-level open-source neural networks API, written in Python and capable of running on top of **TensorFlow**, **CNTK**, or **Theano**. It was developed with a focus on enabling fast experimentation.

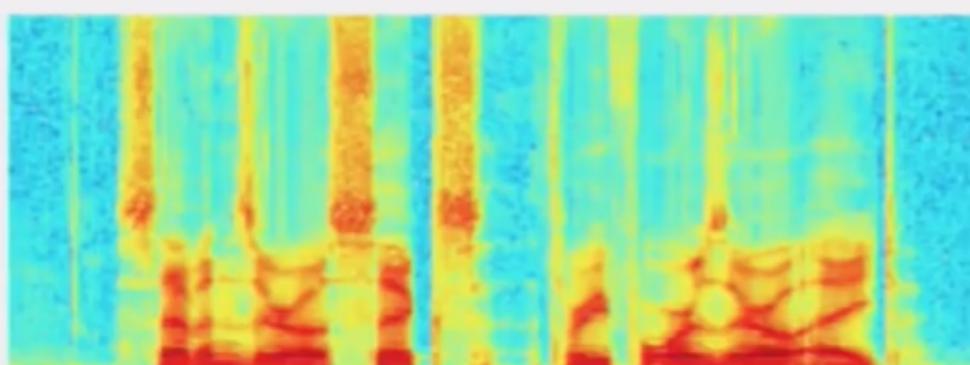
- Designed to be minimalistic & straight forward yet extensive
- “Deep” enough to build serious models
 - Support Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), combination of both.
- Wrapper on top of TF / CNTK / Theano backend
 - But not only a wrapper



Source: <https://keras.io>

RECAP 2 - Convolution and spatial structure

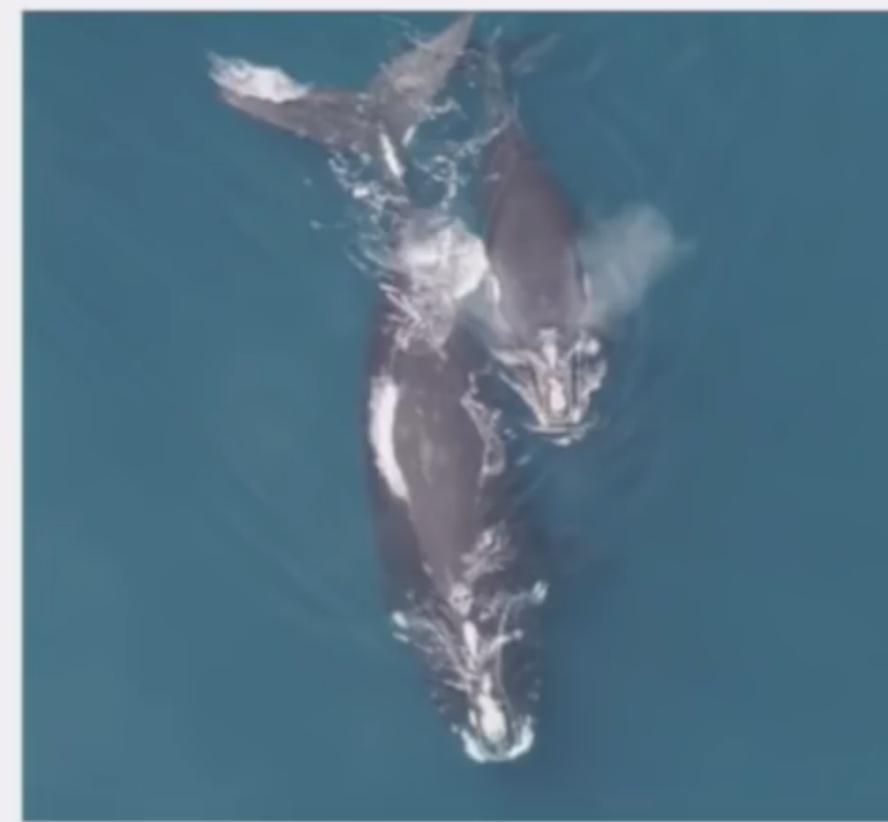
- In most real-world applications input vectors have a spatial **structure**, a “local connectivity” that you would like to model
- **Convolution layers** have the property to preserve the spatial structure and discover the local connectivity



Spectrograms

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

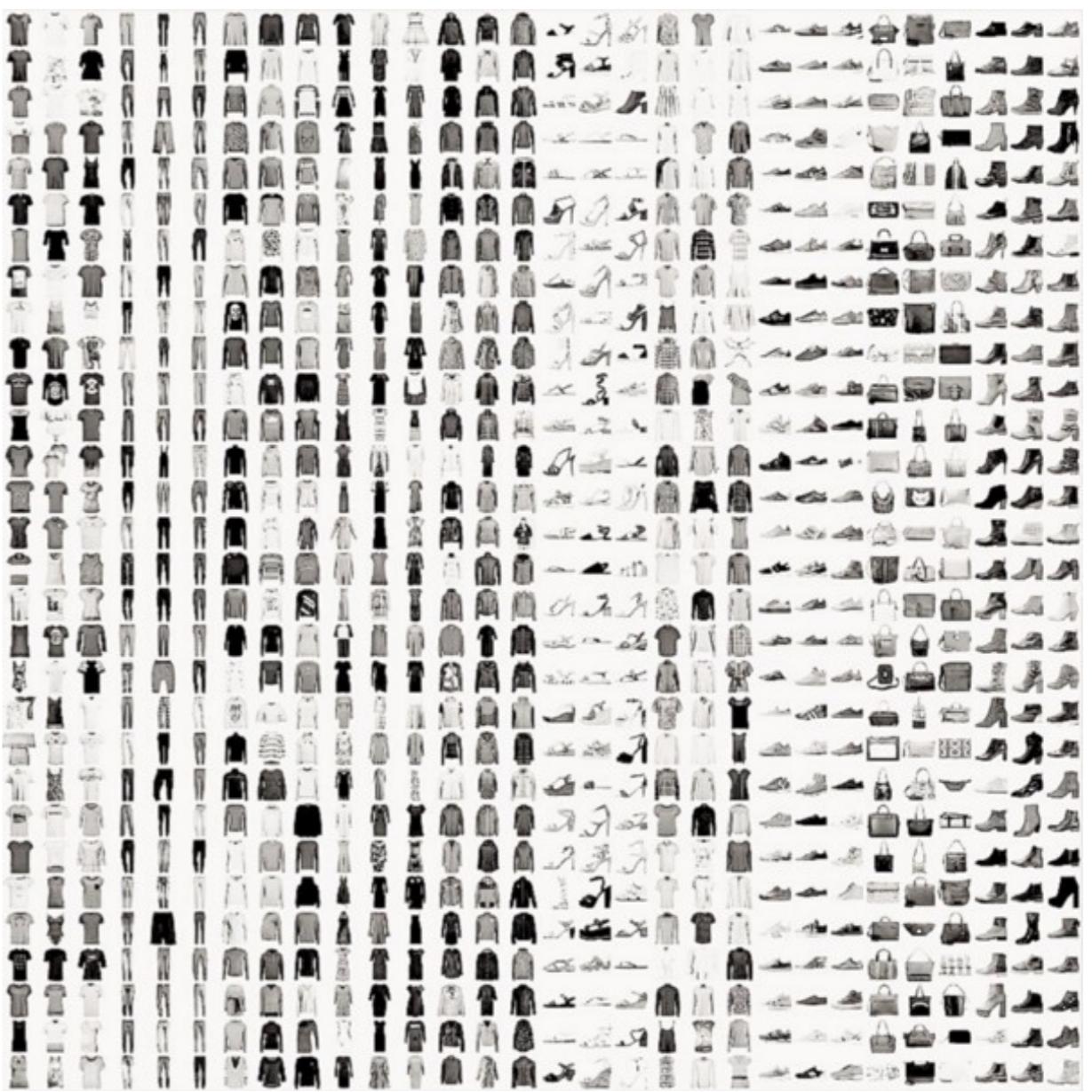
Text



Images

CNN for FashionMNIST

- <https://github.com/zalandoresearch/fashion-mnist>
- Alternative to MNIST
 - 60'000 train images
 - 10'000 test images
 - 28x28x1 grayscale
 - 10 classes
- A bit more challenging than MNIST



CNN for FashionMNIST - data preparation

Load the train and test data using panda

Get train data as numpy arrays.
Get I-hot target vector

Split the train set into a train/
validation set 80/20%

Get test data as numpy arrays

X_train has shape (48000, 784) after reading from csv with pandas. We need to reshape into images of 28x28x1, cast it to 'float32' and normalise [0.0, 1.0]

```
# Import libraries
import numpy as np
import pandas as pd
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# Load training and test data into dataframes
data_train = pd.read_csv('fashionmnist/fashion-mnist_train.csv')
data_test = pd.read_csv('fashionmnist/fashion-mnist_test.csv')

# X forms the training images, and y forms the training labels
X = np.array(data_train.iloc[:, 1:])
y = to_categorical(np.array(data_train.iloc[:, 0]))

# Split training data into train (80%) and validation (20%)
X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                test_size=0.2,
                                                random_state=13)

# X_test forms the test images, and y_test forms the test labels
X_test = np.array(data_test.iloc[:, 1:])
y_test = to_categorical(np.array(data_test.iloc[:, 0]))
```

```
input_shape = (28, 28, 1) # images are 28x28 px

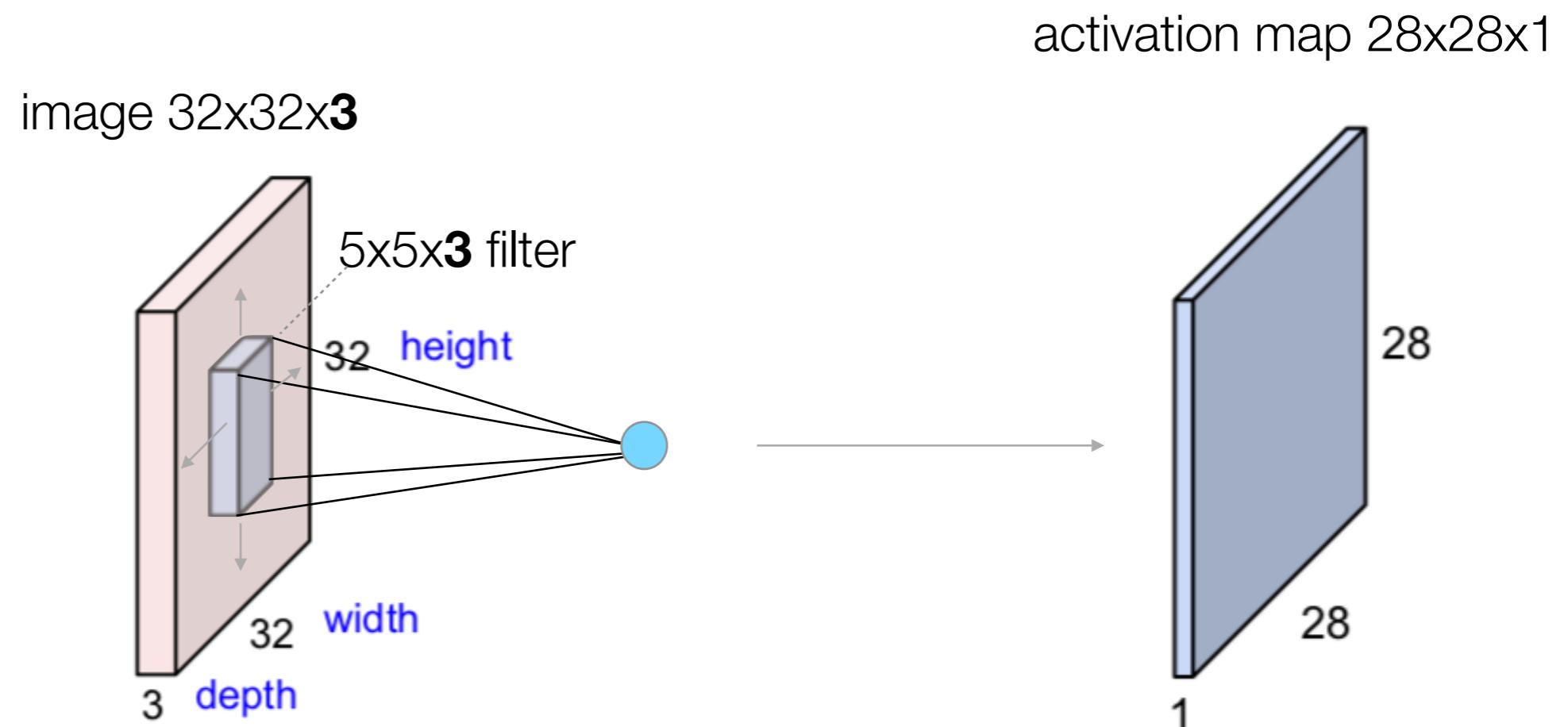
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')
X_train /= 255 #train images ready

X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32')
X_test /= 255 #test images ready

X_val = X_val.reshape(X_val.shape[0], 28, 28, 1).astype('float32')
X_val /= 255 #validation images ready
```

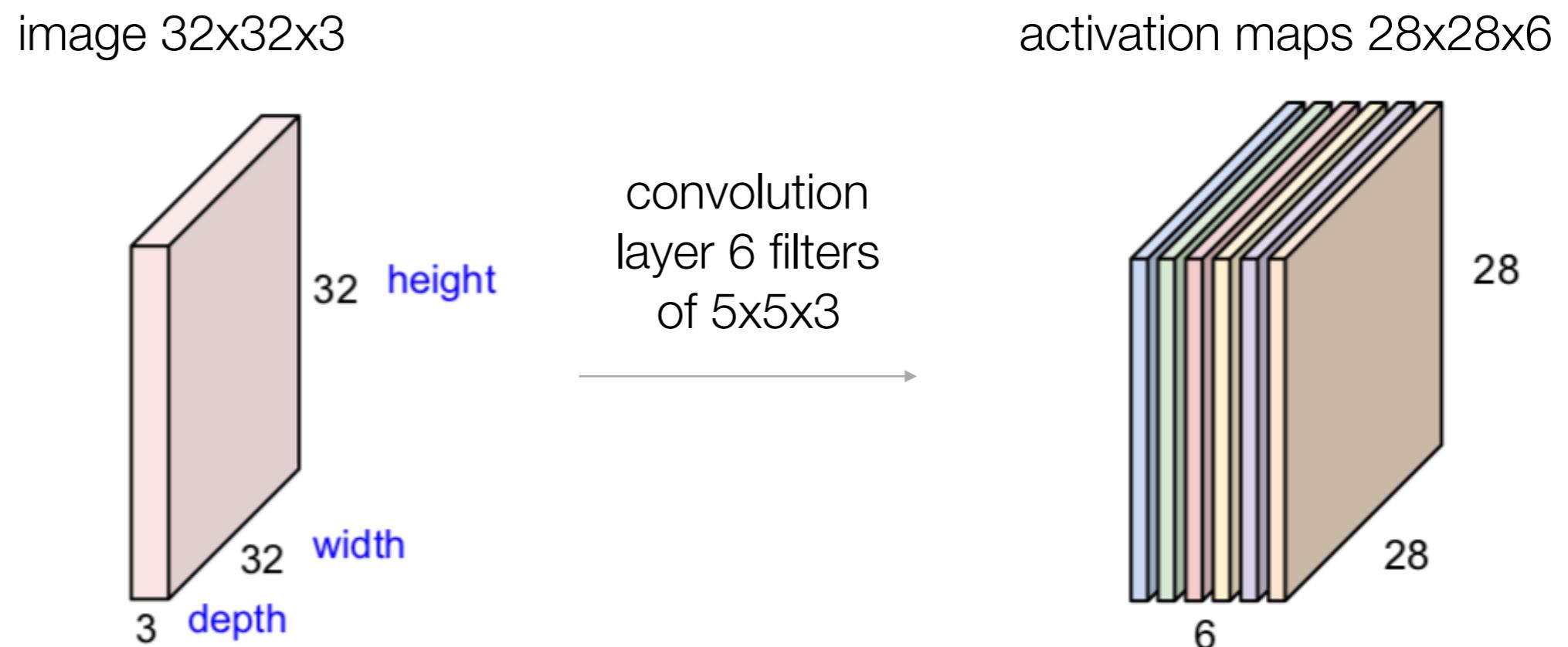
RECAP 3 - Convolution layer - 2D signals

- Notion of **activation map** that is the result of the convolution of the filter, i.e. the sliding over all possible spatial locations of the image.
- This sliding of the filter gives us **translation invariance** regarding to what the filter is sensitive for



RECAP 4 - Convolution layer - 2D signals

- Notion of **several filters** producing **several activation maps**
- Let's say we have 6 $5 \times 5 \times 3$ filters, we'll get 6 maps



CNN for FashionMNIST - network definition

We use the Sequential() model from Keras that allow to **stack** layers to compose a CNN.

1st layer is a Conv2D layer for the **convolution** operation that extracts features from the input images by sliding a convolution filter over the input to produce a feature map. Here we choose filters with size 3×3 . The first layer must also specify the size of the input with `input_shape=(28, 28, 1)`

The next layer is a ReLU **activation** layer.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Activation
from keras.layers import Conv2D, MaxPooling2D

cnn1 = Sequential()

cnn1.add(Conv2D(32, kernel_size=(3, 3), input_shape=data_shape))
cnn1.add(Activation('relu'))
cnn1.add(MaxPooling2D(pool_size=(2, 2)))
cnn1.add(Dropout(0.2))

cnn1.add(Flatten())

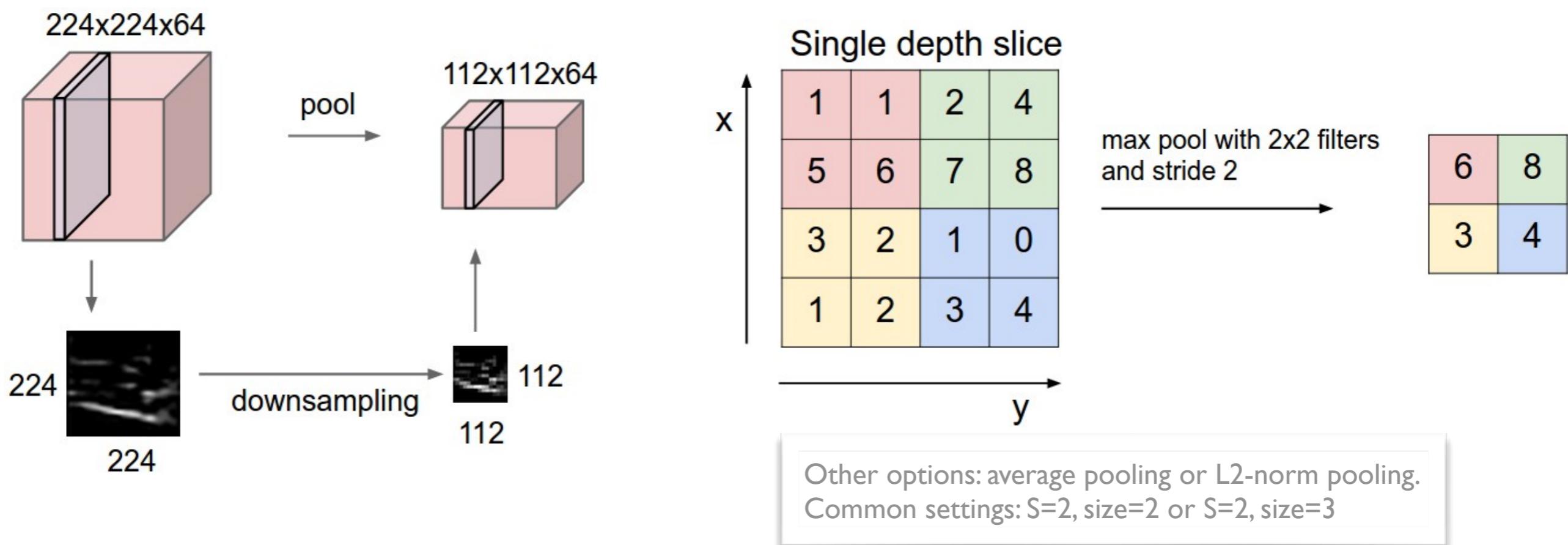
cnn1.add(Dense(128, activation='relu'))
cnn1.add(Dense(10, activation='softmax'))

cnn1.summary()
```

What is the size of the output feature maps?

RECAP 5 - Max pooling layer

- Reduce the spatial size of the representation
 - Applies independently to every depth, defined by a stride and size
 - Rationale:
 - most significant activations are kept, brings hierarchical approach
 - reduce the amount of computation and control overfitting



CNN for FashionMNIST - network definition

The next layer is a MaxPooling2D layer for the **max-pooling** operation that reduces the dimensionality of each feature, which helps shorten training time and reduce number of parameters. Here we choose the pooling window with size 2×2 .

To "fight" overfitting, we may add a **dropout** layer. It forces the model to learn multiple independent representations of the same data by randomly disabling neurons in the learning phase. We specify the quantity of neurons randomly disabled: 20%.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Activation
from keras.layers import Conv2D, MaxPooling2D

cnn1 = Sequential()

cnn1.add(Conv2D(32, kernel_size=(3, 3), input_shape=data_shape))
cnn1.add(Activation('relu'))
cnn1.add(MaxPooling2D(pool_size=(2, 2)))
cnn1.add(Dropout(0.2))

cnn1.add(Flatten())

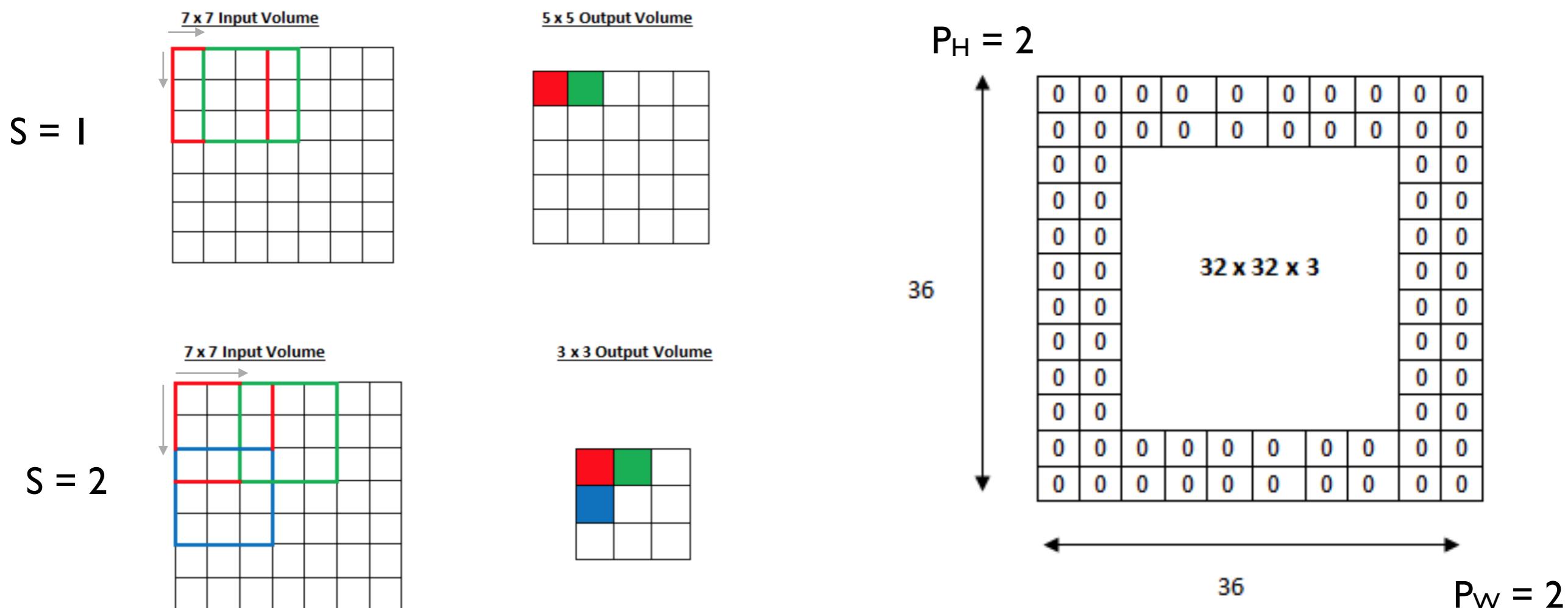
cnn1.add(Dense(128, activation='relu'))
cnn1.add(Dense(10, activation='softmax'))

cnn1.summary()
```

What is the size of the max pooling outputs?

RECAP 6 - Convolution layer - stride & padding

- The **stride S** specifies a step size when moving the filter across the signal.
- The **padding P** specifies the size of a zeroed frame added around the input.



CNN for FashionMNIST - network definition

The next step is to feed the last output tensor into one or several Dense, **fully-connected** layers.

These layers take as input vectors that are 1D. In our case, the output of the conv-relu-pool-dropout layers is a 3D tensor (13, 13, 32). Thus, we need to flatten the 3D outputs to 1D using a **Flatten** layer.

10 classes = 10 neurons
Softmax activation enables to estimate class posterior probabilities, i.e. winner class is the one with the highest value

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Activation
from keras.layers import Conv2D, MaxPooling2D

cnn1 = Sequential()

cnn1.add(Conv2D(32, kernel_size=(3, 3), input_shape=data_shape))
cnn1.add(Activation('relu'))
cnn1.add(MaxPooling2D(pool_size=(2, 2)))
cnn1.add(Dropout(0.2))

cnn1.add(Flatten())

cnn1.add(Dense(128, activation='relu'))
cnn1.add(Dense(10, activation='softmax'))

cnn1.summary()
```

The summary() methods prints out the structure of the network created with the sequential model.

CNN for FashionMNIST - network definition

Output of the summary() method.

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 26, 26, 32)	320
activation_4 (Activation)	(None, 26, 26, 32)	0
max_pooling2d_4 (MaxPooling2D)	(None, 13, 13, 32)	0
dropout_4 (Dropout)	(None, 13, 13, 32)	0
flatten_4 (Flatten)	(None, 5408)	0
dense_7 (Dense)	(None, 128)	692352
dense_8 (Dense)	(None, 10)	1290
<hr/>		
Total params: 693,962		
Trainable params: 693,962		
Non-trainable params: 0		

CNN for FashionMNIST - compile & train

We choose here `categorical_crossentropy` as the loss function. This loss is relevant for multiclass, single-label classification problem.

We chose here the `adam` optimizer. The Adam optimizer is an improvement over SGD(Stochastic Gradient Descent). The optimizer is defining the update rule for the weights of the neurons during backpropagation gradients.

```
cnn1.compile(loss='categorical_crossentropy', optimizer='adam',  
             metrics=['accuracy'])
```

```
history1 = cnn1.fit(X_train, y_train,  
                     batch_size=256,  
                     epochs=10,  
                     verbose=1,  
                     validation_data=(X_val, y_val))
```

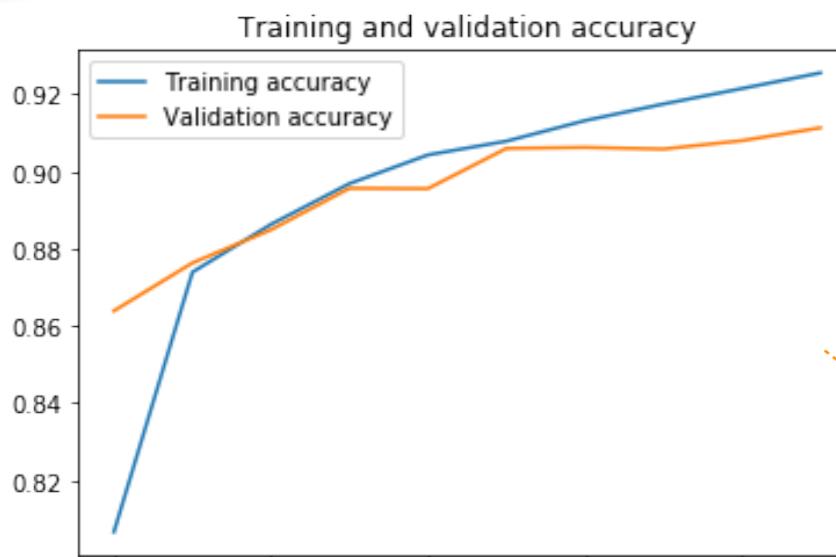
We can now train the model. We need to select the batch size and the number of epochs. In this case we can set it to batches of 256 with 10 epochs. We also need to pass in the training and validation data.

CNN for FashionMNIST - evaluate

We can then evaluate on the test set. The evaluation is an accuracy.

```
score1 = cnn1.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score1[0])
print('Test accuracy:', score1[1])
```

```
Test loss: 0.2484699842095375
Test accuracy: 0.9104
```



```
import matplotlib.pyplot as plt
%matplotlib inline

epochs = range(len(history1.history['acc']))

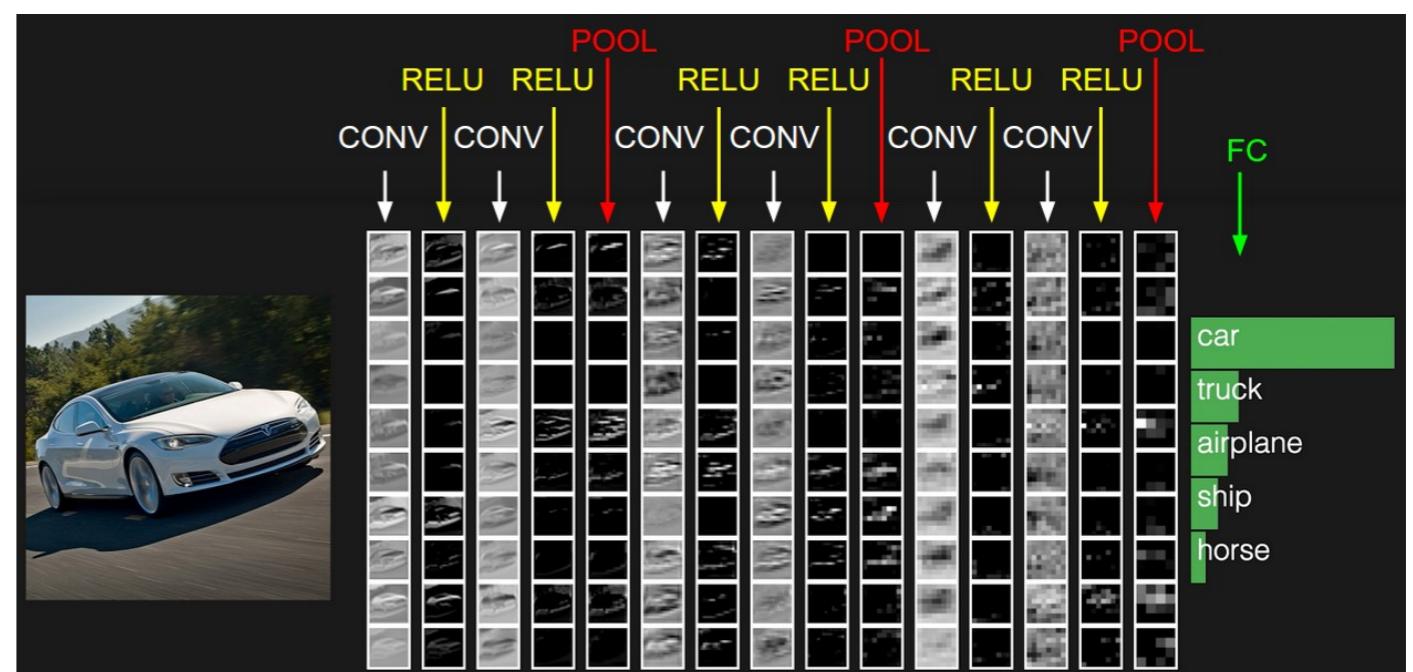
plt.plot(epochs, history1.history['acc'],
         label='Training accuracy')
plt.plot(epochs, history1.history['val_acc'],
         label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()

plt.plot(epochs, history1.history['loss'],
         label='Training loss')
plt.plot(epochs, history1.history['val_loss'],
         label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

CNN for FashionMNIST - going deeper

As seen in the previous exercises, we may improve the model performances by stacking CONV-RELU-POOL layers and playing with the different hyperparameters of the different layers: number of filters, kernel size, etc.

```
# add blocks such as these one below to increase the depth
cnn1.add(Conv2D(32, kernel_size=(3, 3), input_shape=input_shape))
cnn1.add(Activation('relu'))
cnn1.add(MaxPooling2D(pool_size=(2, 2)))
cnn1.add(Dropout(0.2))
```

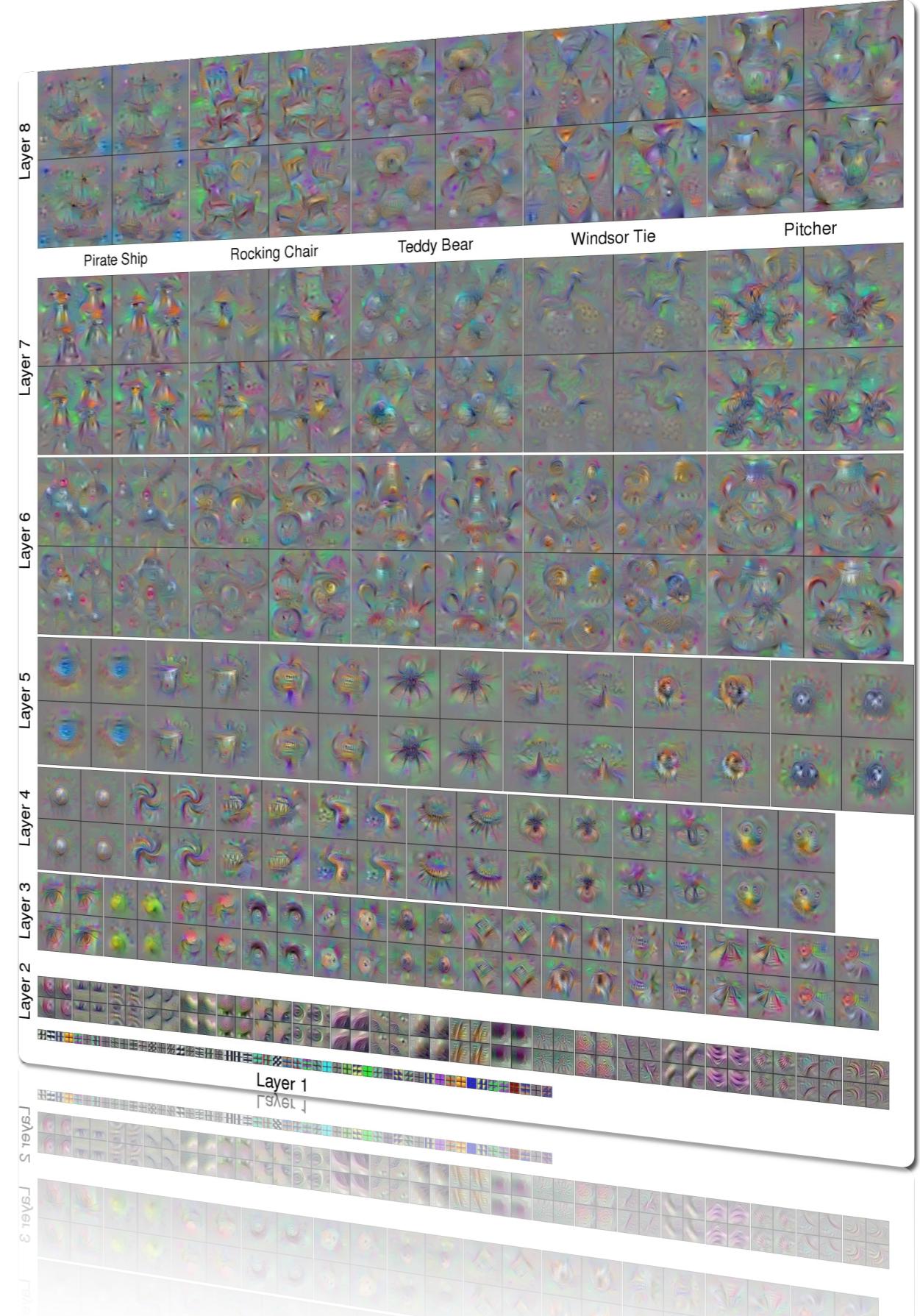


CIFAR10 - online demo

- [https://cs.stanford.edu/people/karpathy/convnetjs/
demo/cifar10.html](https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html)

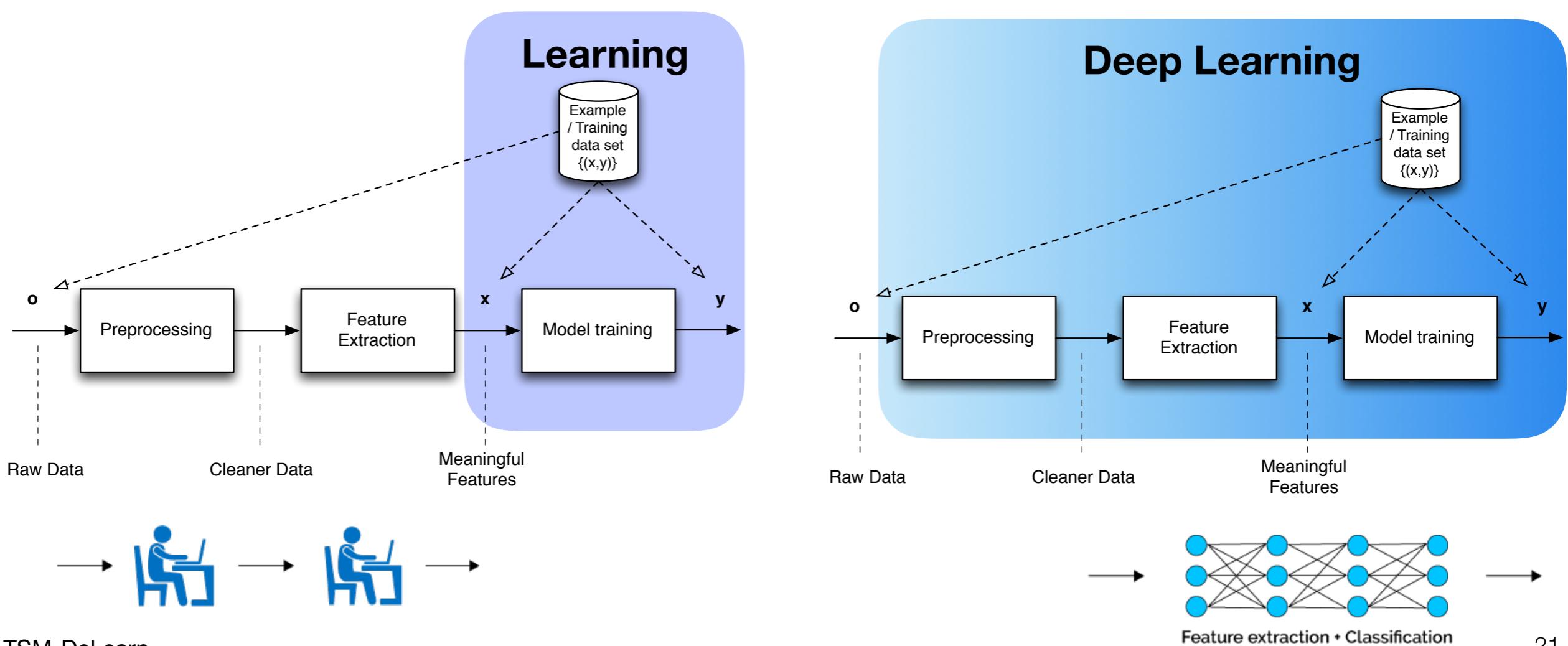
CNN and hierarchical features

Feature extraction
Hierarchy of features
Visualisation
Activation maps
Inputs with max outputs

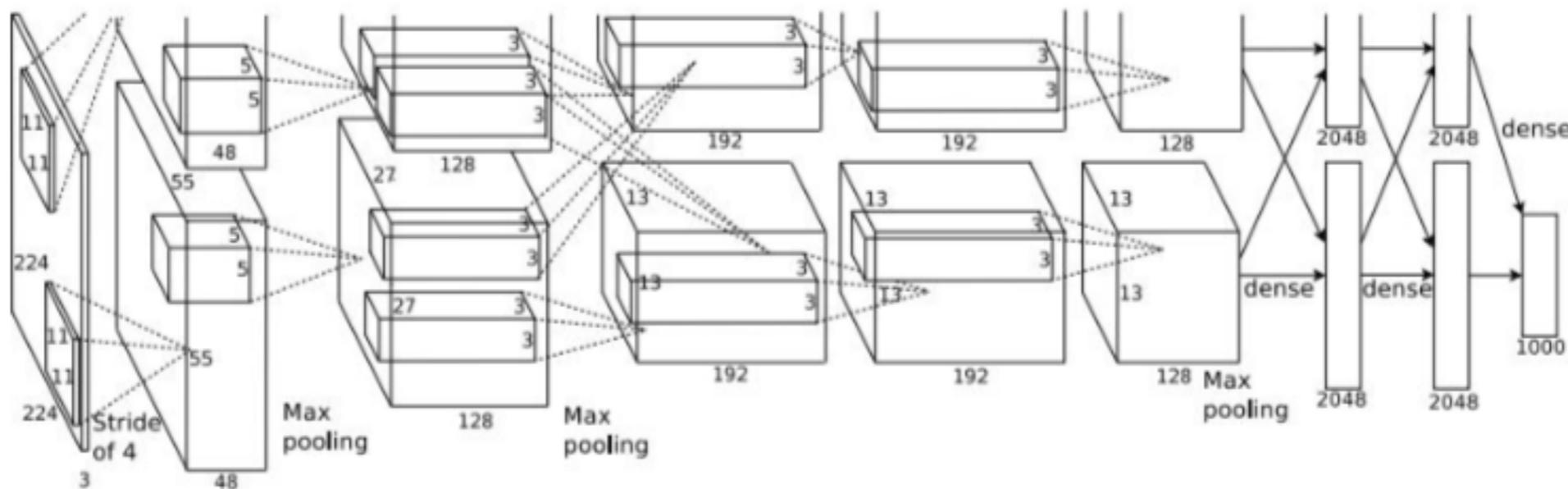


Convolution layers are extracting features

- The filter parameters are learnt through the training process. The whole system is learning not only to classify but also to extract features.



A hierarchy of features

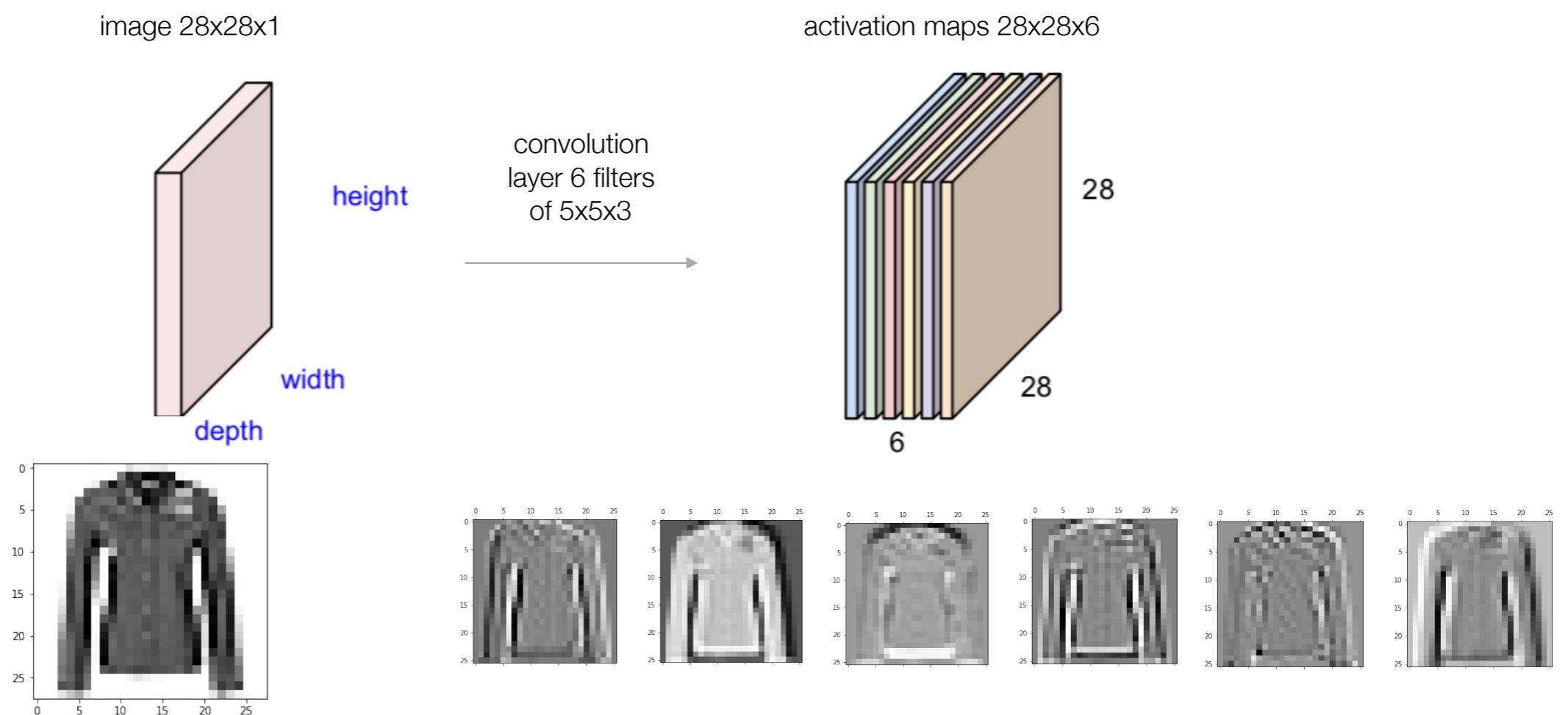


- Above is the AlexNet architecture [2012]
- The deeper the network:
 - The bigger are the number of filters in the conv layers
 - The smaller are the activation maps due to max pooling
- Intuitively:
 - Early layers will extract lower-level features
 - Late layers will extract higher-level features

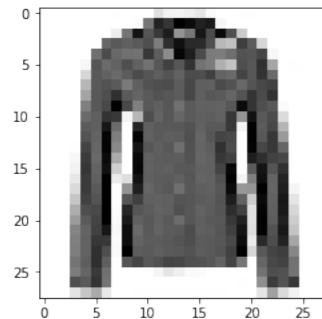
**How can we visualise
what has learnt the
network?**

Visualisation - activation maps

- First strategy: **visualise the activation maps**
 - For a given conv layer, consider each filter independent and visualise the corresponding activation map



Visualisation - activation maps



Extract the output of the 1st layer from the CNN model.

Rebuild a new Keras Model with the input layer and the output of the first layer.

The network expects a batch of images so we need to reshape the input image into a batch of 1.

Do the forward pass and plot the activations for 6 of the filters.

```
test_im1 = X_train[500]
plt.imshow(test_im1.reshape(28,28), cmap='Greys',
           interpolation='none')
plt.show()

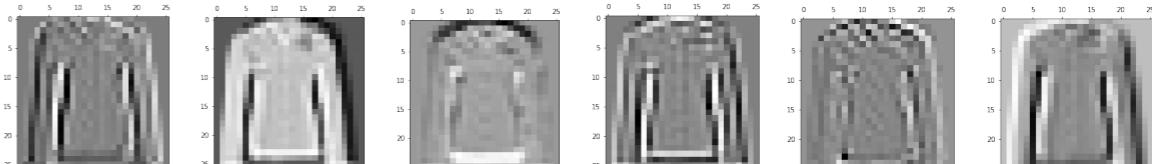
from keras import models
# extracts the output of the first layer
layer_1st_conv = cnn1.layers[0].output

# creates a model able to return these outputs, given an input
activation_model = models.Model(inputs=cnn1.input,
                                 outputs=layer_1st_conv)

# we need to reshape the image (28,28,1) into (1,28,28,1)
# as the network expect a batch of images as input
test_im1 = test_im1.reshape(1,28,28,1)

# returns the first layer activation
first_layer_activation = activation_model.predict(test_im1)

# display 6 of the activations of the 1st conv layer
plt.matshow(first_layer_activation[0, :, :, 0], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 1], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 2], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 3], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 4], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 5], cmap='Greys')
```

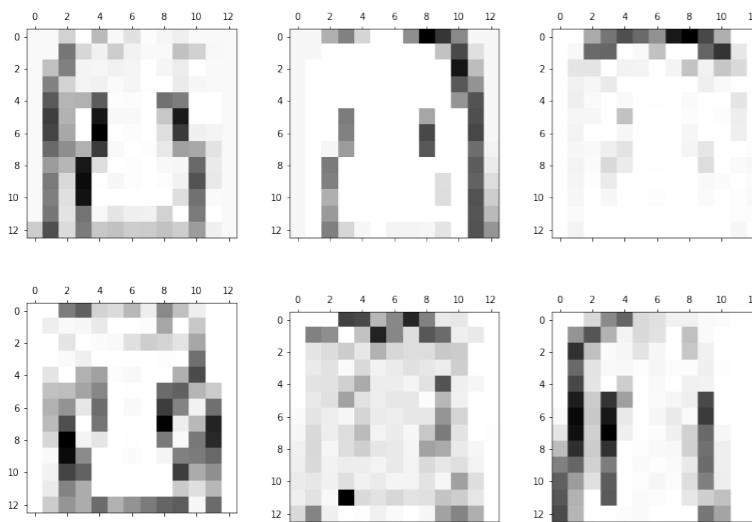


Visualisation - activation maps

- The same principle applies to all layers.
 - Below example for the first max pooling layer

Extract the output of the 3rd layer
from the CNN model = max pool

Rebuild a new Keras Model with
the input layer and the output of
the first max pooling layer.



```
from keras import models
# extracts the output of the first layer
layer_1st_maxpool = cnn1.layers[2].output

# creates a model able to return these outputs, given an input
activation_model = models.Model(inputs=cnn1.input,
                                 outputs=layer_1st_maxpool)

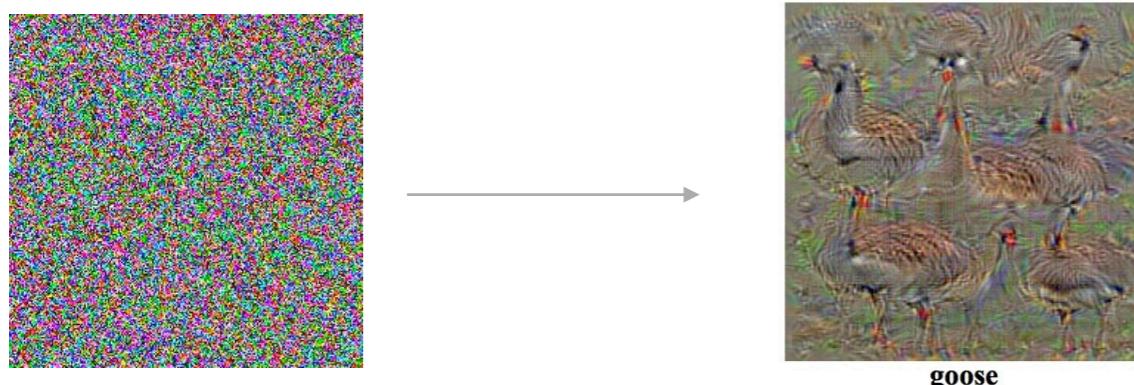
# we need to reshape the image (28,28,1) into (1,28,28,1)
# as the network expect a batch of images as input
test_im1 = test_im1.reshape(1,28,28,1)

# returns the first layer activation
first_maxpool_activation = activation_model.predict(test_im1)

# display 6 of the activations of the 1st conv layer
plt.matshow(first_maxpool_activation[0, :, :, 0], cmap='Greys')
plt.matshow(first_maxpool_activation[0, :, :, 1], cmap='Greys')
plt.matshow(first_maxpool_activation[0, :, :, 2], cmap='Greys')
plt.matshow(first_maxpool_activation[0, :, :, 3], cmap='Greys')
plt.matshow(first_maxpool_activation[0, :, :, 4], cmap='Greys')
plt.matshow(first_maxpool_activation[0, :, :, 5], cmap='Greys')
```

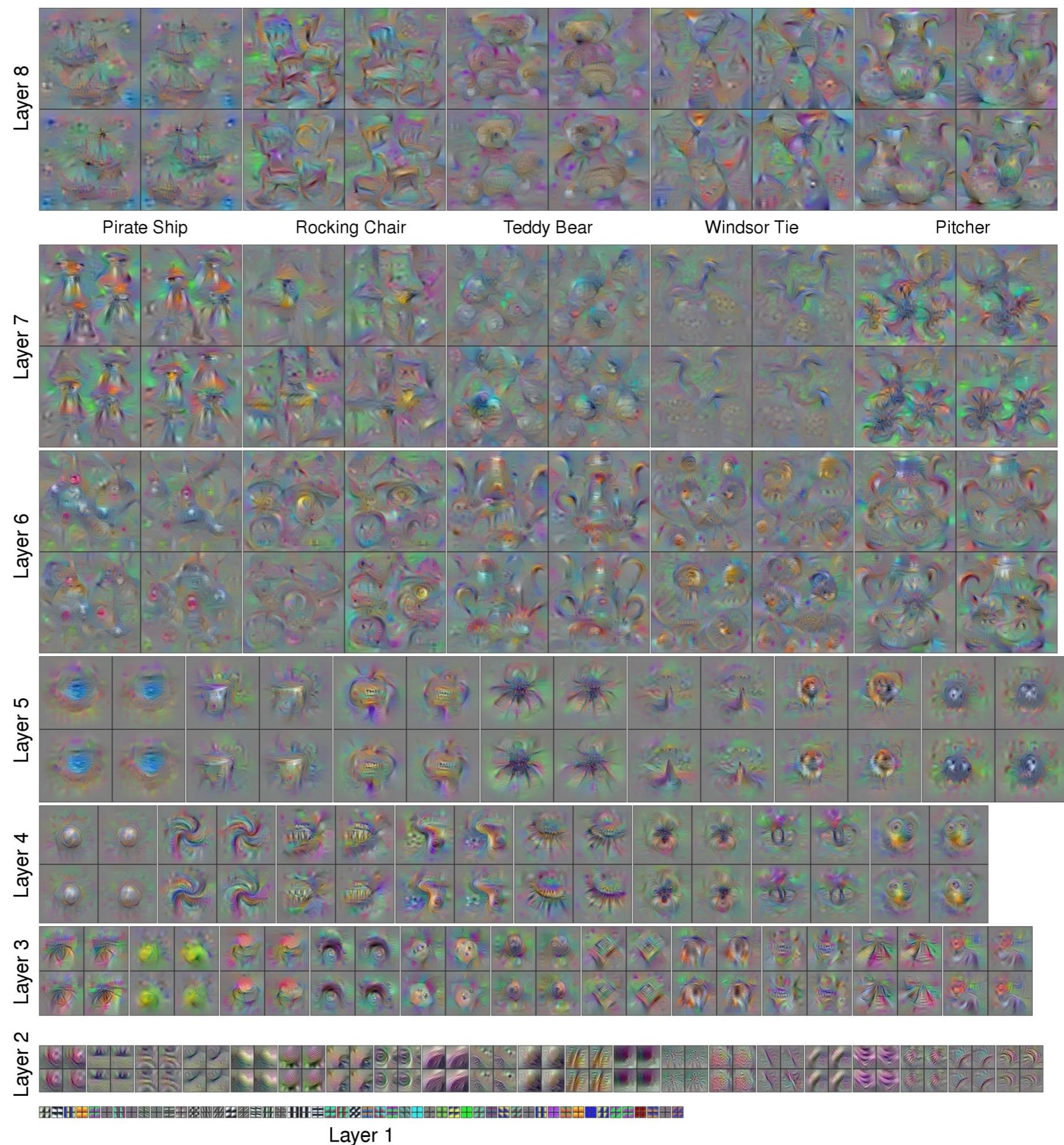
Visualisation - inputs with max activations

- Second strategy: find **input images that maximise the activation** of a given neuron
 - Start with a random image \mathbf{x}
 - Forward: compute the activation the neuron $a_{i,j}(\mathbf{x})$
 - Backward: perform the backprob of the gradient of $a_{i,j}(\mathbf{x})$ up to the pixel values: $\frac{\partial a_{i,j}(\mathbf{x})}{\partial \mathbf{x}}$
 - This gradient tells us how to change the pixel values to increase the activation of the neuron. We can then apply an update rule in the form of a gradient ascent:
$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \frac{\partial a_{i,j}(\mathbf{x})}{\partial \mathbf{x}} + \text{Regularisation Term}$$
 - Iterate in 2 until convergence



The **regularisation** term impeach pixels to be at extreme values. It kind of pressure to converge to natural looking images. See e.g. <http://yosinski.com/deepvis>

Source: <http://yosinski.com/deepvis>



Deep Visualization Toolbox

yosinski.com/deepvis

#deepvis



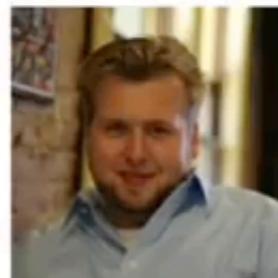
Jason Yosinski



Jeff Clune



Anh Nguyen



Thomas Fuchs



Hod Lipson



Cornell University



UNIVERSITY
OF WYOMING

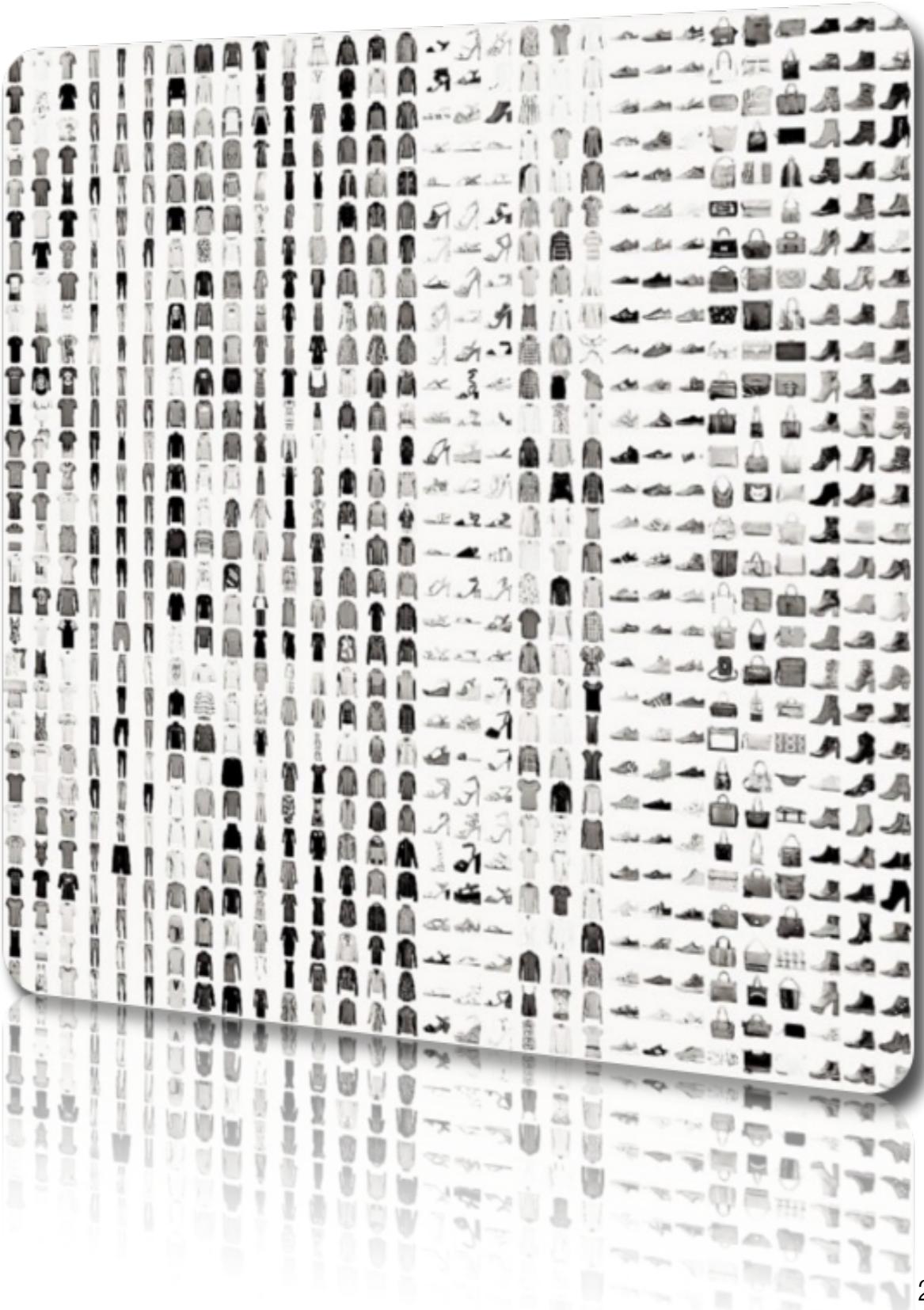


Jet Propulsion Laboratory
California Institute of Technology

<http://yosinski.com/deepvis>
<https://youtu.be/AgkflQ4IGaM>

Data Augmentation

Principles
Example on
FashionMNIST

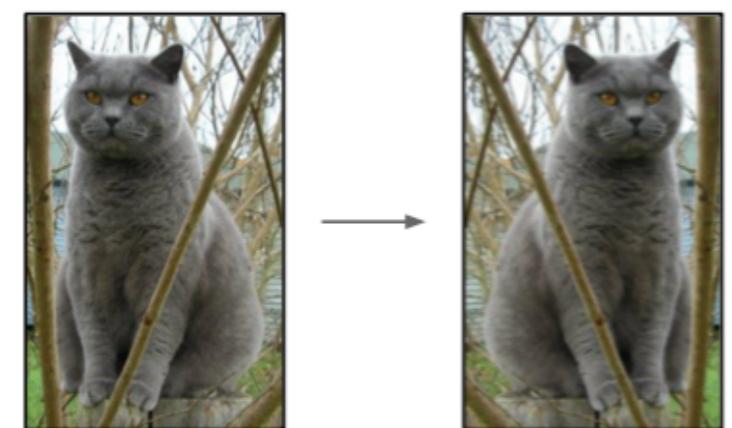
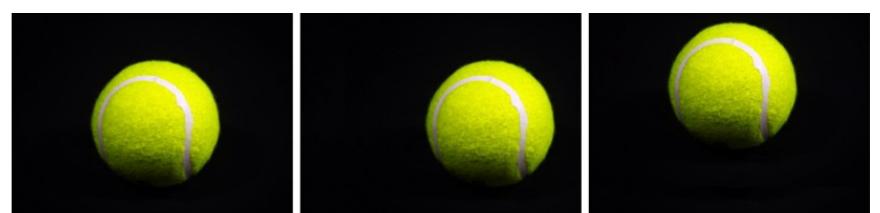


Principles of data augmentation

- Overfitting can be caused by having networks with too many parameters that are trained on too few samples. Through training, the model learns *by heart* and generalises poorly.
- **Data augmentation** takes the approach of generating artificially more training data from existing training samples.
- For images, data augmentation is performed via a number of random transformations that yield believable-looking images.
- The goal is that at training time, the model will not see the exact same picture twice. This helps expose the model to more aspects of the data and generalise better.

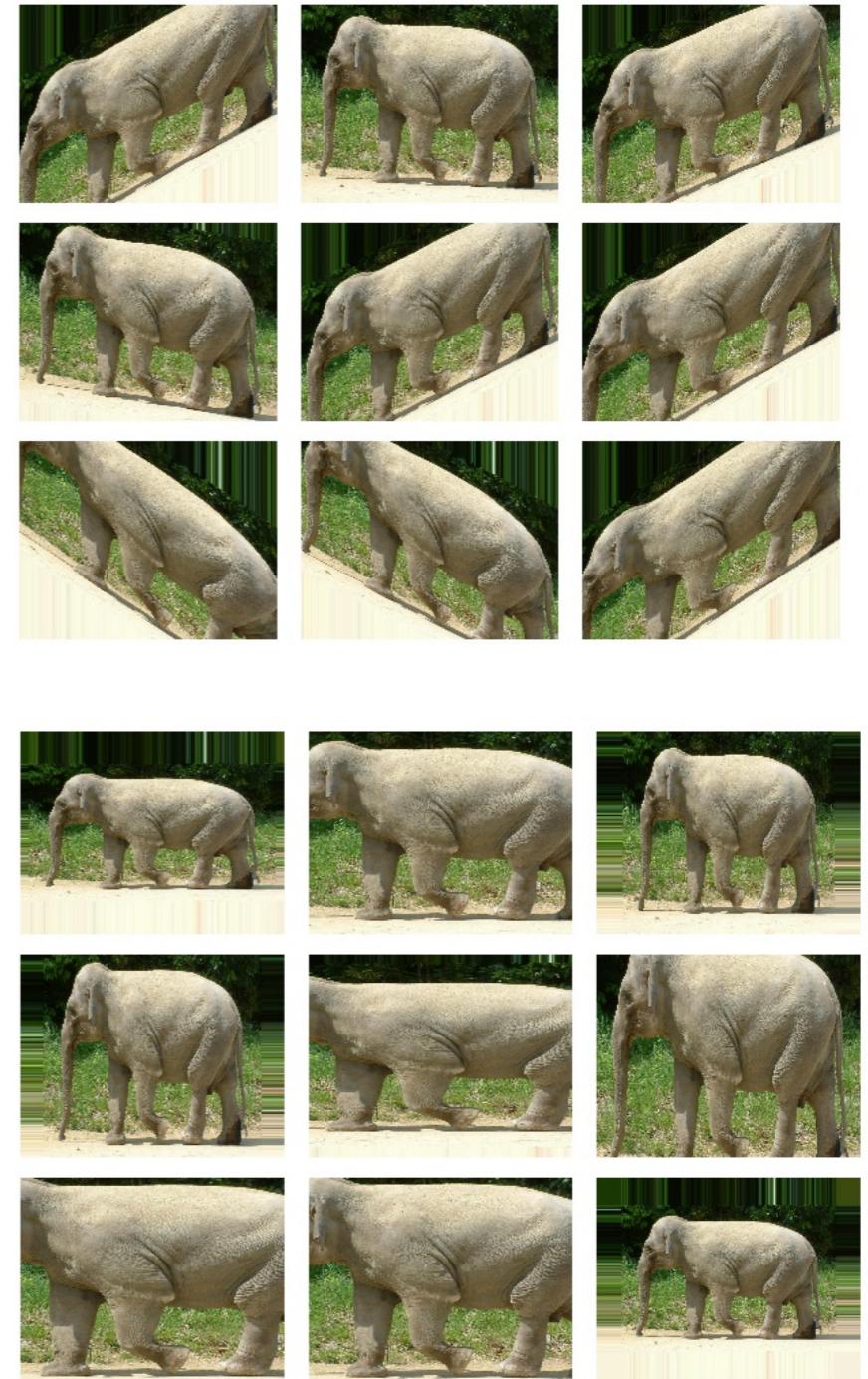
2D data augmentation

- Quantity of “perturbations” are randomly chosen
- **Rotation**
 - range of degrees
- **Translation**
 - percentage of shift in a range
 - for width and height
- **Flip**
 - may not be applicable to all types of images
 - e.g. images with text



2D data augmentation

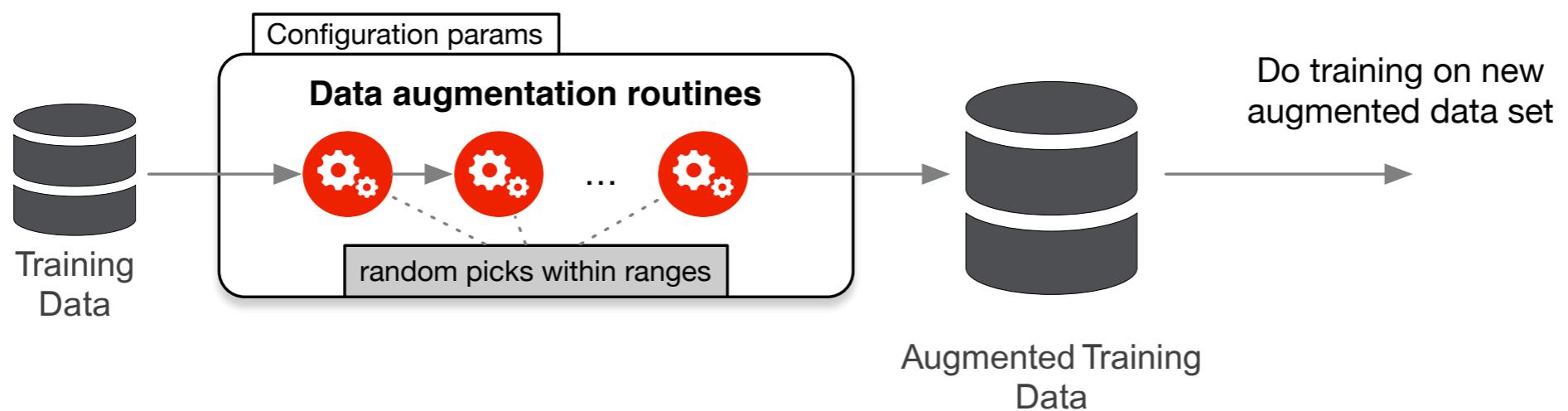
- **Shear**
- **Zoom**
- **Color**
 - shifting
 - illumination
 - ...



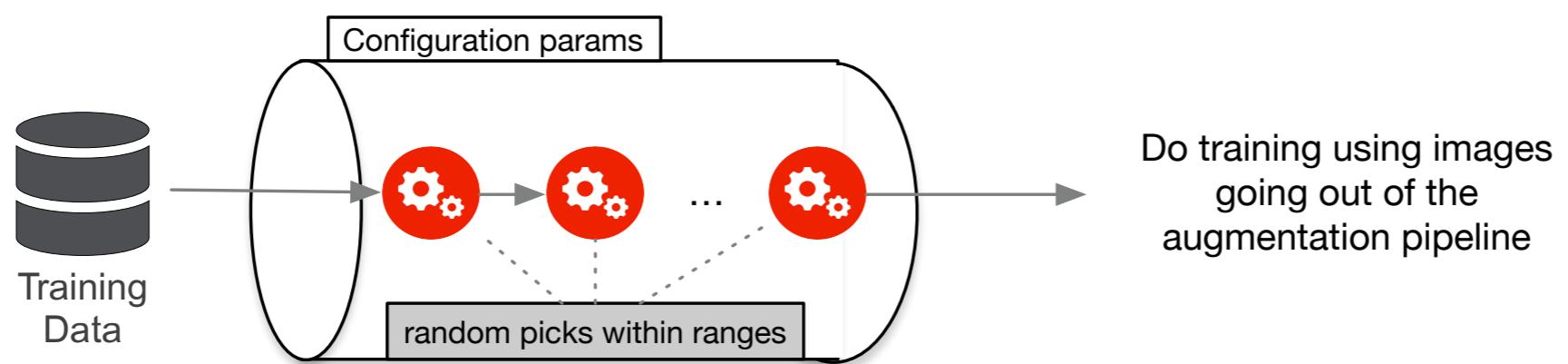


Data augmentation - implementation strategies

Data augmentation - pre-augmentation

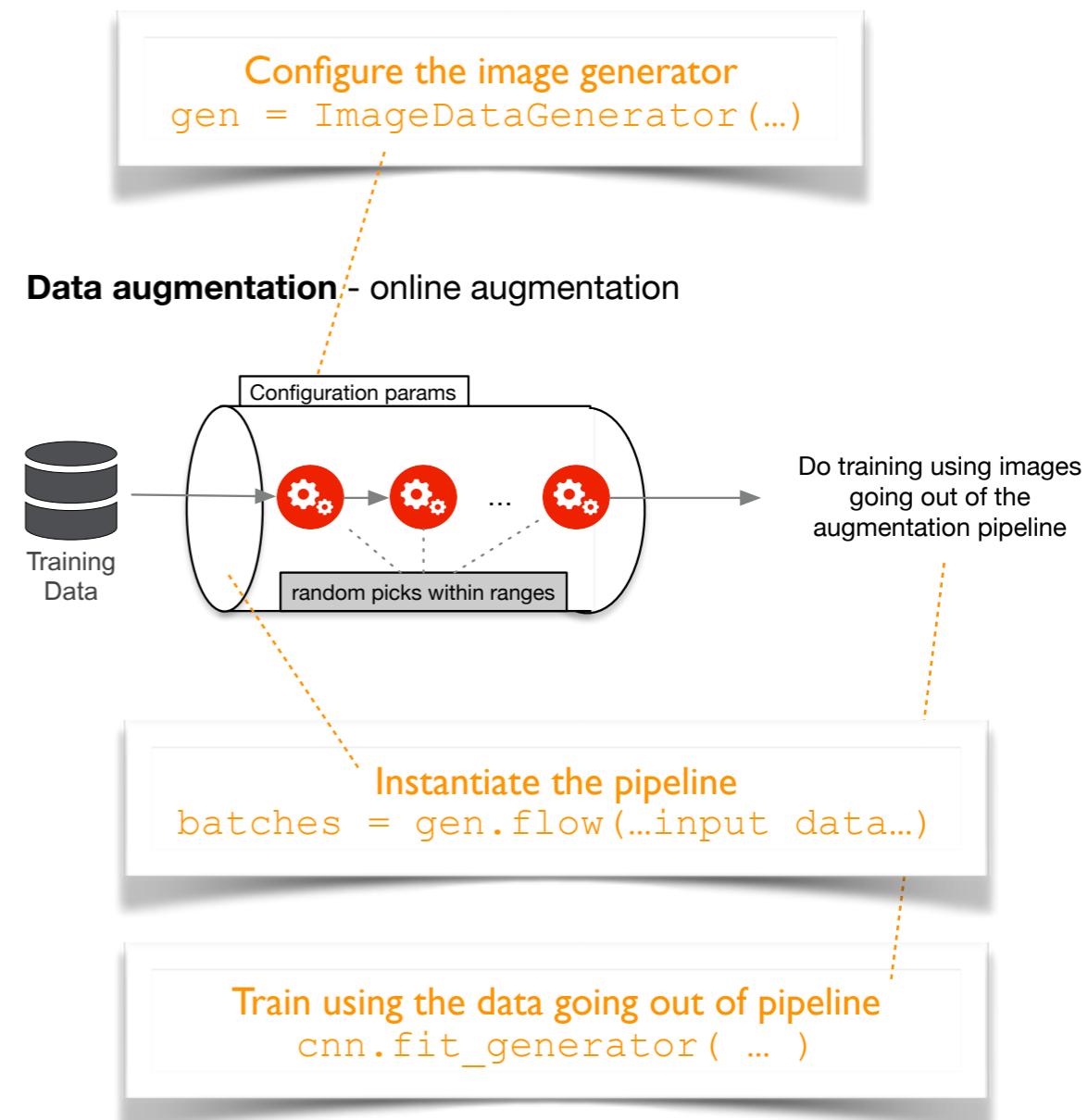


Data augmentation - online augmentation



Data augmentation - Keras implementation

- Principle: online augmentation strategy
- Steps:
 - Configure the image data generator
 - Which augmentation to apply
 - In which ranges
 - Instantiate a pipeline on the input data set
 - Tell the training method that you use a pipeline



CNN for FashionMNIST - data augmentation

In Keras, the `ImageDataGenerator()` can be configured with many different options for rotation, shift, shear, zoom , etc.

Flow pipelines can then be declared, here fed by training and validation images

Training can be done calling the `fit_generator()` method that will take the outputs of the data augmentation pipelines

We can evaluate as usual

```
from keras.preprocessing.image import ImageDataGenerator
gen = ImageDataGenerator(rotation_range=8,
                          width_shift_range=0.08,
                          shear_range=0.3,
                          height_shift_range=0.08,
                          zoom_range=0.08)

batches = gen.flow(X_train, y_train, batch_size=256)
val_batches = gen.flow(X_val, y_val, batch_size=256)

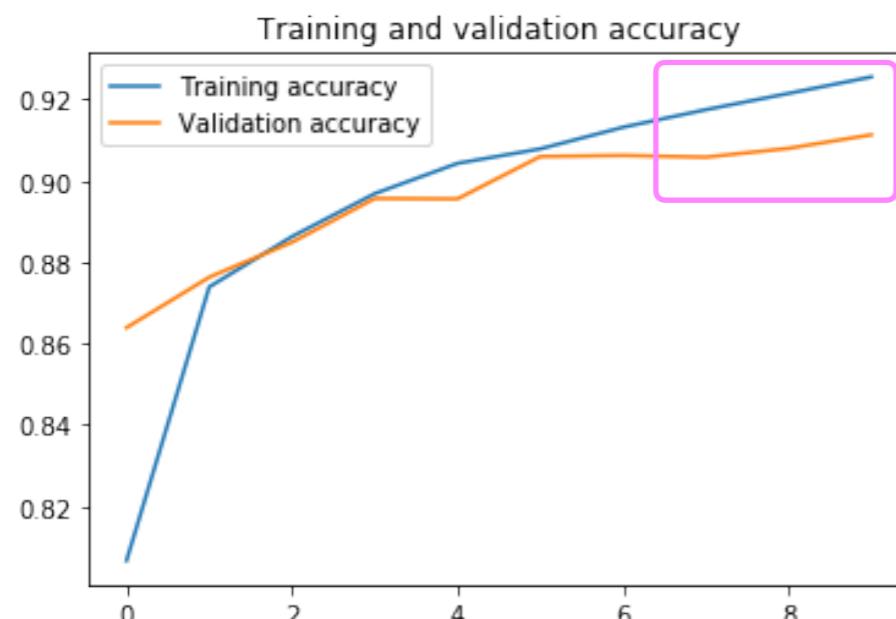
history1 = cnn1.fit_generator(batches,
                             steps_per_epoch=48000//256,
                             epochs=50,
                             validation_data=val_batches,
                             validation_steps=12000//256,
                             use_multiprocessing=True)

score1 = cnn1.evaluate(X_test, y_test, verbose=1)
print('Test loss:', score1[0])
print('Test accuracy:', score1[1])

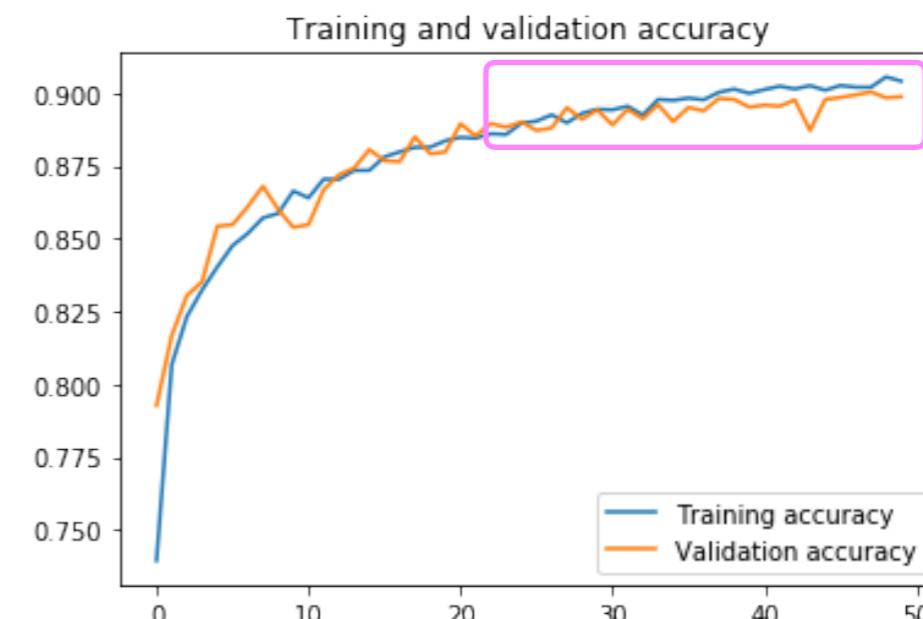
10000/10000 [=====] - 1s 90us/step
Test loss: 0.225622302877903
Test accuracy: 0.92
```

CNN for FashionMNIST - data augmentation

Without data augmentation



With data augmentation



```
score1 = cnn1.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score1[0])
print('Test accuracy:', score1[1])

Test loss: 0.2484699842095375
Test accuracy: 0.9104
```

```
score1 = cnn1.evaluate(X_test, y_test, verbose=1)
print('Test loss:', score1[0])
print('Test accuracy:', score1[1])

10000/10000 [=====] - 1s 90us/step
Test loss: 0.225622302877903
Test accuracy: 0.92
```

- We may observe that with data augmentation:
 - Training shows less difference between training set and validation set - less prone to overfitting
 - Overall improvement is not so high: from 91.0% to 92.0%
 - This is probably due to the “stability” of the images of FashionMNIST

Deep CNN Architectures

Convolutional pattern

LeNet-5

AlexNet

VGGNet

GoogLeNet

ResNet



Convolutional layer patterns

- Common forms of conv. network stack layers as follows:

```
INPUT -> [ [CONV -> RELU]*N -> POOL?] *M -> [FC -> RELU]*K -> FC
```

- Where * indicates repetition, POOL? indicates optional pooling layer with: $N \geq 0$ (usually ≤ 3), $M \geq 0$, $K \geq 0$ (usually $K < 3$)

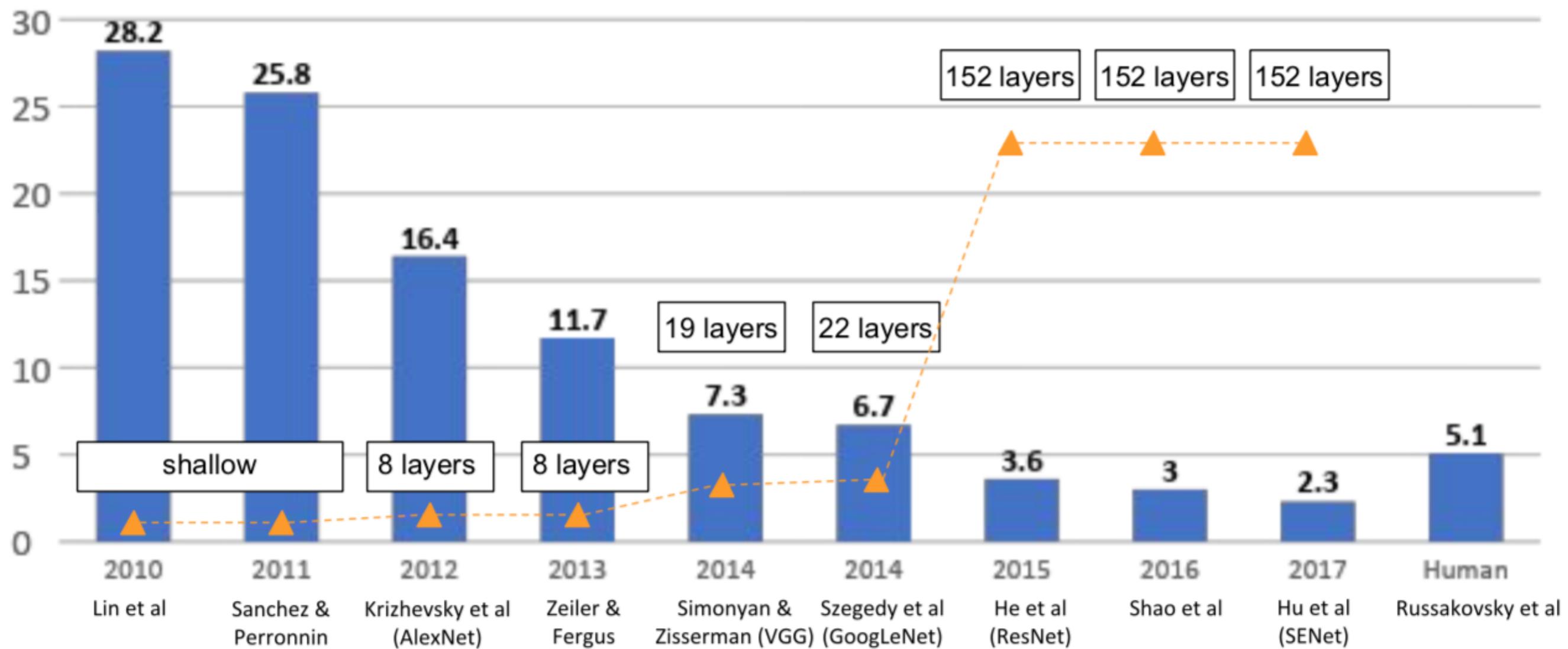
- For example:

```
INPUT -> CONV -> RELU -> FC
```

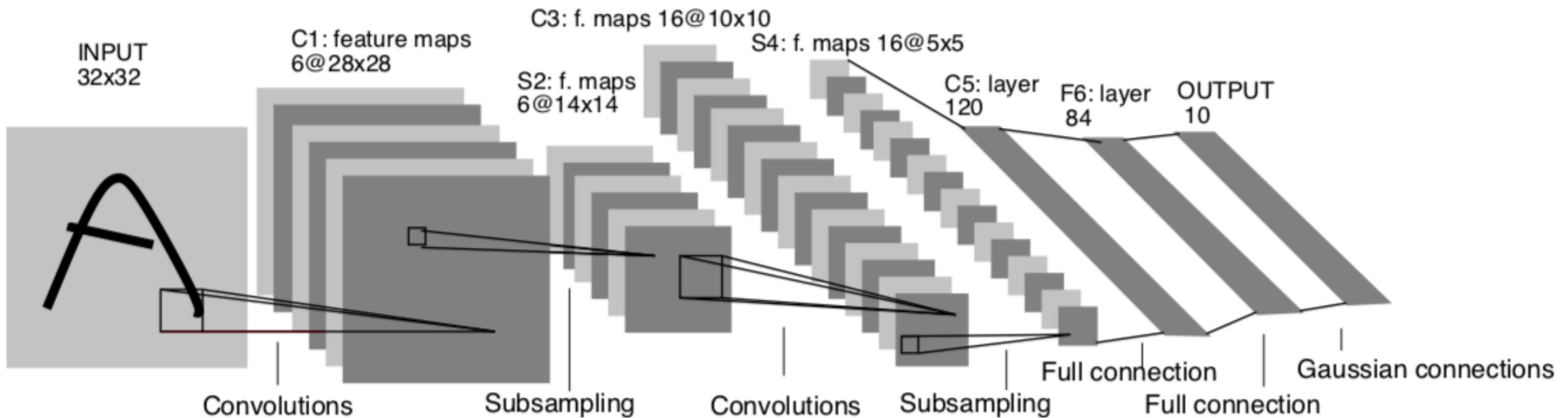
INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC. Here we see that there is a single CONV layer between every POOL layer.

INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) - winners



LeNet-5 [1998]

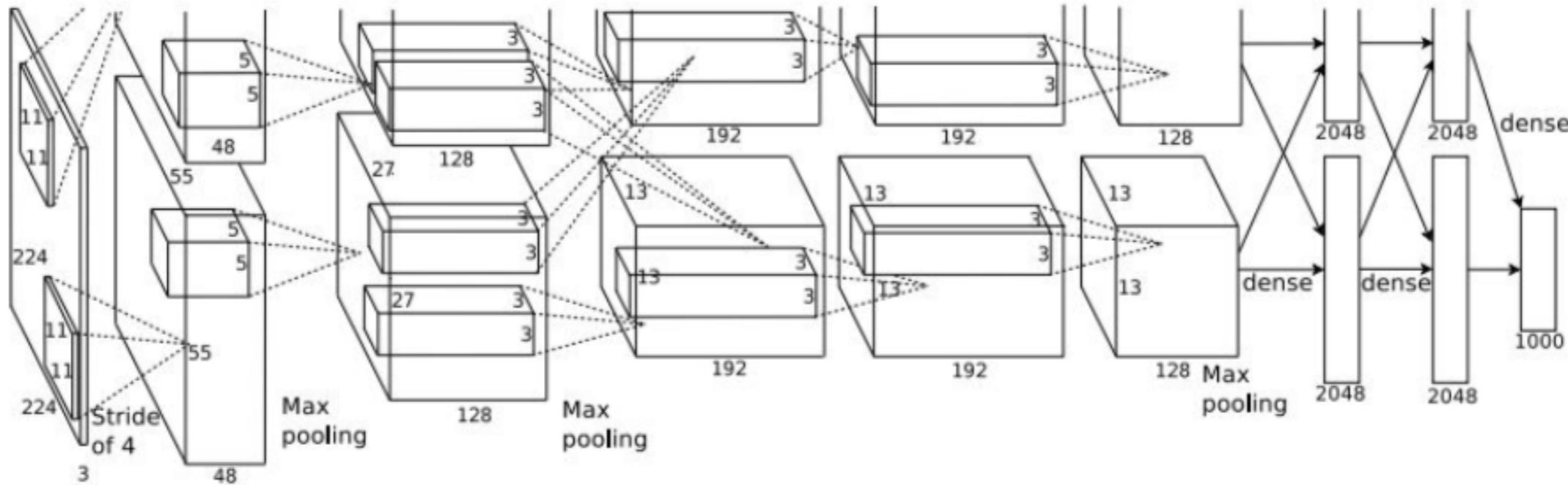


Conv filters were 5×5 , applied at stride 1

Subsampling (Pooling) layers were 2×2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

- LeNet. The first successful applications of Convolutional Networks were developed by **Yann LeCun** in 1990's. Of these, the best known is the LeNet architecture that was used to read zip codes, digits, etc.
- <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>

AlexNet [2012]



Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

CONV5

Max POOL3

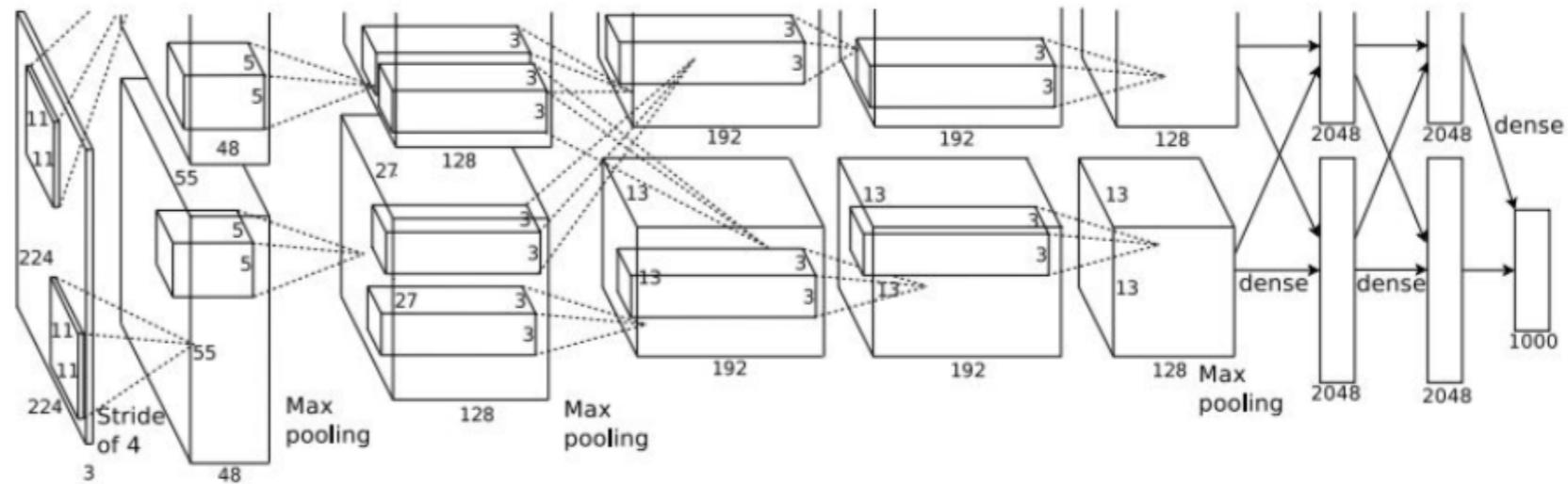
FC6

FC7

FC8

- The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by **Alex Krizhevsky**, Ilya Sutskever and Geoff Hinton.
- The AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error).
- The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
- <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>

AlexNet [2012]



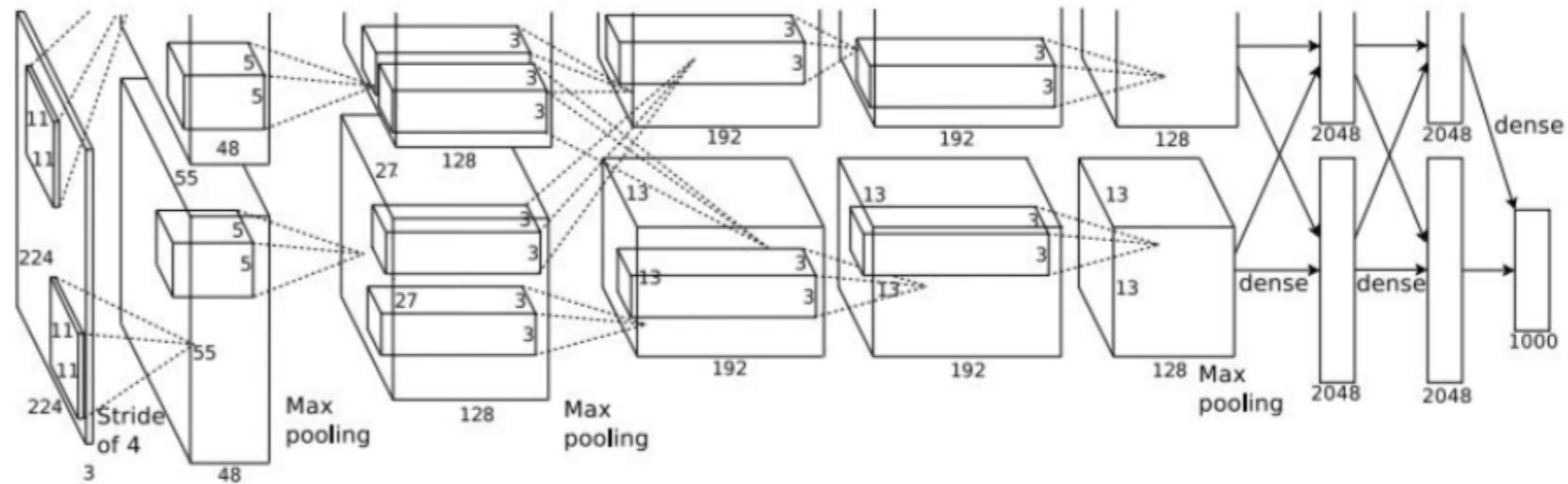
Input : 227x227x3 images

First layer (CONV1): 96 11x11
filters, S = 4

Q1: what is the output volume size of CONV1?

Q2: how many parameters for CONV1?

AlexNet [2012]



Input : 227x227x3 images

First layer (CONV1): 96 11x11 filters, S = 4

Q1: what is the output volume size of CONV1?

Q2: how many parameters for CONV1?

Q1:

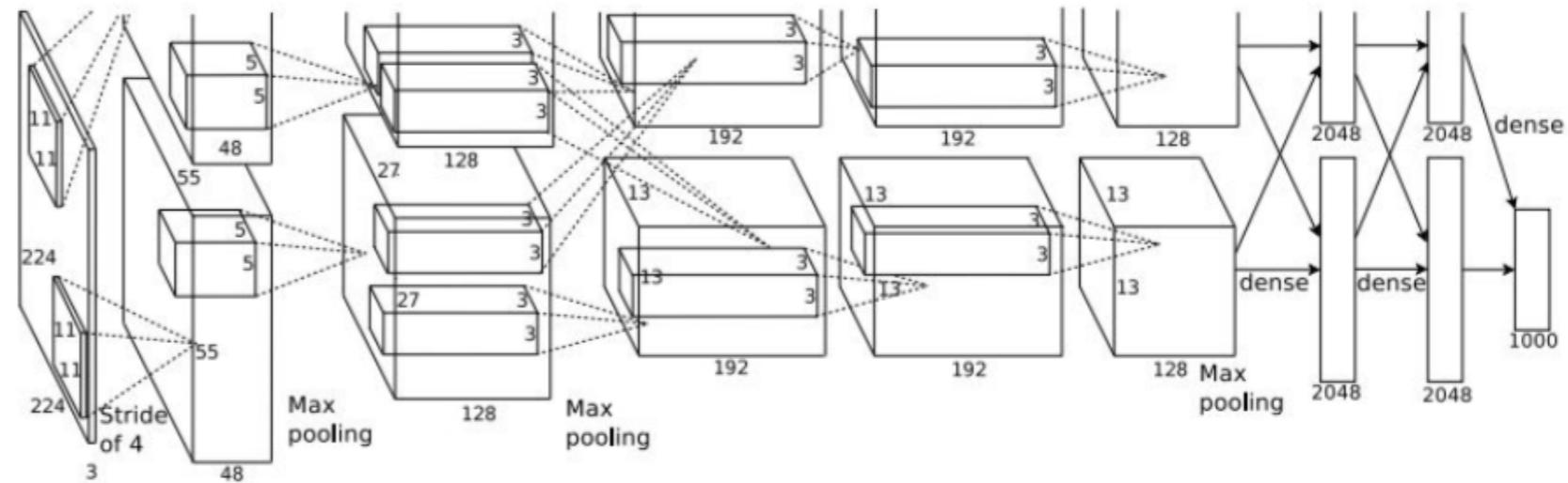
$$\text{activation map size} = (227-11)/4 + 1 = 55 \times 55$$

$$\text{output volume} = (55 \times 55 \times 96)$$

Q2:

$$96 \times (11 \times 11 \times 3) + 96 = 34'944 \sim 35K$$

AlexNet [2012]



Input : 227x227x3 images

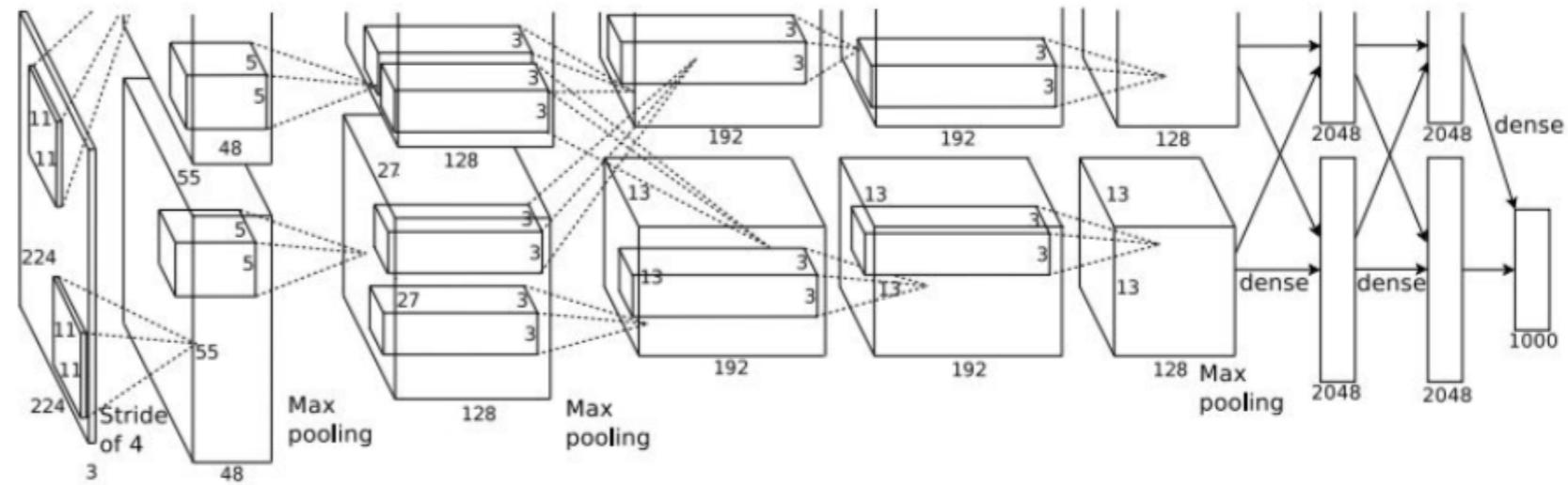
After CONV1: 55x55x96

Second layer (POOL1): 3x3, S = 2

Q3: what is the output volume size of
POOL1?

Q4: how many parameters for
POOL1?

AlexNet [2012]



Input : 227x227x3 images

After CONV1: 55x55x96

Second layer (POOL1): 3x3, S = 2

Q3: what is the output volume size of POOL1?

Q4: how many parameters for POOL1?

Q3:

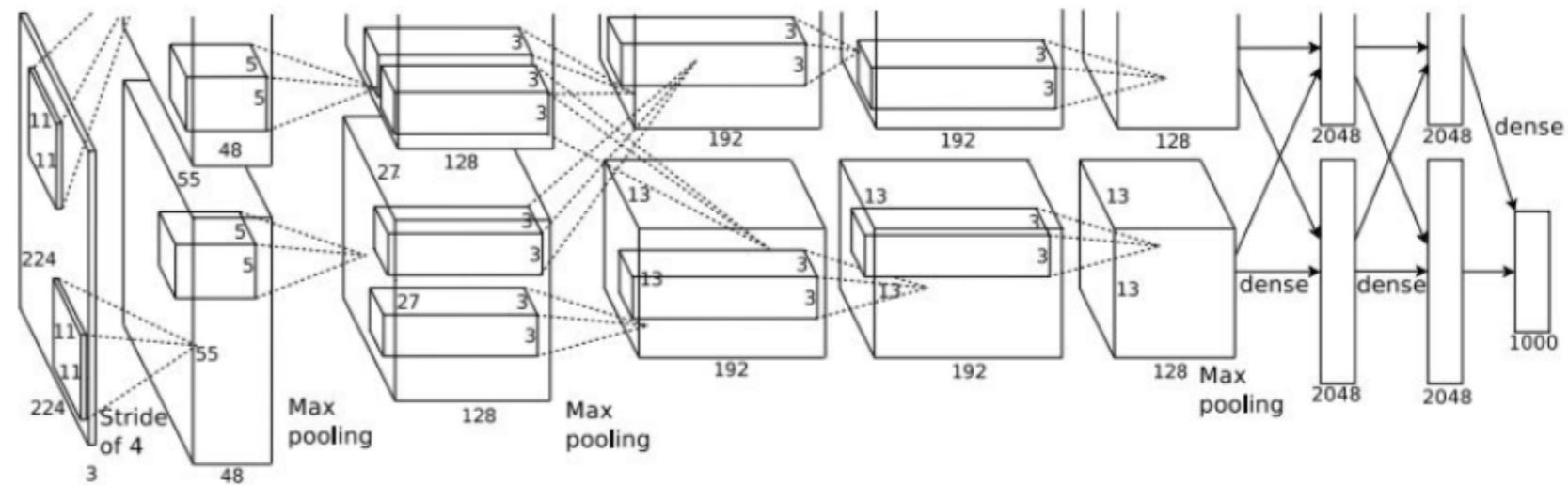
$$\text{activation map size} = (55-3)/2 + 1 = 27 \times 27$$

$$\text{output volume} = 27 \times 27 \times 96$$

Q4:

0

AlexNet [2012]



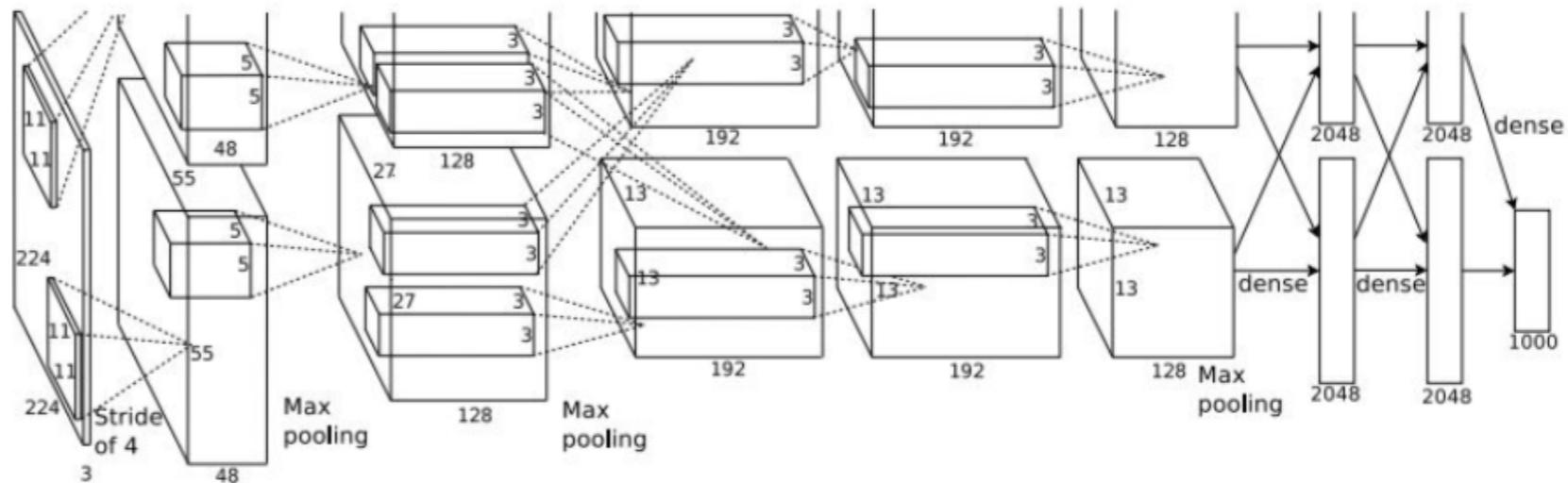
Input : 227x227x3 images

After CONV1: 55x55x96

After POOL1: 27x27x96

...

AlexNet [2012]



Full (simplified) AlexNet 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

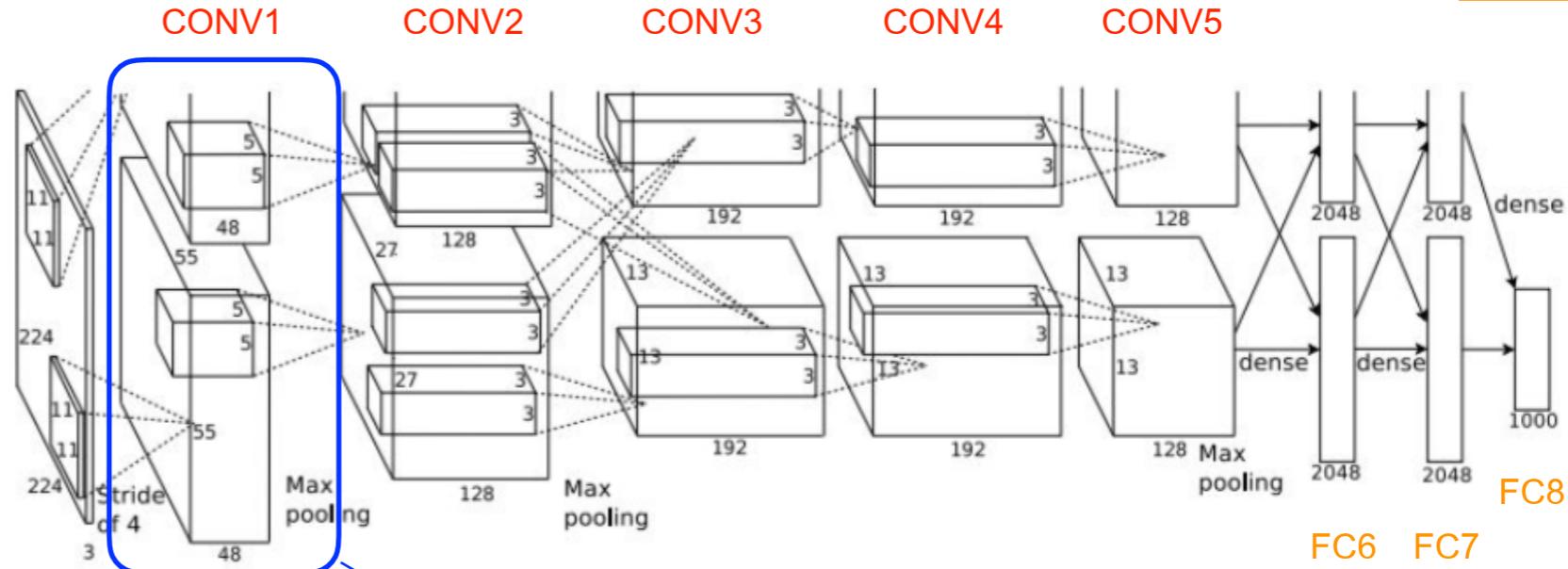
[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:

- first use of ReLU
- used Norm layers (not common anymore) - heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

AlexNet [2012]



Full (simplified) AlexNet 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)

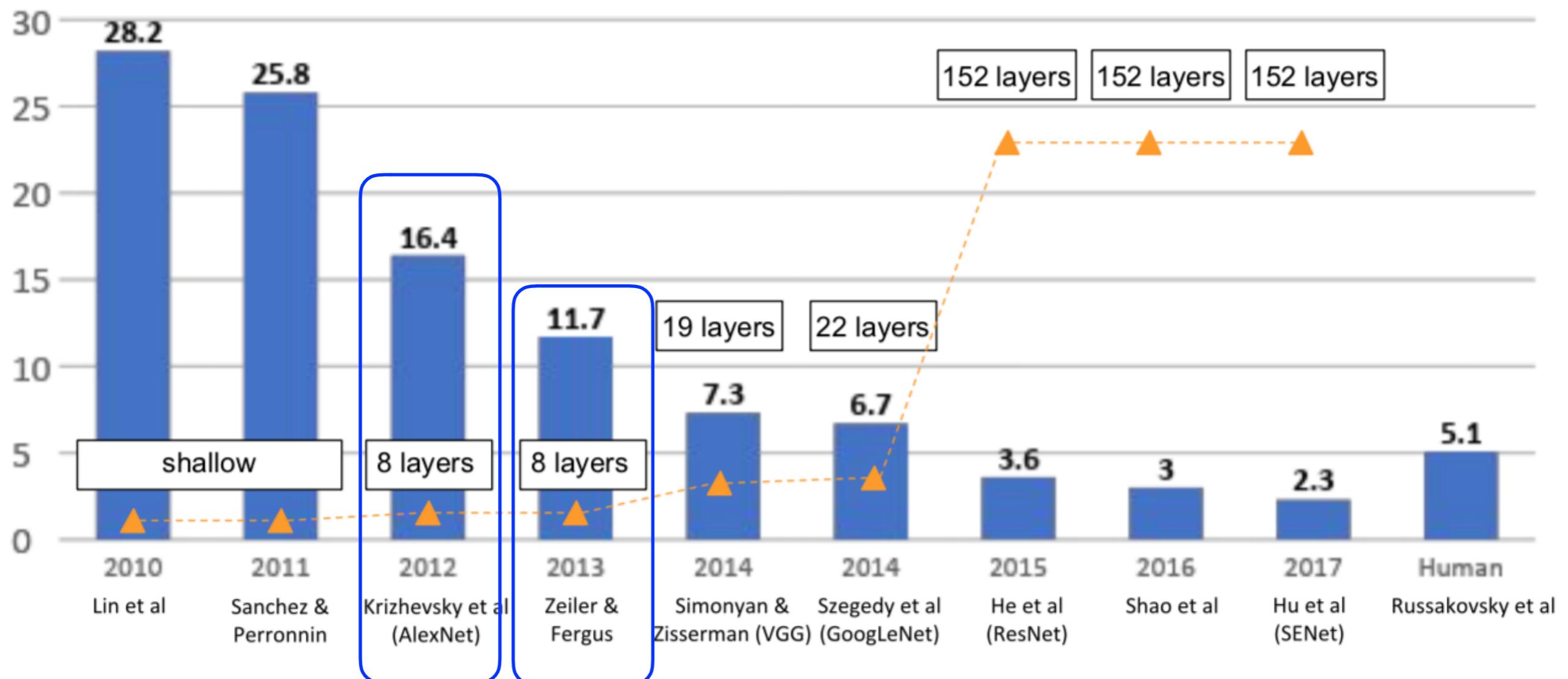
Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

CONV1, CONV2, CONV4, CONV5: Connections only with feature maps on same GPU

CONV3, FC6, FC7, FC8: Connections with all feature maps in preceding layer, communication across GPUs

Total of 60M parameters !

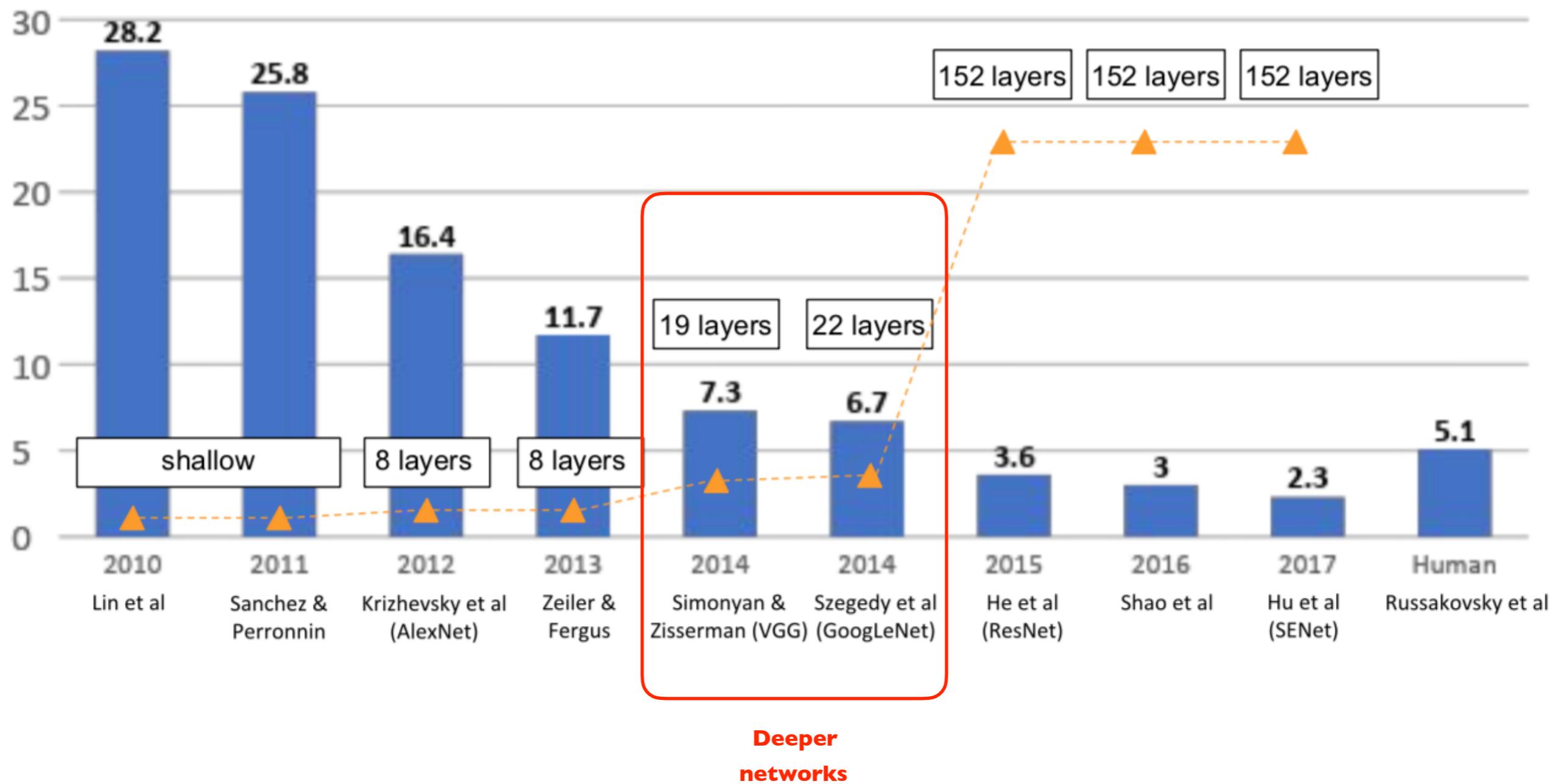
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) - winners



AlexNet : First deep learning based approach winner of the competition.

ZFNet : similar to AlexNet but with hyper-parameter tuning.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) - winners



VGGNet [2014]

Small filters, Deeper networks

From 8 layers (AlexNet) to 16 - 19 layers (VGGNet)

Only 3x3 CONV S=1, P=1 and 2x2 MAX POOL S=2

7.3% top 5 error in ILSVRC'14

Stack of three 3x3 conv (stride 1) layers has same effective receptive field as one 7x7 conv layer.

But deeper, more non-linearities.

And fewer parameters: $3 * (32C2)$ vs. $72C2$ for C channels per layer



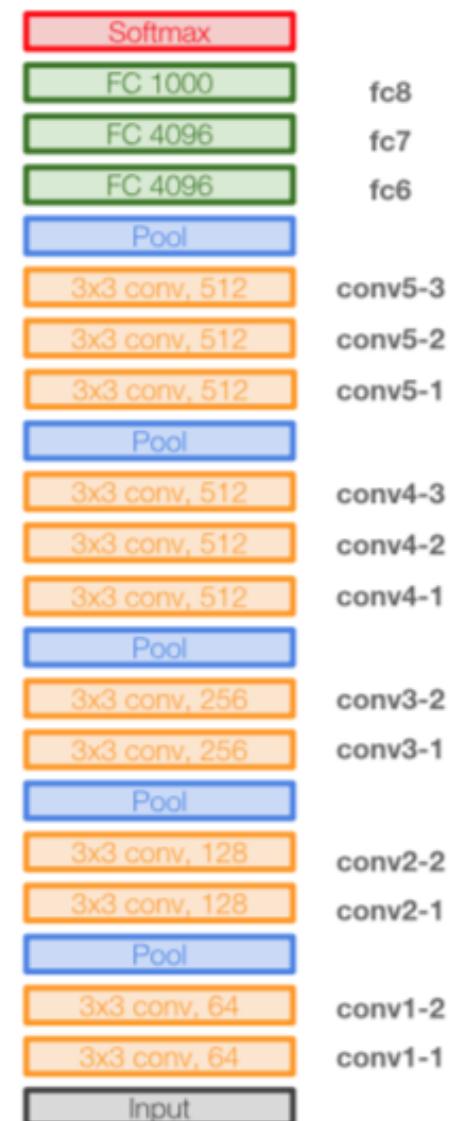
- Simonyan and Zisserman, 2014

VGGNet [2014]

INPUT: [224x224x3]	memory: $224 \times 224 \times 3 = 150K$	params: 0 (not counting biases)
CONV3-64: [224x224x64]	memory: $224 \times 224 \times 64 = 3.2M$	params: $(3 \times 3 \times 3) \times 64 = 1,728$
CONV3-64: [224x224x64]	memory: $224 \times 224 \times 64 = 3.2M$	params: $(3 \times 3 \times 64) \times 64 = 36,864$
POOL2: [112x112x64]	memory: $112 \times 112 \times 64 = 800K$	params: 0
CONV3-128: [112x112x128]	memory: $112 \times 112 \times 128 = 1.6M$	params: $(3 \times 3 \times 64) \times 128 = 73,728$
CONV3-128: [112x112x128]	memory: $112 \times 112 \times 128 = 1.6M$	params: $(3 \times 3 \times 128) \times 128 = 147,456$
POOL2: [56x56x128]	memory: $56 \times 56 \times 128 = 400K$	params: 0
CONV3-256: [56x56x256]	memory: $56 \times 56 \times 256 = 800K$	params: $(3 \times 3 \times 128) \times 256 = 294,912$
CONV3-256: [56x56x256]	memory: $56 \times 56 \times 256 = 800K$	params: $(3 \times 3 \times 256) \times 256 = 589,824$
CONV3-256: [56x56x256]	memory: $56 \times 56 \times 256 = 800K$	params: $(3 \times 3 \times 256) \times 256 = 589,824$
POOL2: [28x28x256]	memory: $28 \times 28 \times 256 = 200K$	params: 0
CONV3-512: [28x28x512]	memory: $28 \times 28 \times 512 = 400K$	params: $(3 \times 3 \times 256) \times 512 = 1,179,648$
CONV3-512: [28x28x512]	memory: $28 \times 28 \times 512 = 400K$	params: $(3 \times 3 \times 512) \times 512 = 2,359,296$
CONV3-512: [28x28x512]	memory: $28 \times 28 \times 512 = 400K$	params: $(3 \times 3 \times 512) \times 512 = 2,359,296$
POOL2: [14x14x512]	memory: $14 \times 14 \times 512 = 100K$	params: 0
CONV3-512: [14x14x512]	memory: $14 \times 14 \times 512 = 100K$	params: $(3 \times 3 \times 512) \times 512 = 2,359,296$
CONV3-512: [14x14x512]	memory: $14 \times 14 \times 512 = 100K$	params: $(3 \times 3 \times 512) \times 512 = 2,359,296$
CONV3-512: [14x14x512]	memory: $14 \times 14 \times 512 = 100K$	params: $(3 \times 3 \times 512) \times 512 = 2,359,296$
POOL2: [7x7x512]	memory: $7 \times 7 \times 512 = 25K$	params: 0
FC: [1x1x4096]	memory: 4096	params: $7 \times 7 \times 512 \times 4096 = 102,760,448$
FC: [1x1x4096]	memory: 4096	params: $4096 \times 4096 = 16,777,216$
FC: [1x1x1000]	memory: 1000	params: $4096 \times 1000 = 4,096,000$

Most memory usage in
CONV layers

Most parameters in FC layers



VGG16

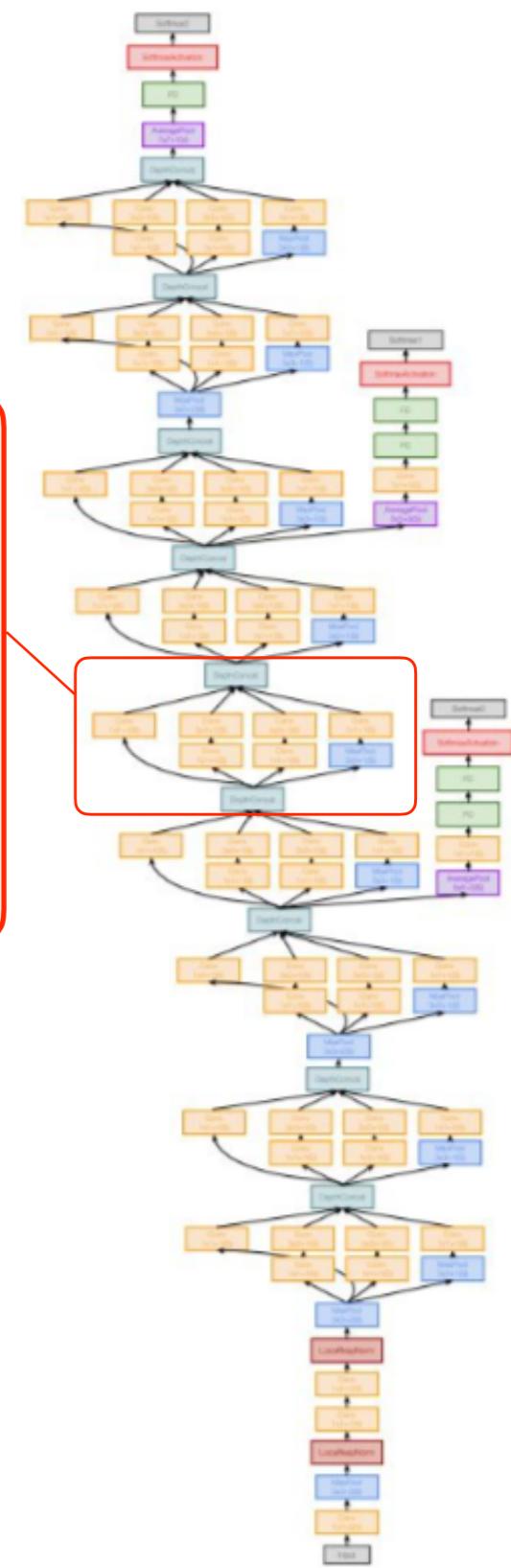
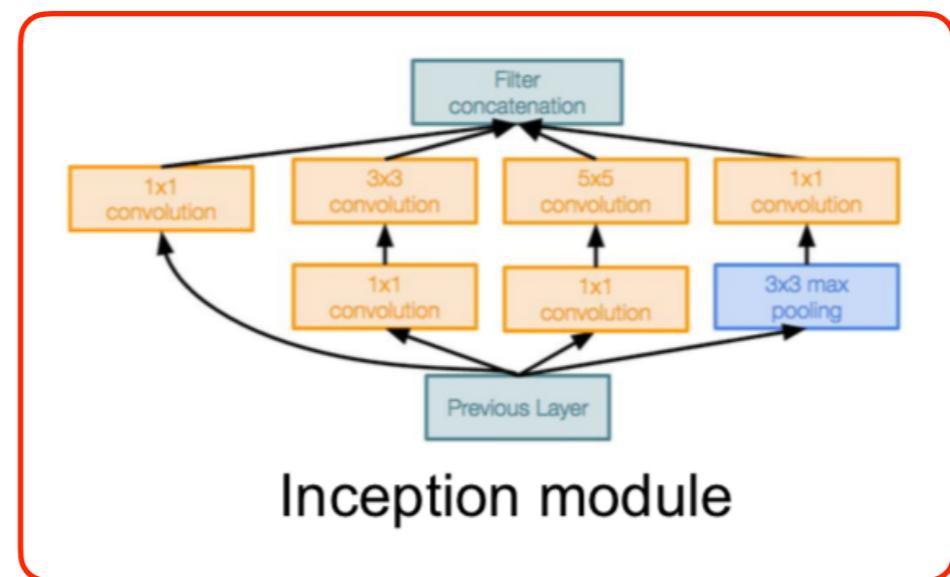
Common names

TOTAL memory: $24M * 4 \text{ bytes} \approx 96\text{MB} / \text{image}$ (only forward! ~ 2 for bwd)
TOTAL params: 138M parameters

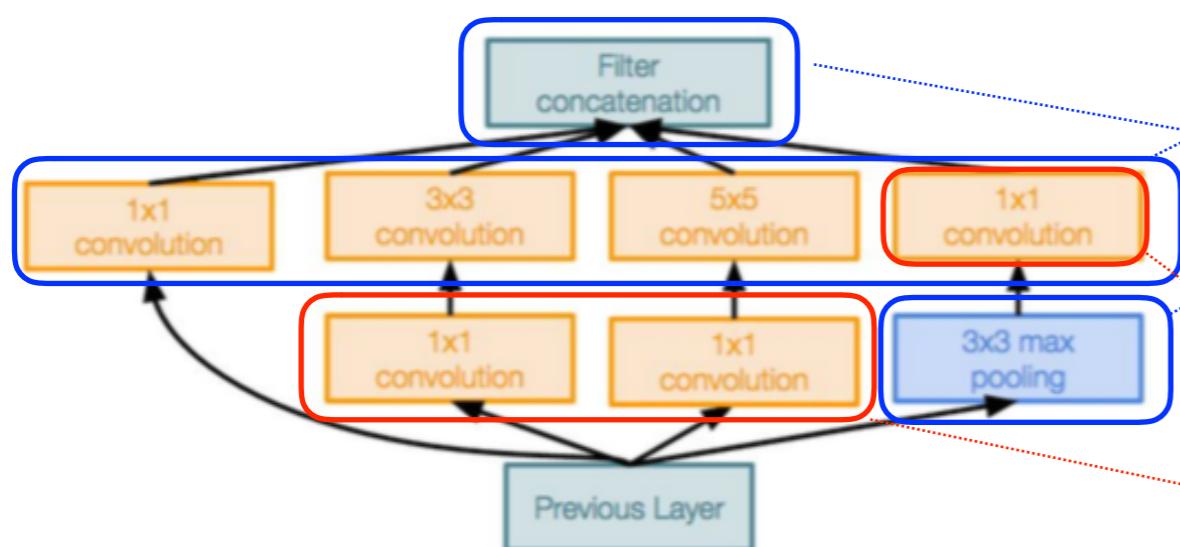
GoogLeNet [2014]

Deeper networks, with computational efficiency

- 22 layers
- "Network in the network" efficient "**Inception**" module
- No FC layers
- Only 5 million parameters!
12x less than AlexNet
- ILSVRC'14 classification winner
(6.7% top 5 error)
- Szegedy et al., 2014



GoogLeNet [2014]



Inception module

Intuition: filter size allows to look for more or less details from the previous layer, i.e. related to the receptive field size. We don't know in advance what is the best size, e.g. close or far object.

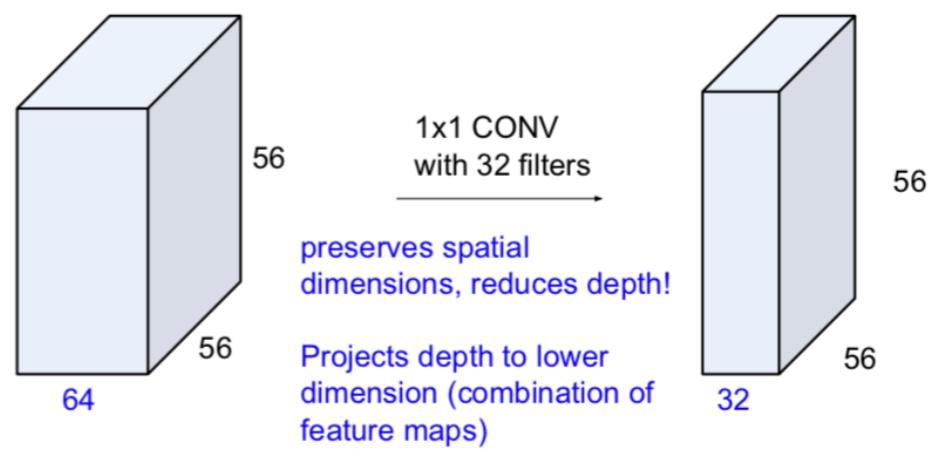
So let's use **in parallel** different filter size, (1x1, 3x3, 5x5).

Ultimately no filter but pooling operation (3x3).

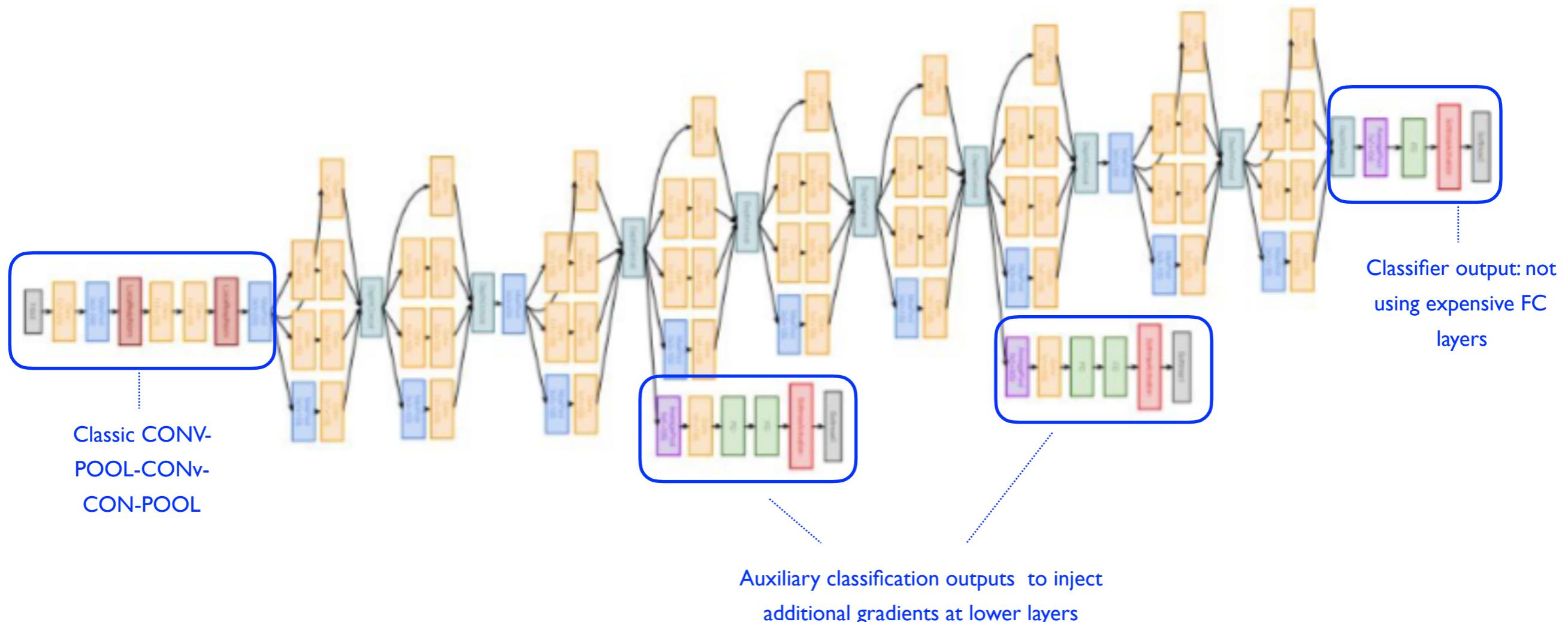
Then concatenate all filter outputs together depth-wise.

To reduce the number of operation, 1x1 conv are used. So-called **bottleneck layers**.

Reminder: 1x1 convolutions

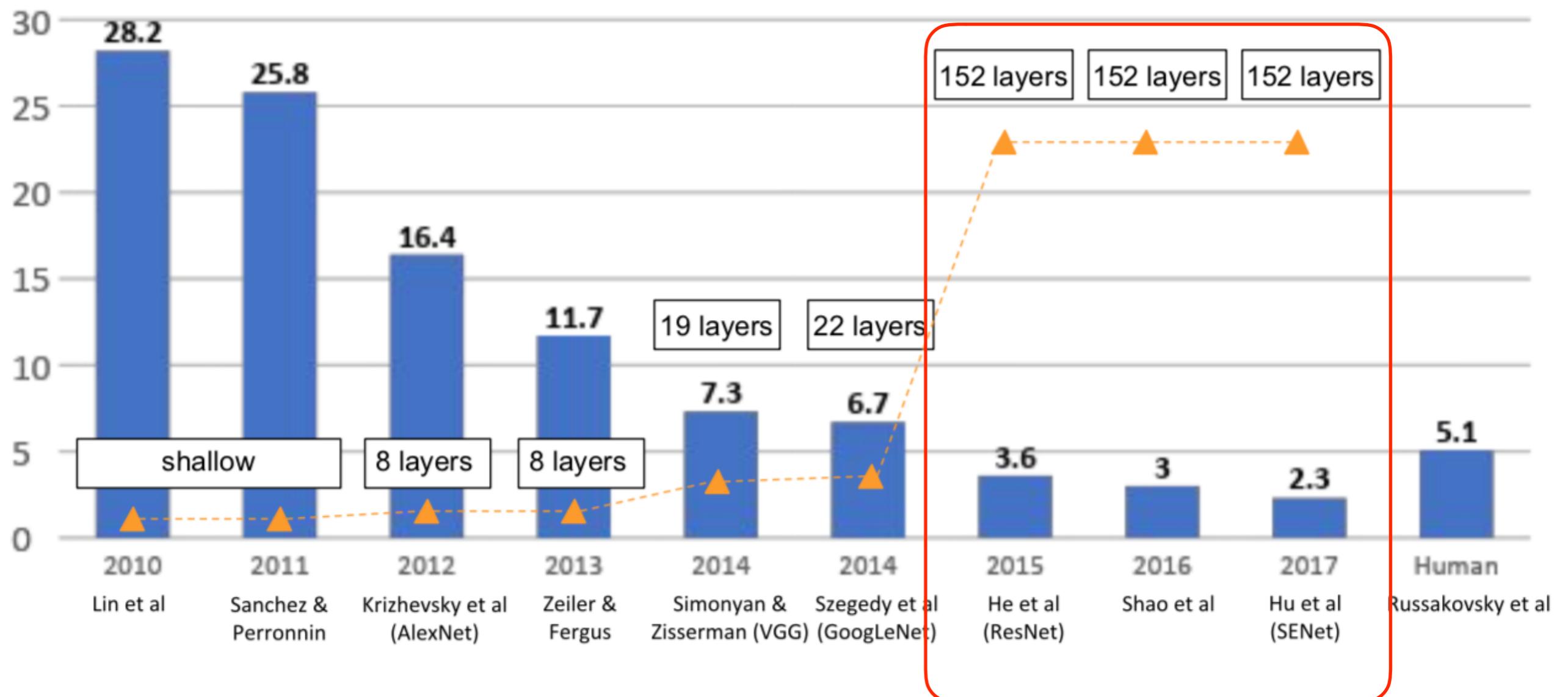


GoogLeNet [2014]



The architecture becomes non-sequential and complex !

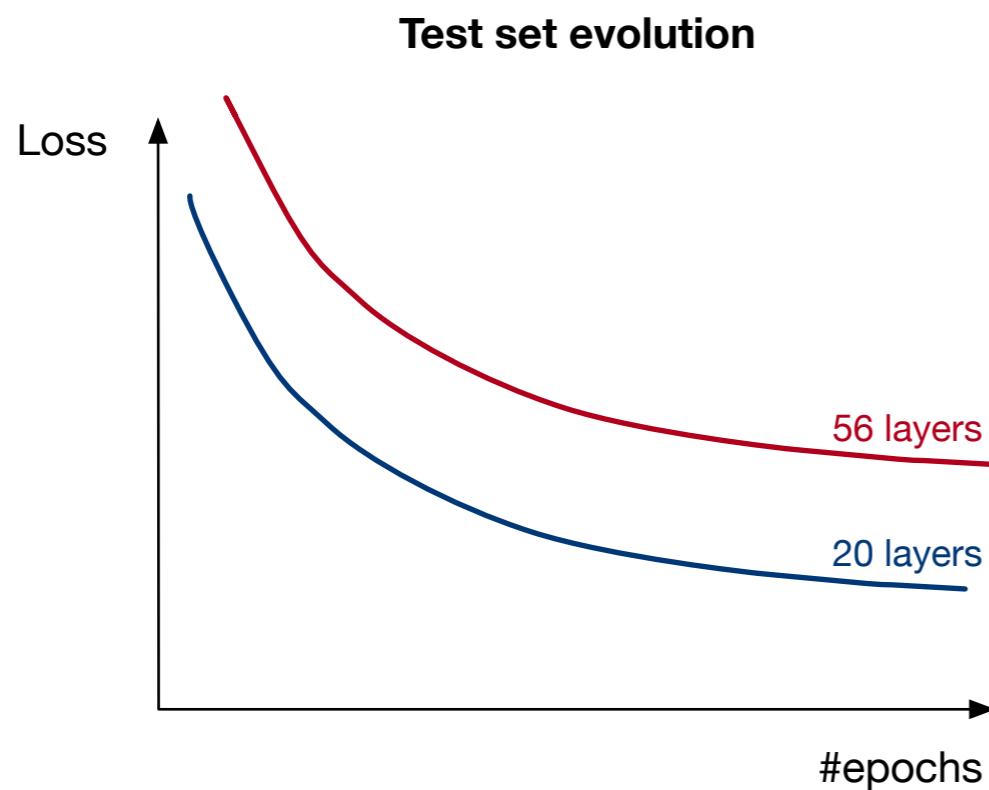
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) - winners



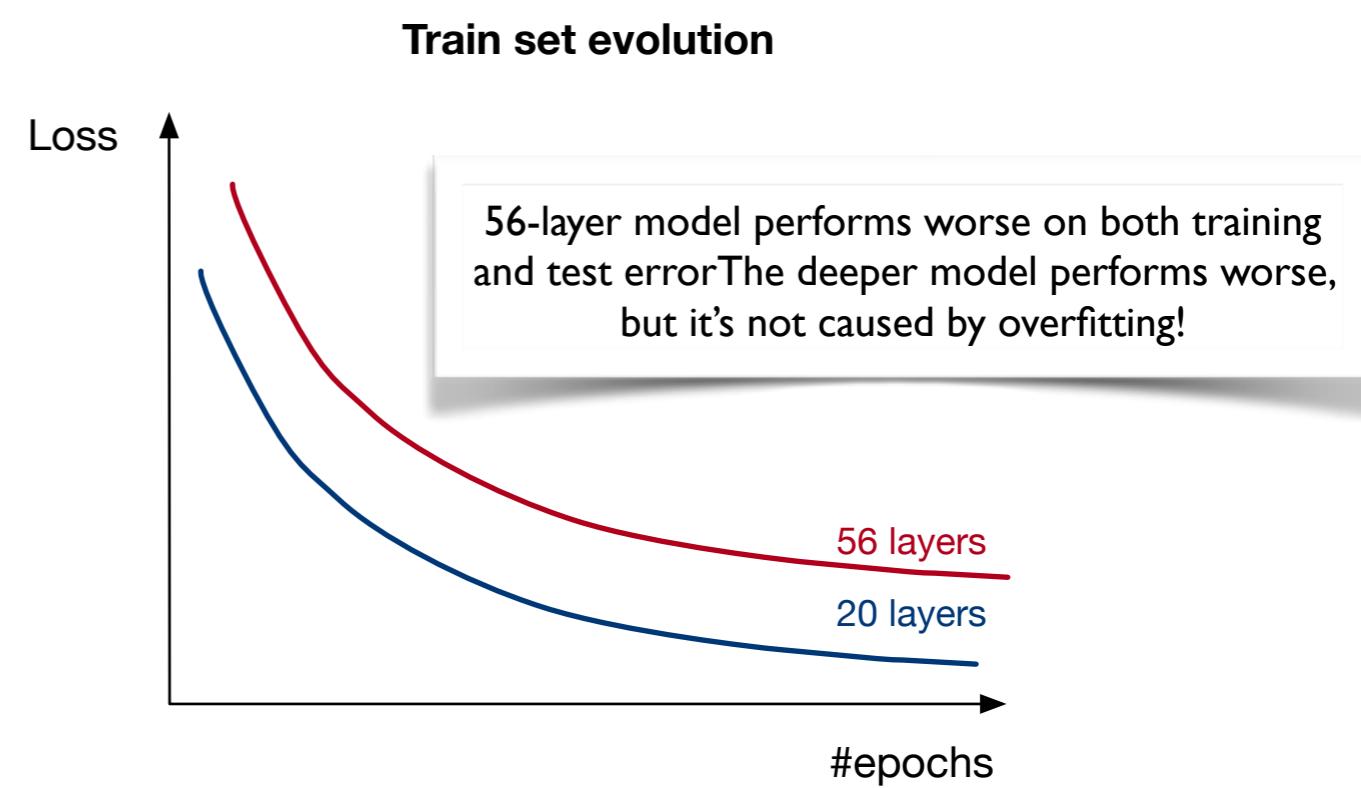
Revolution of
depth !

ResNet [2015]

- He et al., 2015
- Observation they did: from AlexNet and VGGNet, increasing the number of layers did not succeed. They observed something like this:

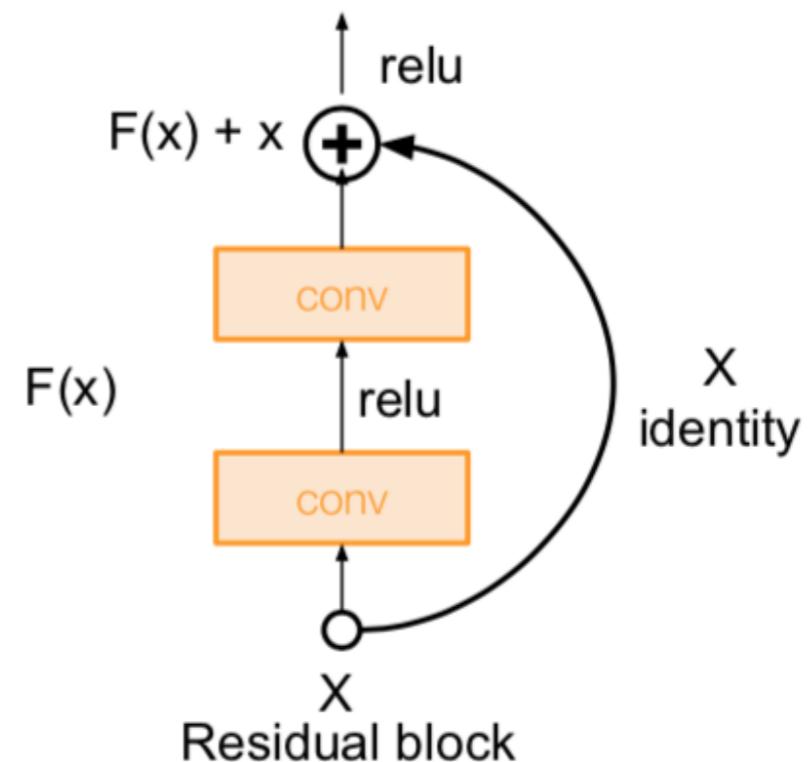


Overfitting?



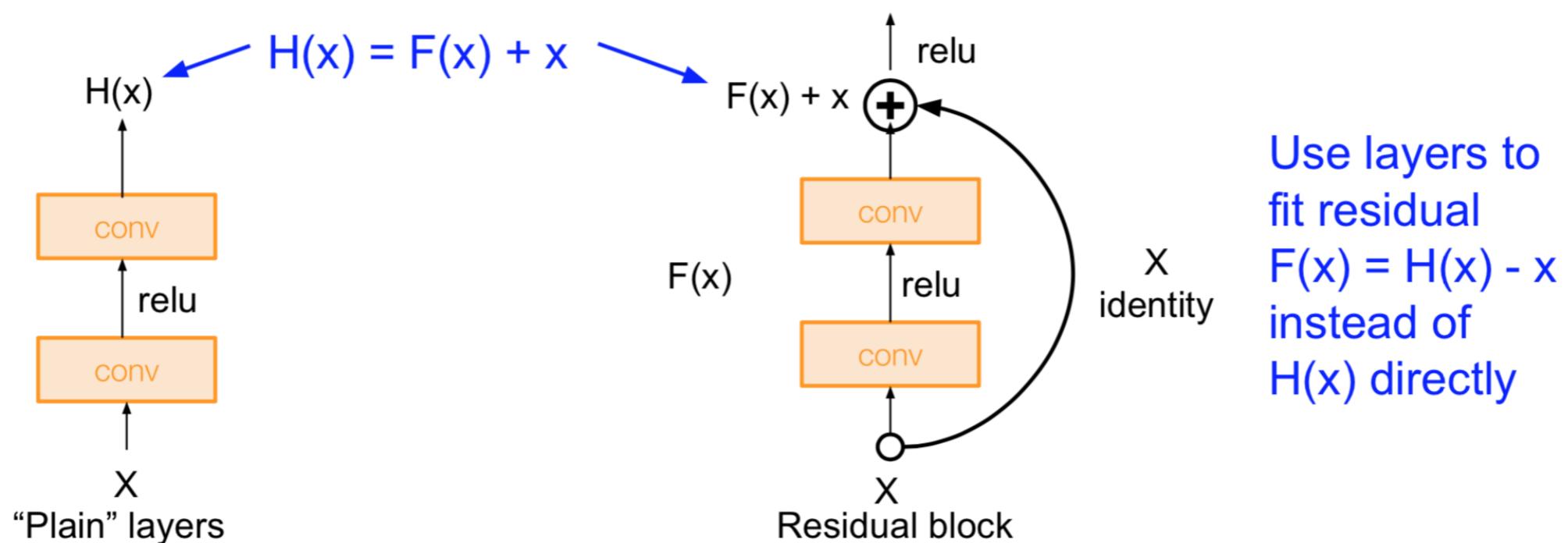
ResNet [2015]

- Hypothesis: the problem is an optimization problem, deeper models are more difficult to optimize
- Ideas:
 - Deeper layers should be able to “fall back” to identity mapping if difficulties to converge
 - Easier to model a “delta” from one layer to the other than a full feature



ResNet [2015]

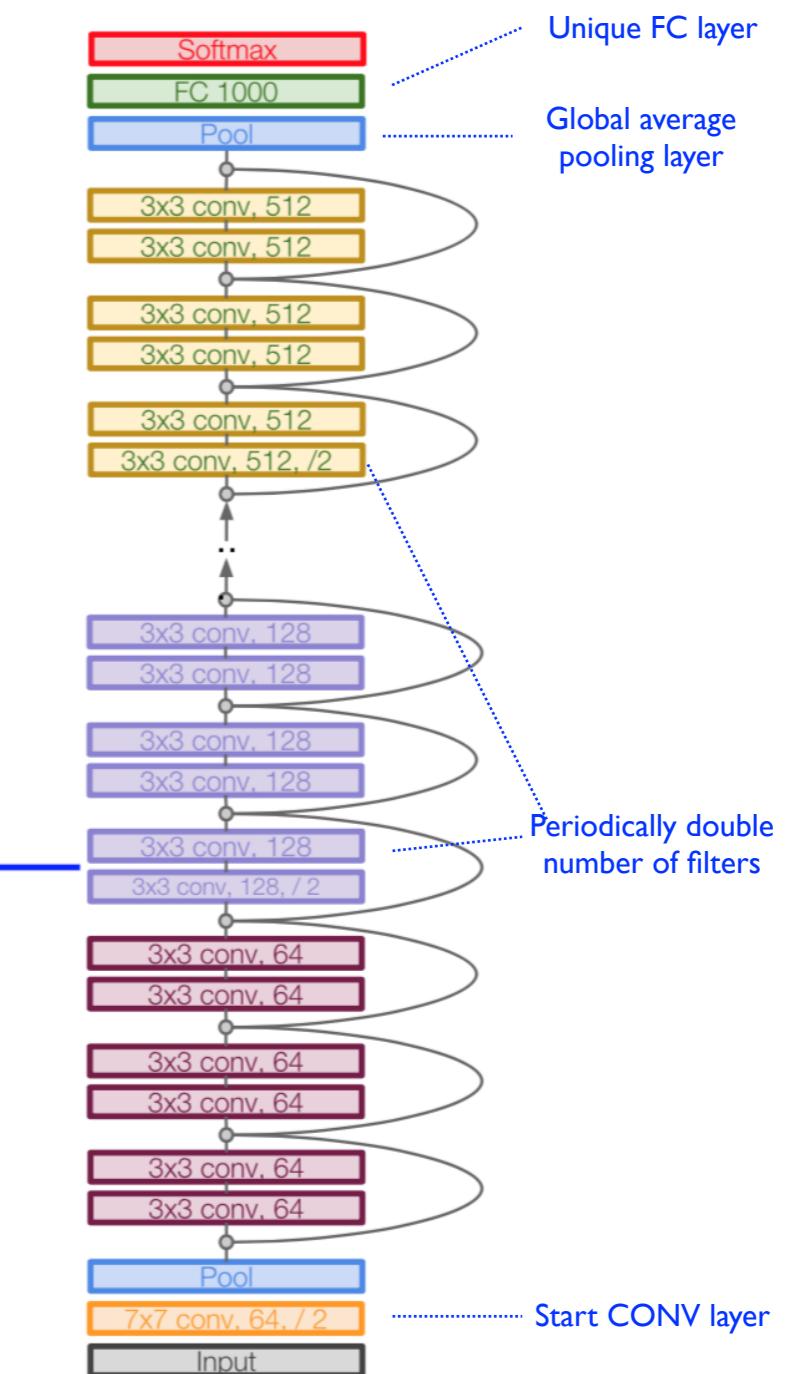
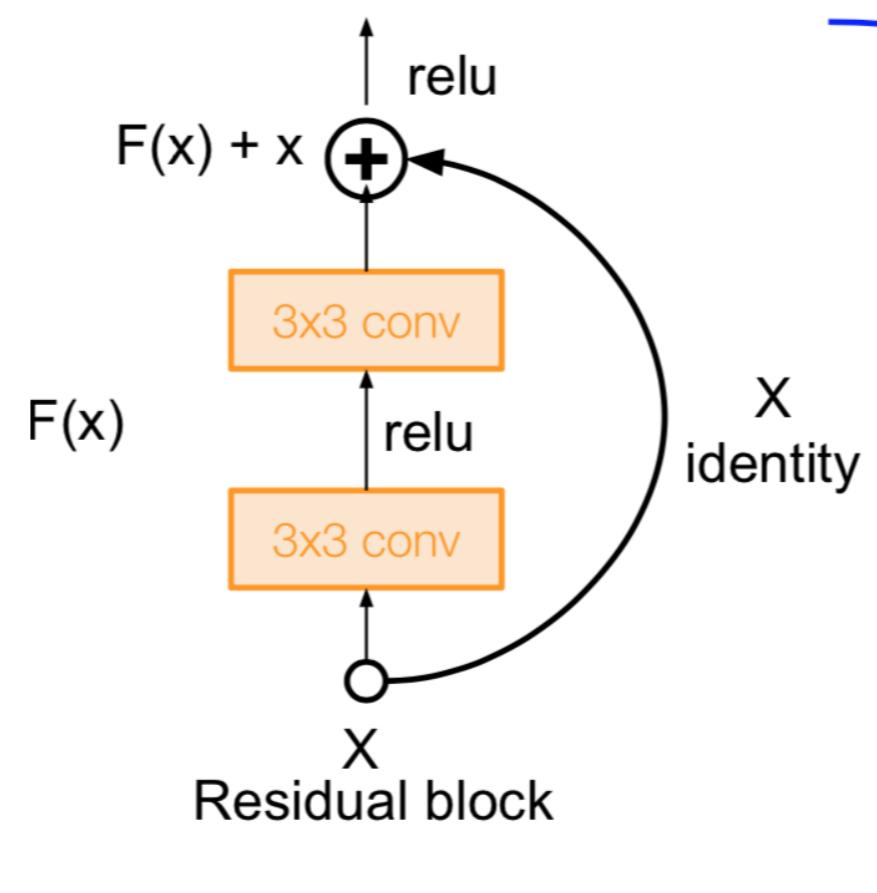
- **Hypothesis:** the problem is an optimization problem, deeper models are more difficult to optimize
- **Ideas:**
 - Deeper layers should be able to “fall back” to identity mapping if difficulties to converge
 - Easier to model a “delta” from one layer to the other than a full feature
- **Solution:** Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



ResNet [2015]

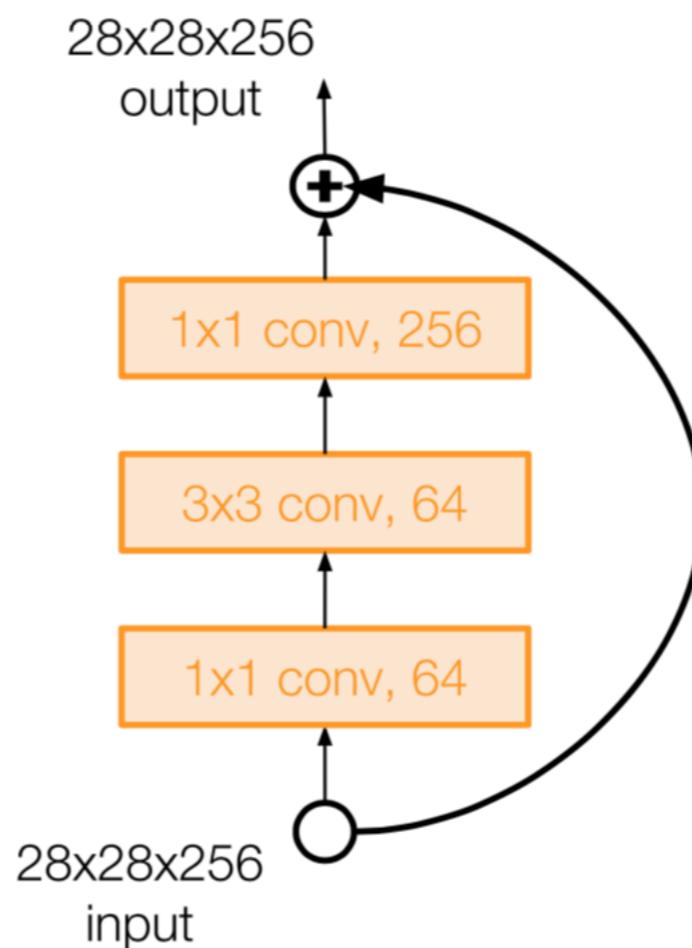
Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!

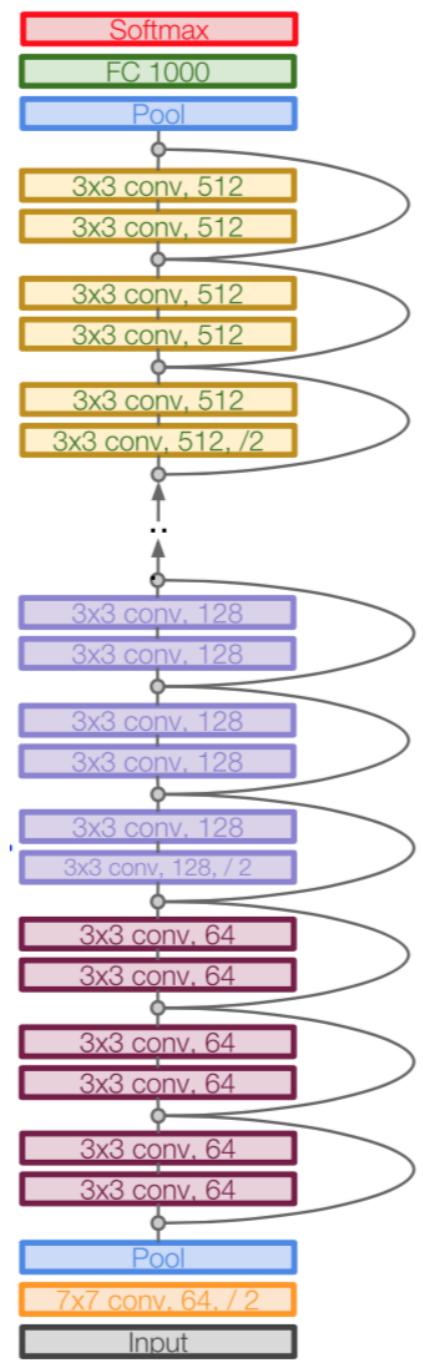


ResNet [2015]

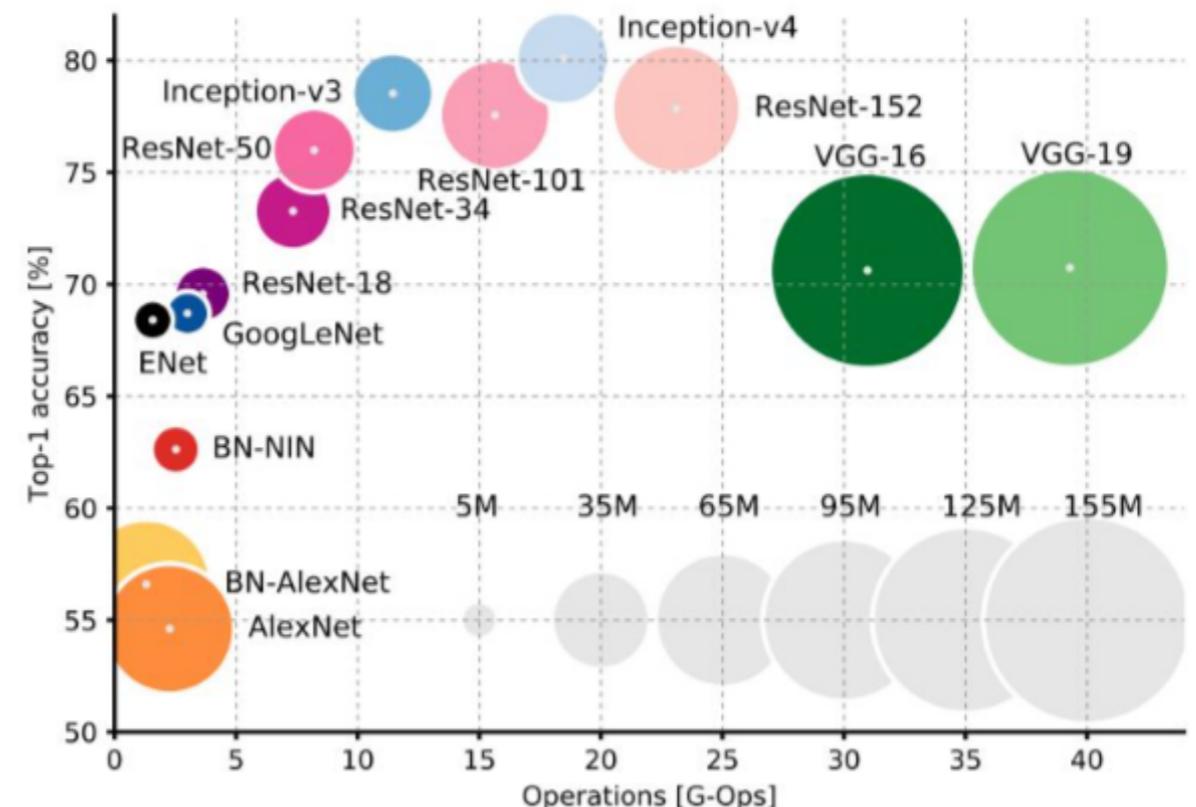
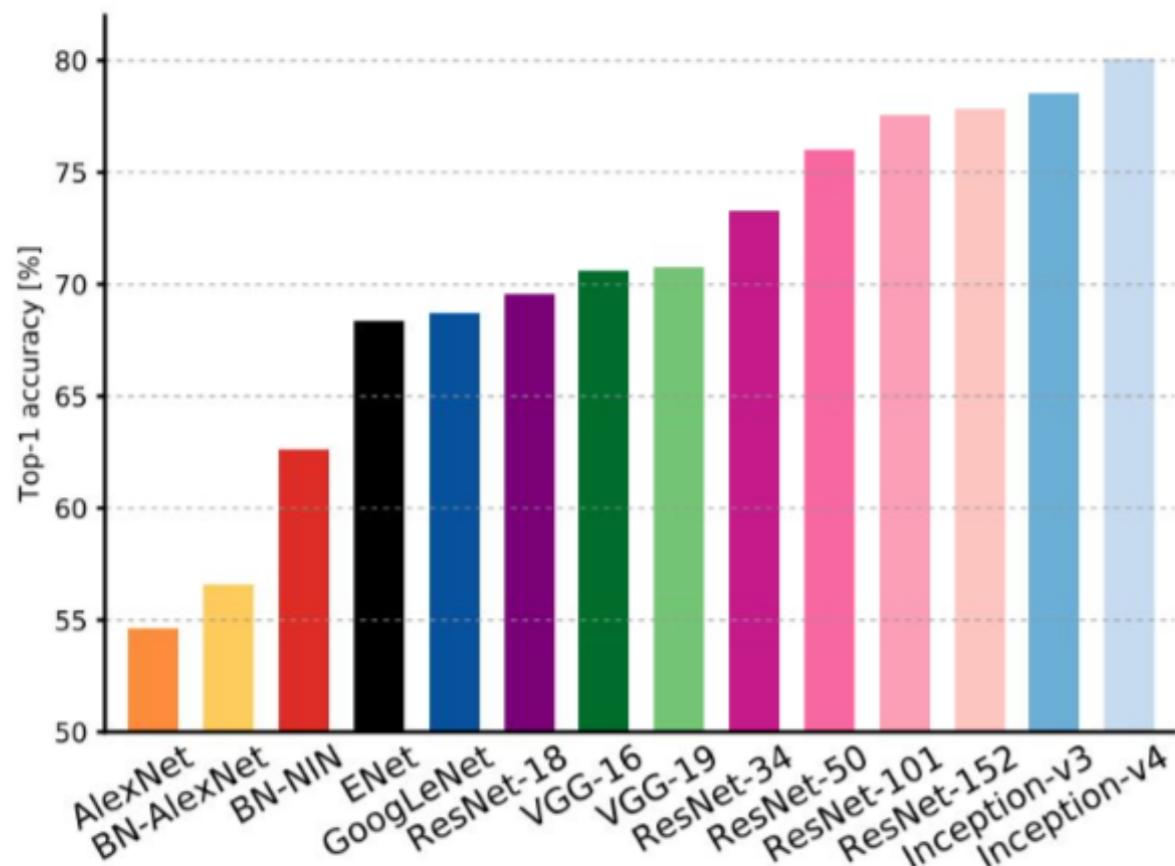
- **Trick to go deeper than 50 layers:**
 - Use bottleneck layers $1 \times 1 \times 64$ to reduce the number of operations and parameters
- **Other training strategies:**
 - Batch Normalization after every CONV layer
 - Xavier 2/ initialization from He et al.
 - SGD + Momentum (0.9)
 - Learning rate: 0.1, divided by 10 when validation error plateaus
 - Mini-batch size 256
 - Weight decay of $1e-5$
 - No dropout used



Total depth = 152 layers for imageNet competition !



Comparing complexity



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Wrap-up

- CNNs are learning the **feature extraction**
 - Notion of hierarchical features going deeper in the network
- Visualisation strategies:
 - **Plot of activation maps** at each layer
 - Finding **images** that are **maximising the activation** of a given neuron
- Data augmentation is a good strategy to **fight overfitting**
 - 2 implementation strategies: off-line and on-line
- **Deep CNN architectures**
 - Big fights were going on in the ImageNet competitions to discover the best architectures
 - AlexNet: the “boot” of deep architectures
 - VGGNet: going deeper with simpler filter
 - GoogLeNet: inception module = network in network to get features at different resolutions in a given layer, injection of gradient at early stages
 - ResNet: going way deeper using modules that model residual information from layer to layer.

