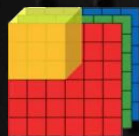
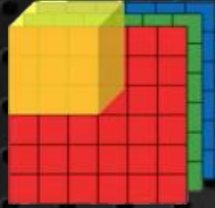
AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering
Department of Biocybernetics and Biomedical Engineering

Convolutional Neural Networks



Adrian Horzyk
horzyk@agh.edu.pl

Google: Adrian Horzyk



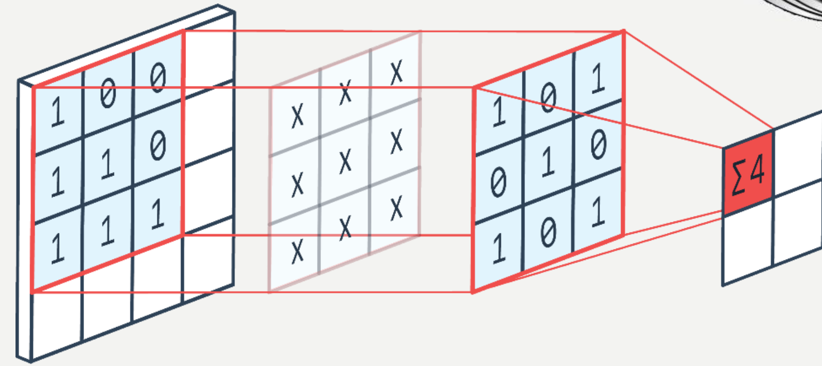
Convolutional Neural Network

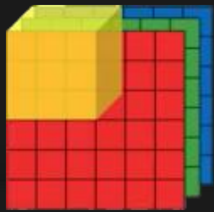
What is specific in this kind of neural networks?

Where we use them and to what data?

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are very popular today thanks to special convolution operations based on adaptive filtering, which work well, especially with images:





CONV RELU CONV RELU POOL CONV RELU CONV RELU POOL CONV RELU CONV RELU POOL FC

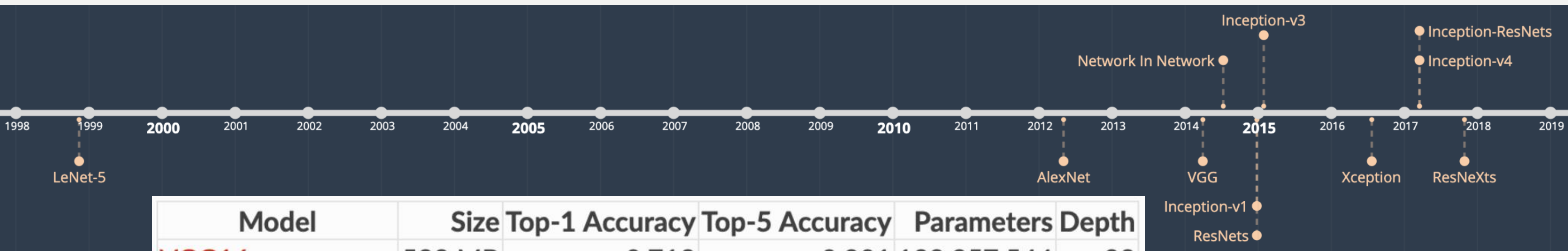
car
truck
airplane
ship
horse

Benefits of using CNNs

Convolutional Neural Networks:

- **Share parameters** - so **the same features may be recognized in any part of the image**!
- **Use sparse connections**, so the convolutional layers are not connected in all-to-all manner (densely/fully-connected), which saves a lot of parameters and allows to train the network faster.
- Outputs depend directly only on some **selected areas** of the input images, so the neurons can specialize in recognizing, but their position in the convolutional layer defines the location where the features have been found.

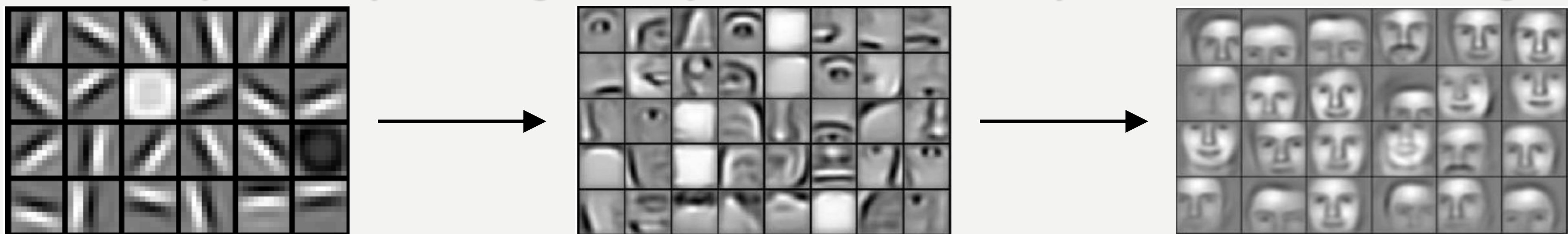
Timeline of the development of Convolutional Neural Networks:



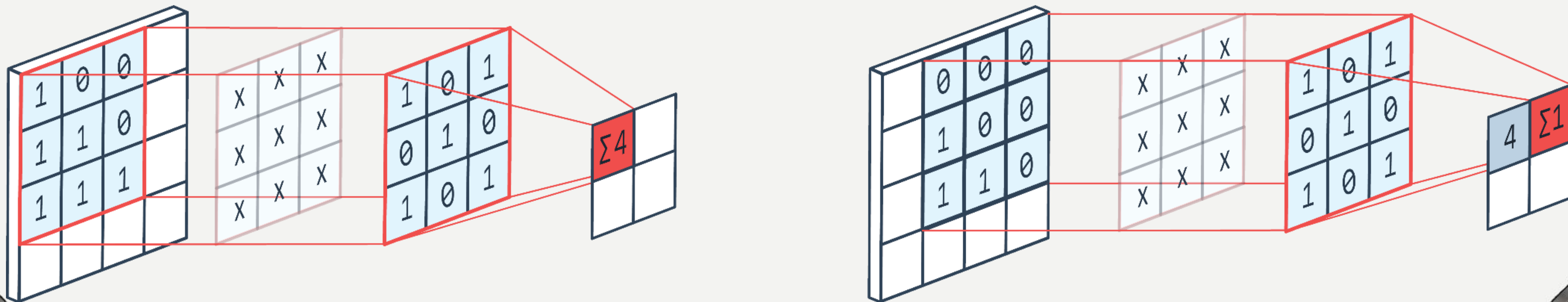
Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
VGG16	528 MB	0.713	0.901	138,357,544	23
InceptionV3	92 MB	0.779	0.937	23,851,784	159
ResNet50	98 MB	0.749	0.921	25,636,712	-
Xception	88 MB	0.790	0.945	22,910,480	126
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
ResNeXt50	96 MB	0.777	0.938	25,097,128	-

Computer Vision

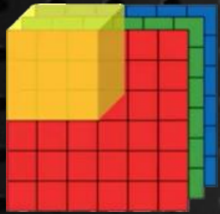
Computer vision (CV) is an interdisciplinary scientific field that deals with how computers can perform various tasks on objects in digital images/videos and automate tasks which the human visual system can do. CV plays a very important role today and can be supported by convolutional neural networks (CNN) due to their unique ability to recognize objects whenever they are located in the image:



Convolutional filters allow us to detect and filter out basic and secondary features gradually in the subsequent layers of the network using adaptive filtering (dot products) where weights of the adaptive filters are adjusted during the CNN training process:



The network adjusts the filters to recognize particular shapes and colors, which are frequent and form patterns that may be adapted many times to various images.



Convolutions and Adaptive Filtering

Why we use convolutions and convolutional neural networks, and why their use is beneficial?

Filters and Convolutions

Filters are commonly used in computer graphics, and allow us to find edges and convolve images:

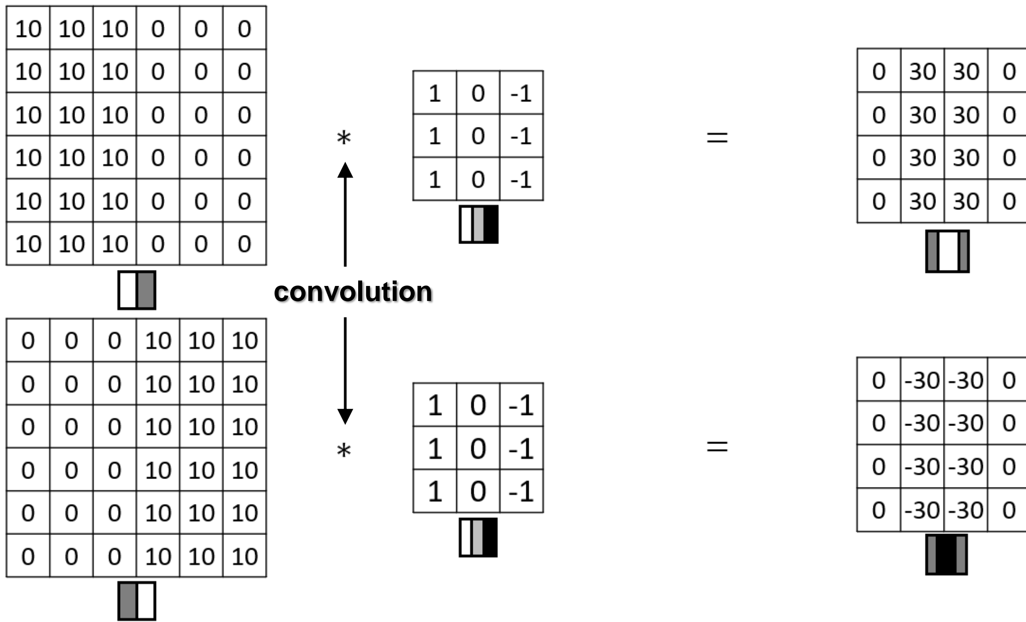
1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

Horizontal

The example result of applying the vertical-line filter:



Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Adaptive Filtering

In convolutional layers, we use adaptive filters, which are composed of non-constant values that we call weights w_i which are adapted during the training process to represent frequent patterns of the filter size in the input images:

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

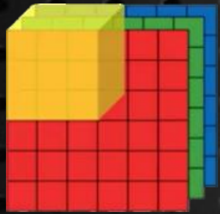
* convolution

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

=

The output value is computed as a dot product of the input area and the filter (an array of the adaptable weights) where the filter is adapted in the input image.

Convolutional weights are parameters of the model, so they are adjusted during the training process to filter out the most frequent features found in the data (training examples).

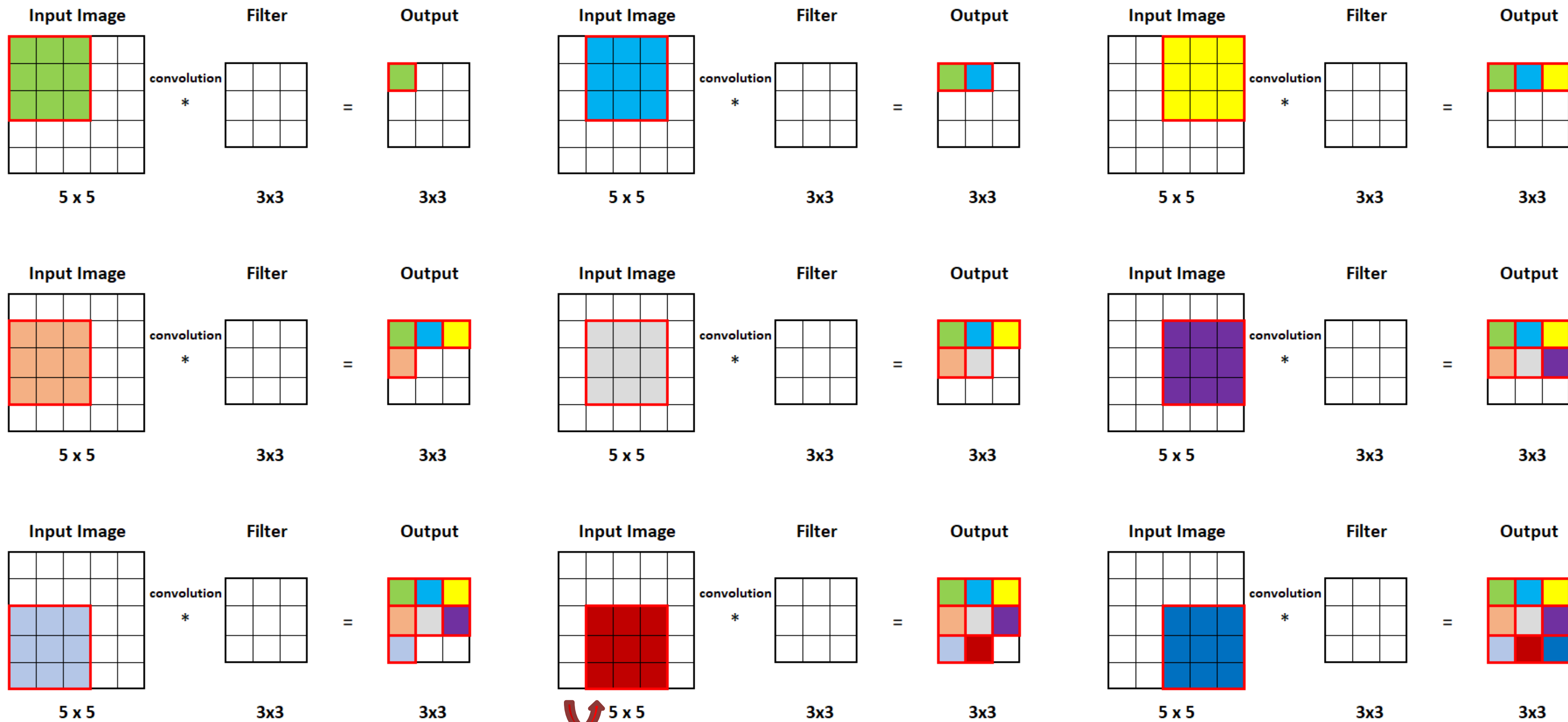


Operations on Filters

How do we use operations on filters, and what do they produce?

Stride 1

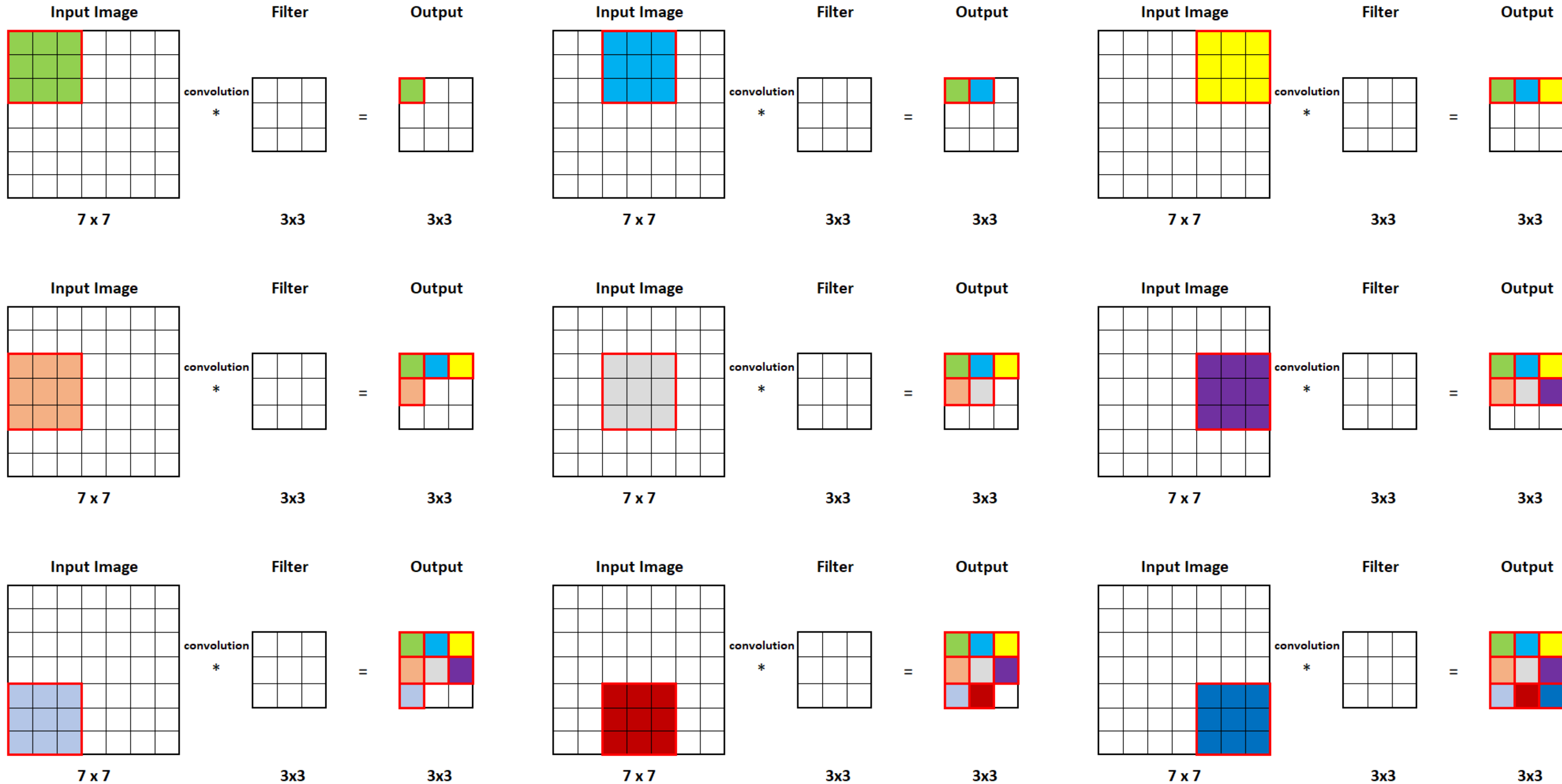
To adapt the filter to the whole image, we must move the filter over the image with a given **stride s** that defines the number of fields (pixels) we move in vertical and horizontal directions (it is a hyperparameter of the model):



For **stride 1**, we jump over one pixel as presented in the figure above.

Stride 2

For stride 2, we jump over two pixels as presented in the figure below:

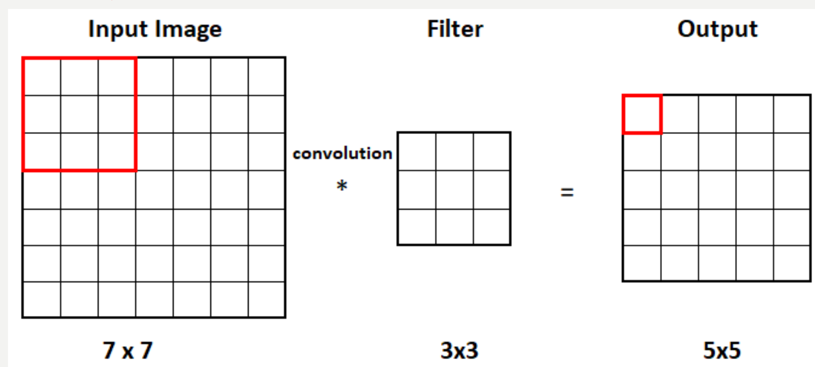


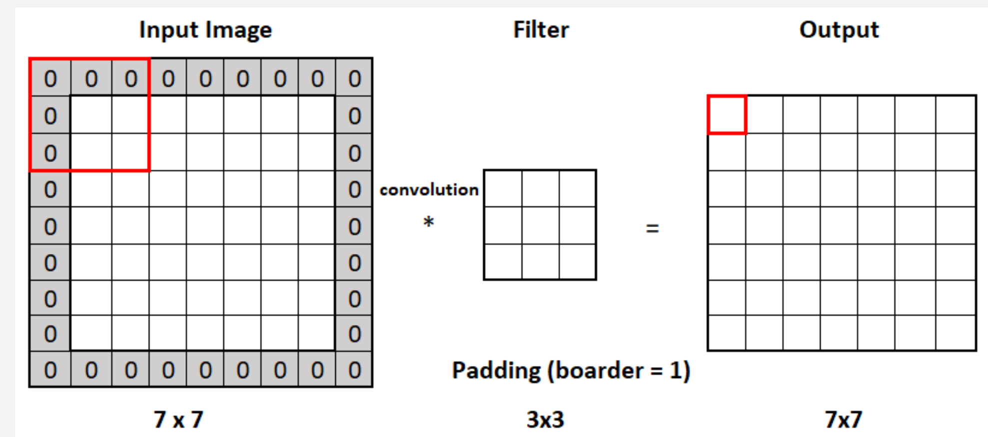
The chosen stride value is one of the **hyperparameters** of the model!

Padding

When moving the filter (f x f) over the image (n x n) with a given stride, we cannot move over the edges/border of the image, so we are forced to treat the pixels on the borders in a different way ("Valid") or add a 0-value border outside the image to adapt filters on the boarders ("Same"):

- Valid Convolution (no padding): Output size is n x n * f x f = (n – f + 1) x (n – f + 1)





- Same Convolution (padding balances the filter size p = (f – 1)/2, then the output size is the same as the one of the input image.
- The chosen way of convolution ("same" or "valid") is one of the hyperparameters of the model!

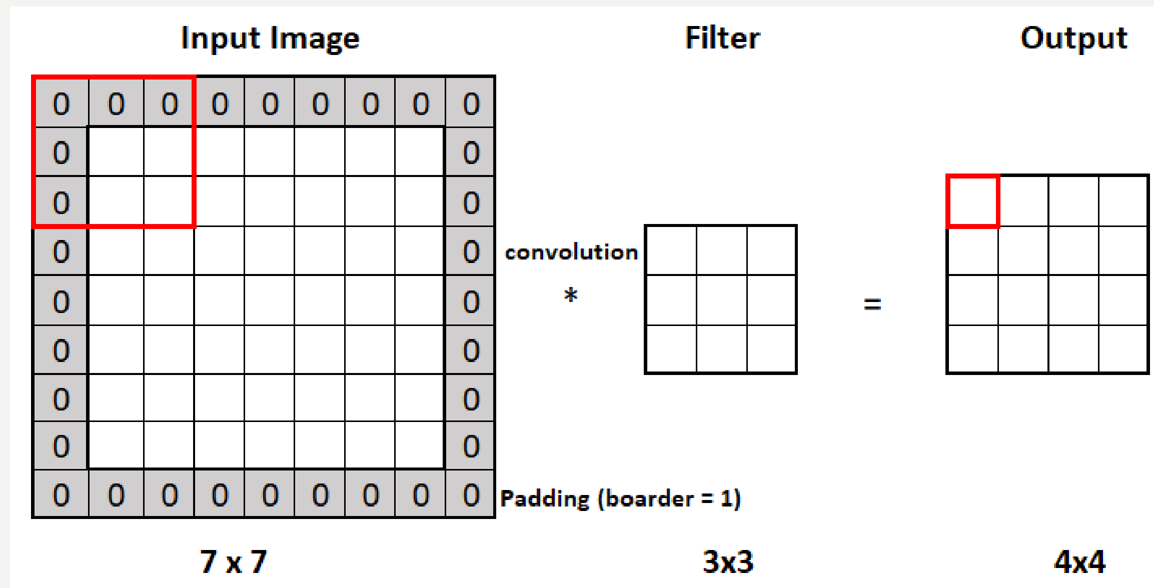
Output Volume Size Calculation

The output array size can be computed for given hyperparameters:

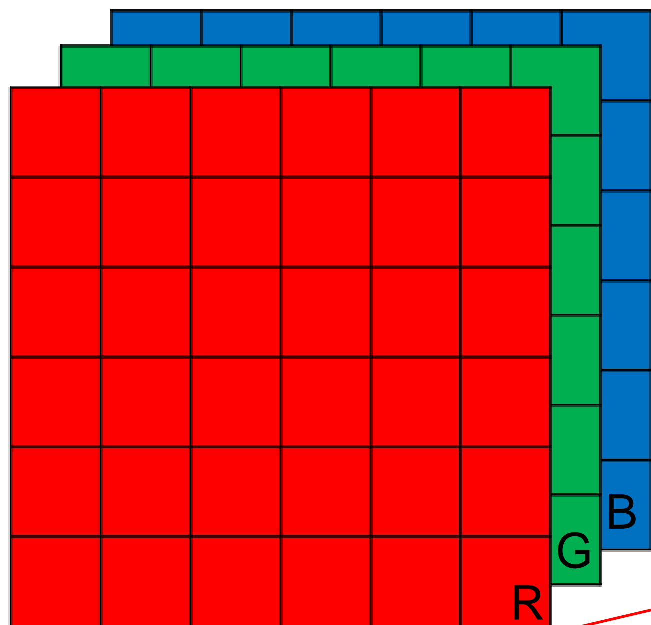
- **Input matrix (image) dimension *n x n***
- **Filter size *f x f***
- **Stride *s***
- **Padding *p***

in the following way: $\left[\frac{n+2p-f}{s}+1\right] \times \left[\frac{n+2p-f}{s}+1\right]$

Example for n = 7, f = 3, s = 2, p = 1: $\left[\frac{7+2\cdot 1-3}{2}+1\right] \times \left[\frac{7+2\cdot 1-3}{2}+1\right] = \mathbf{4 \times 4}$



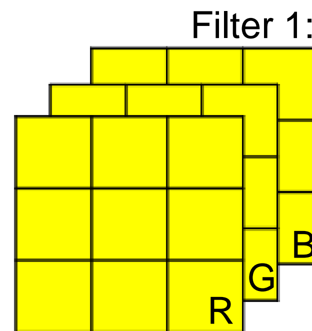
Multiple Adaptive Filters on RGB Images



6 x 6 x 3

$*$

Filter 1:

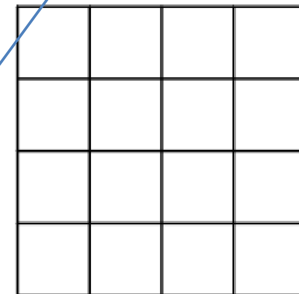


3 x 3 x 3

$=$

Number of channels (filters or depth of the conv. layer):

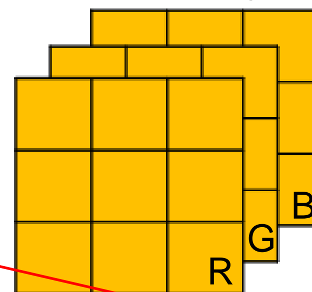
$n_c = 2$



4 x 4

Filter 2:

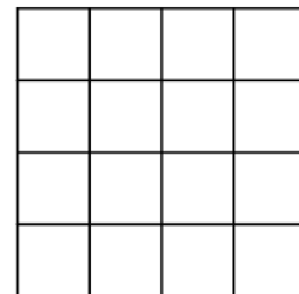
$*$



3 x 3 x 3

$n_c = 2$

$=$



4 x 4

Output Volume Size =

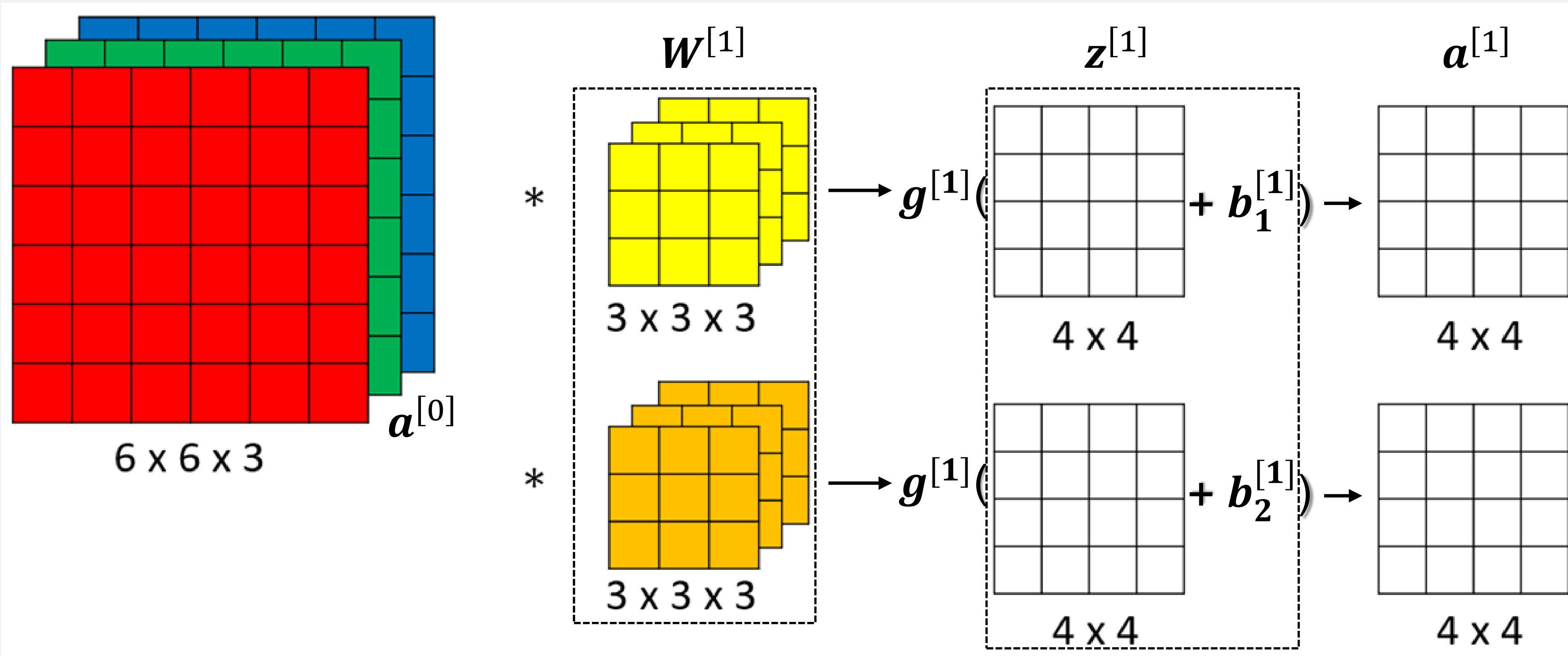
$$\left[\frac{n+2p-f}{s}+1\right] \times \left[\frac{n+2p-f}{s}+1\right] \times n_c$$

If the input image has 3 color channels, then the filters must also have the depth equal to 3, so we always convolve over the whole volume.

Convolutions and Convolutional Layer

What happens in the convolutional layer?

Input $a^{[0]}$ is convolved by the convolutional filters $W^{[1]}$ and adding bias $b^{[1]}$ and using activation function $g^{[1]}$ output $a^{[1]}$ is computed (here, two filters are used):



Number of parameters = (number of weights + bias) * number of filters = (3x3x3 + 1) * 2 = 28 * 2 = 56

$$z^{[1]} = W^{[1]} \cdot a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]}\left(z^{[1]}\right)$$

Convolutional Layer Notation

For convolutional layer l, we will use the following notations:

$f^{[l]}$ - filter size

$p^{[l]}$ - padding

$s^{[l]}$ - stride

$n_H^{[l]}$ - height (vertical dimension)

$n_W^{[l]}$ - width (horizontal dimension)

$n_c^{[l]}$ - number of channels or filters (depth of the layer)

For a given input:

$$n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$$

we get the following filter size:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$$

and weight size:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$$

and the output:

$$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]} = \left[\frac{n_H^{[l-1]} + 2 \cdot p^{[l]} - f^{[l]}}{s^{[l]}} + 1\right] \times \left[\frac{n_W^{[l-1]} + 2 \cdot p^{[l]} - f^{[l]}}{s^{[l]}} + 1\right] \times n_c^{[l]}$$

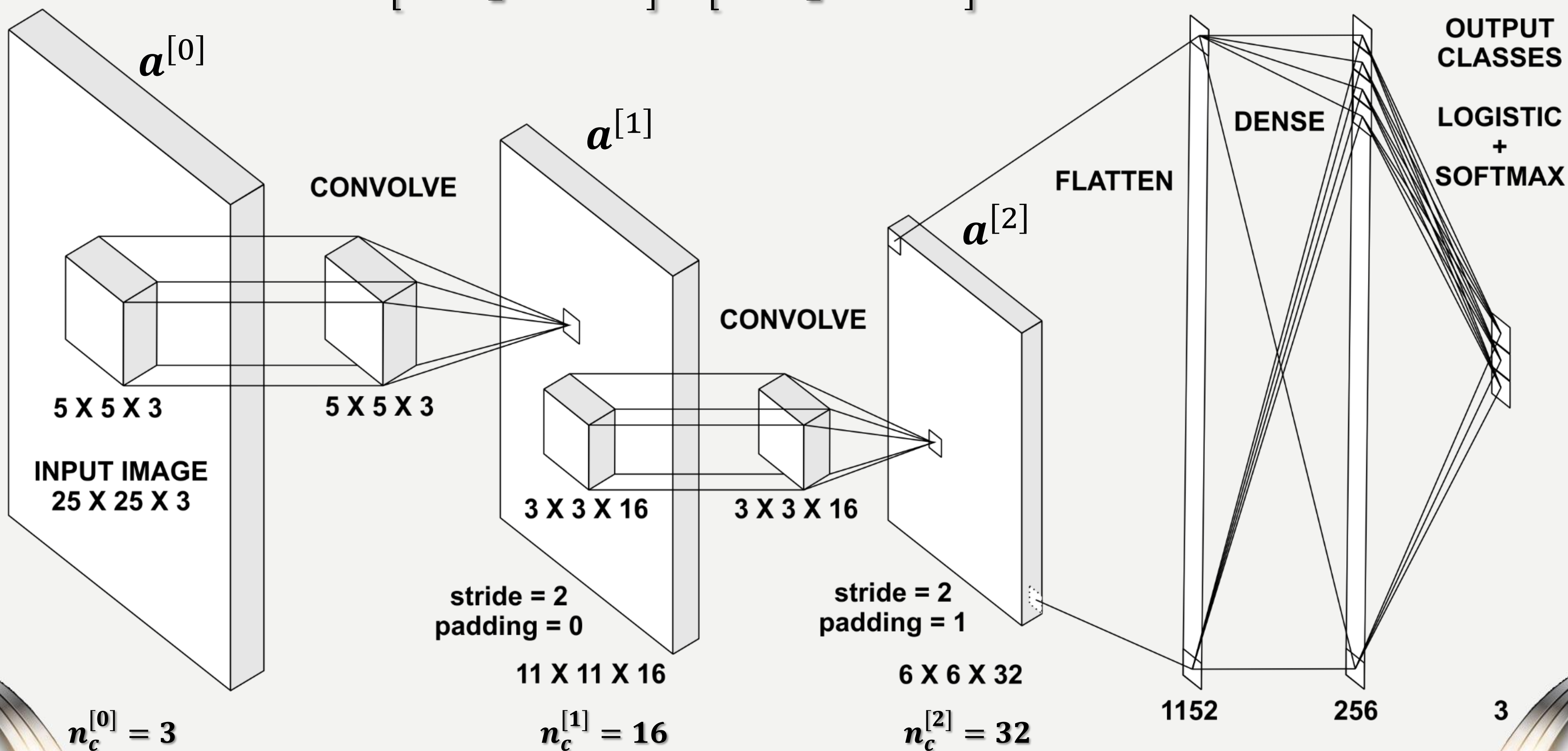
$$A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$$

Simple Convolutional Network

Let's compute the sizes for this exemplar convolutional network:

$$n_H^{[1]} \times n_W^{[1]} \times n_c^{[1]} = \left\lfloor \frac{25 + 2 \cdot 0 - 5}{2} + 1 \right\rfloor \times \left\lfloor \frac{25 + 2 \cdot 0 - 5}{2} + 1 \right\rfloor \times 16 = 11 \times 11 \times 16$$

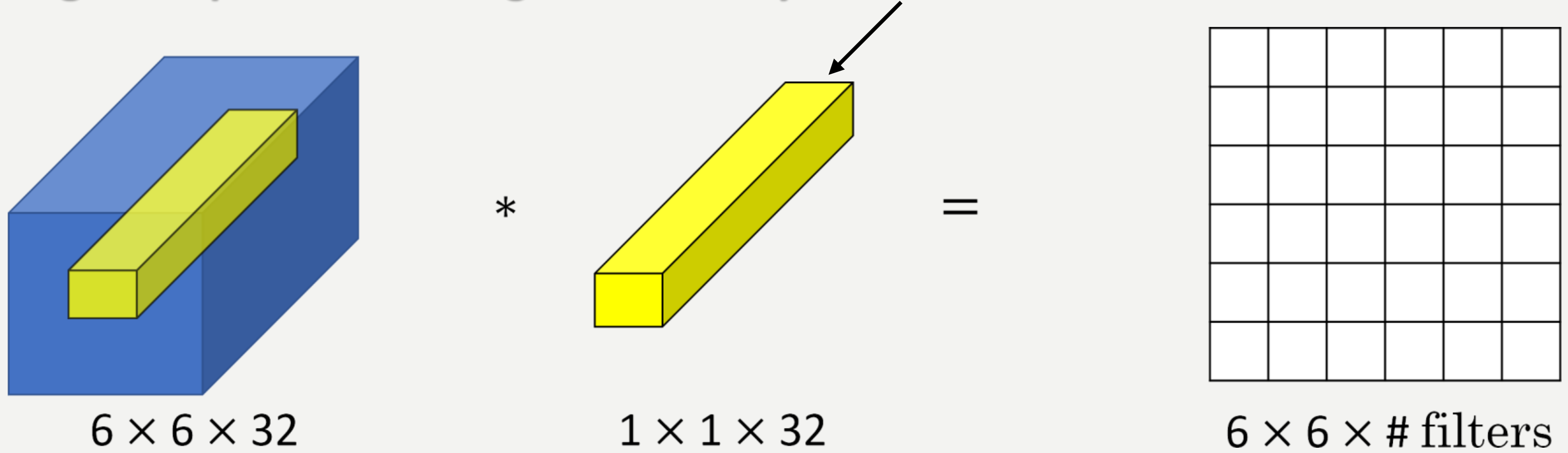
$$n_H^{[2]} \times n_W^{[2]} \times n_c^{[2]} = \left\lfloor \frac{11 + 2 \cdot 1 - 3}{2} + 1 \right\rfloor \times \left\lfloor \frac{11 + 2 \cdot 1 - 3}{2} + 1 \right\rfloor \times 32 = 6 \times 6 \times 32 = 1152 = n_H^{[3]}$$



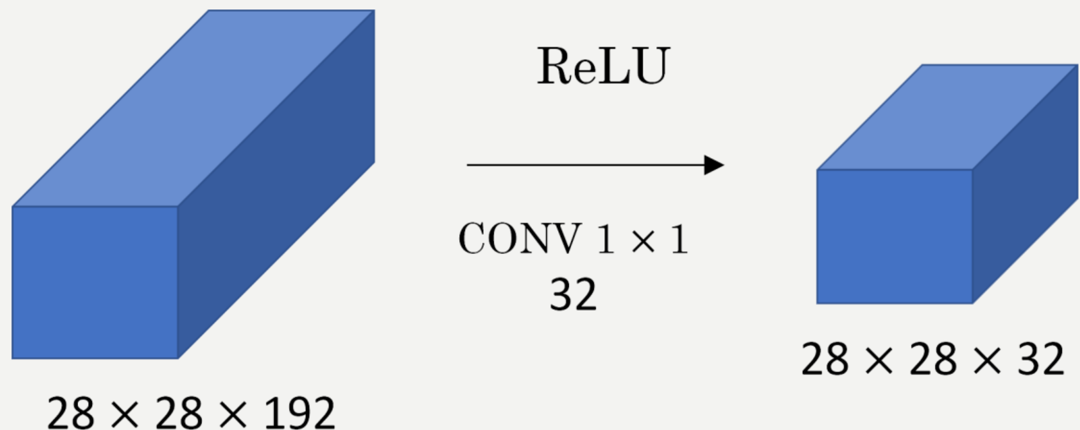
1 x 1 Convolutions

[Paper: Network In Network, Authors: Min Lin, Qiang Chen, Shuicheng Yan. National University of Singapore, arXiv preprint, 2013]:

One-by-one convolutions (called also as network in network) can use various features represented by the various convolutional filters with different strengths expressed through the one-by-one-dimensional convolution filter:



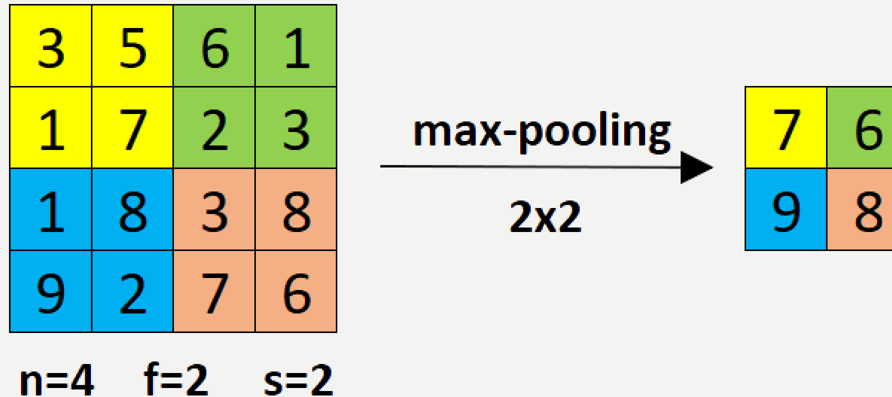
This kind of convolution can be used **to shrink the filter volume (depth)**:



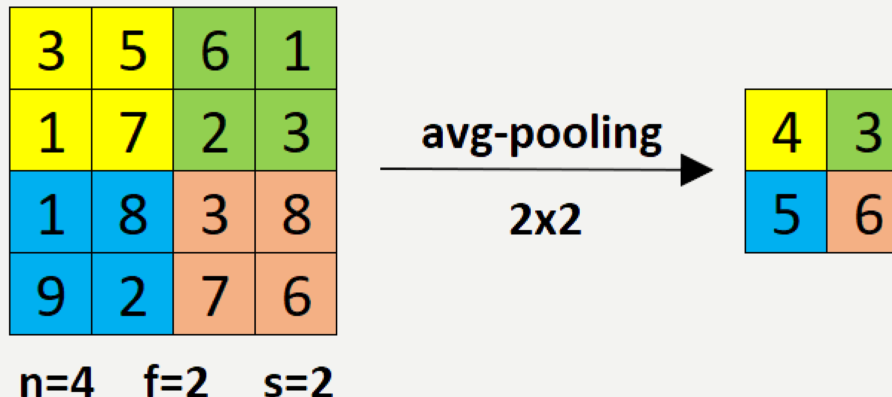
Pooling Layer

To sample the image down (downsampling), we often use pooling layers:

- **Max-pooling** chooses the maximum value from the selected region (stride = 2):



- **Avg-pooling** chooses the average value from the selected region (stride = 2):



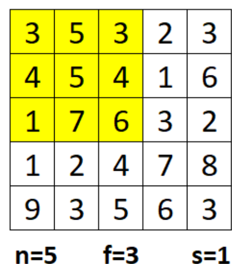
Be careful about using max-pooling because it neglects details.

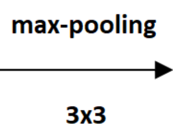
Max-pooling is the most often used in the convolutional networks (CNNs).

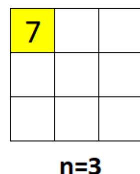
We usually do not use padding (padding = 0) for the pooling operations.

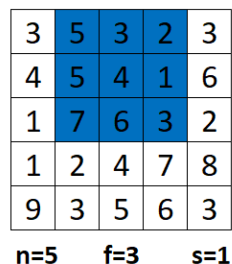
Max-Pooling

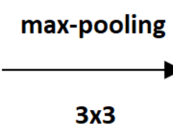
Max-pooling layer for stride = 1, filter size = 3x3:

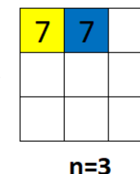


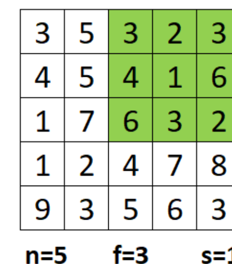


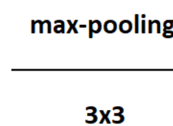


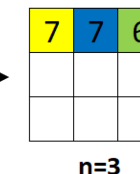


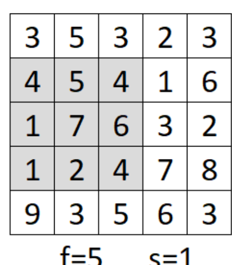


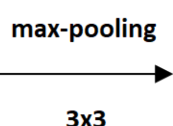


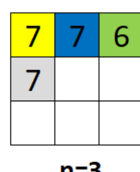


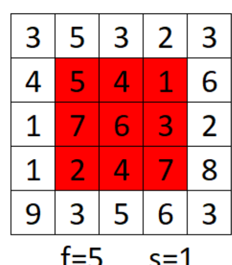


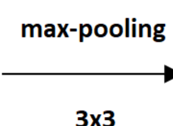


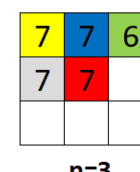


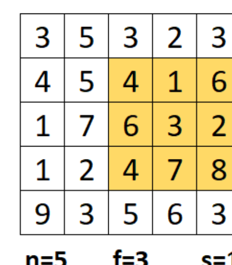


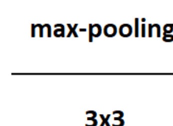


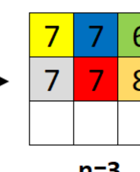


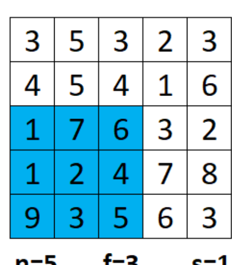


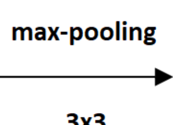


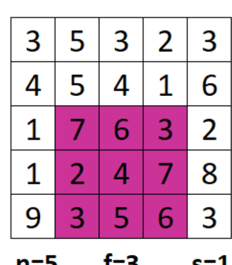


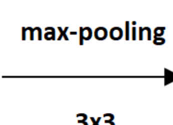


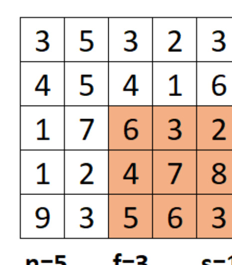


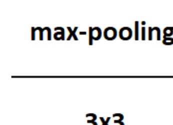


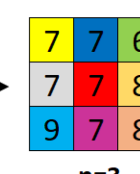












Notice that there are no parameters that can be adapted during the training process!

It is often used to downsample the high-dimensional images, so we use stride > 1.

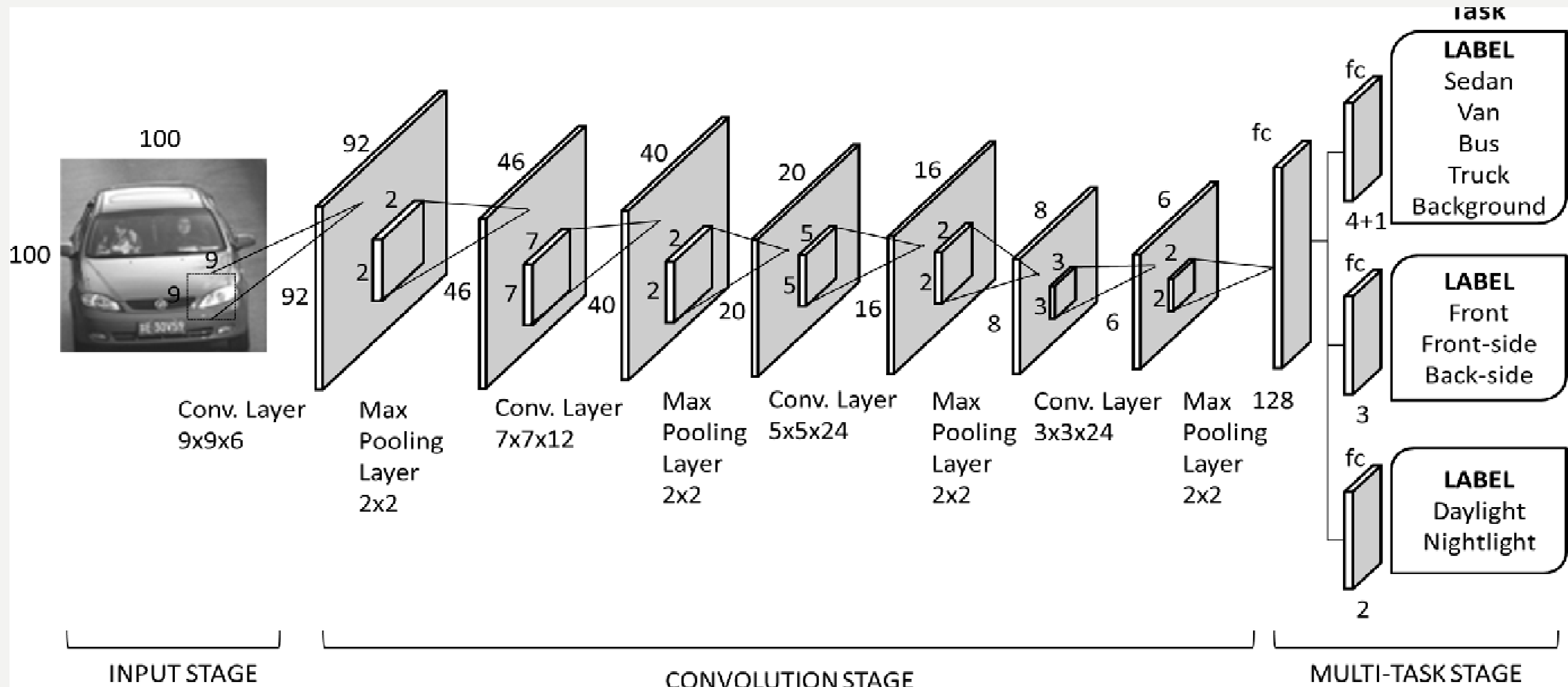
Max-pooling and avg-pooling are computed separately for each channel.

In case of avg-pooling, we calculate averages instead of choosing max values.

Pooling layers

Pooling layers are usually counted together with convolutional layers, however sometimes they are computed separately, so don't get misled!

An example convolutional network with pooling layers:



Popular Convolutional Structures

What structures of CNNs are popular and often used to created non-research models?

CNN Structure

When designing a convolutional model, we can use various numbers and combinations of layers, different numbers of neurons in layers and many more.

We usually present the structure of convolutional networks in the following way:

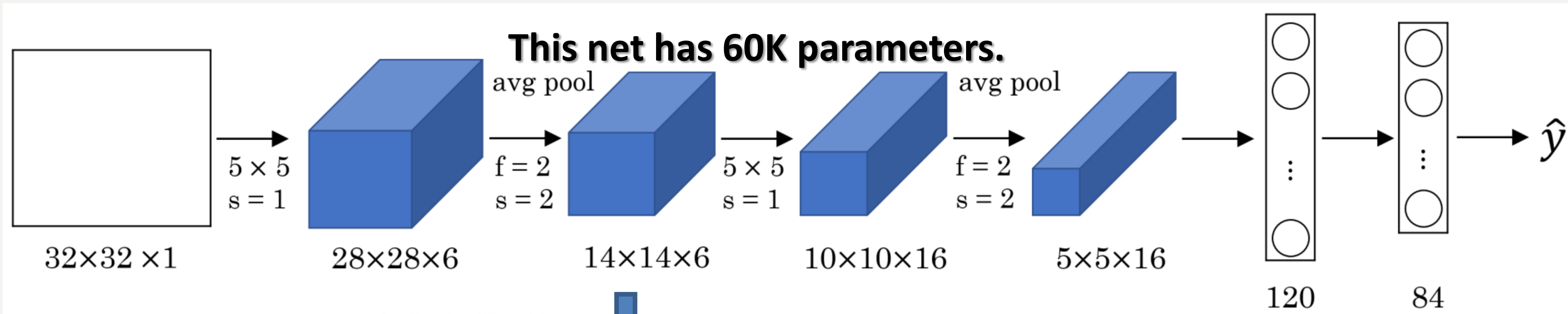
Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

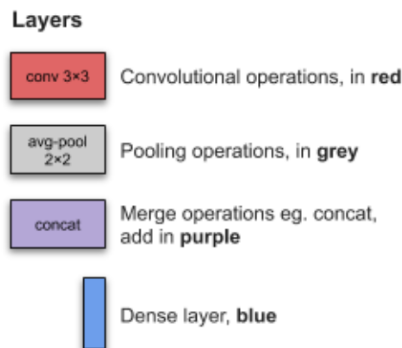
```
model1.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

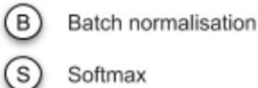
Let’s get inspired by the popular CNN structures developed for various tasks, which we can reuse using transfer learning in the future (because they were used and trained to many problems in the past), and look how we can create our structures to our problems.

LeNet-5 (1998)

[LeCun et al., 1998. Gradient-based learning applied to document recognition]:



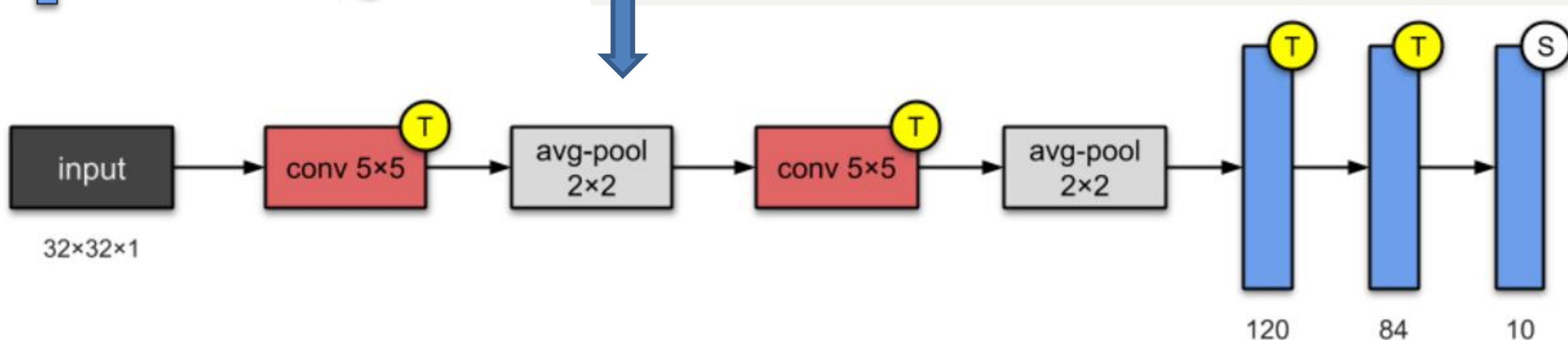




LeNet-5 is one of the simplest architectures.

The average-pooling layer as we know it now was called
a sub-sampling layer and it had trainable weights,
which isn't the current practice of designing CNNs nowadays.

The modern version of LeNet-5 uses SoftMax in the output layer.



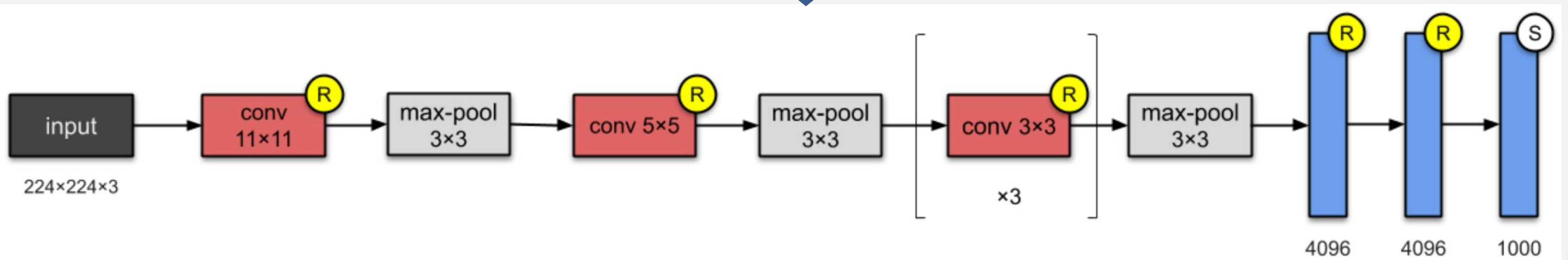
[https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d]

AlexNet (2012)

[Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]:

227×227 ×3
11 × 11
s = 4
55×55 × 96
MAX-POOL
3 × 3
s = 2
27×27 ×96
5 × 5
same
27×27 ×256
MAX-POOL
3 × 3
s = 2
13×13 ×256
3 × 3
same
13×13 ×384
3 × 3
13×13 ×384
3 × 3
13×13 ×256
MAX-POOL
3 × 3
s = 2
6×6 ×256
=
9216
4096
4096
Softmax
1000

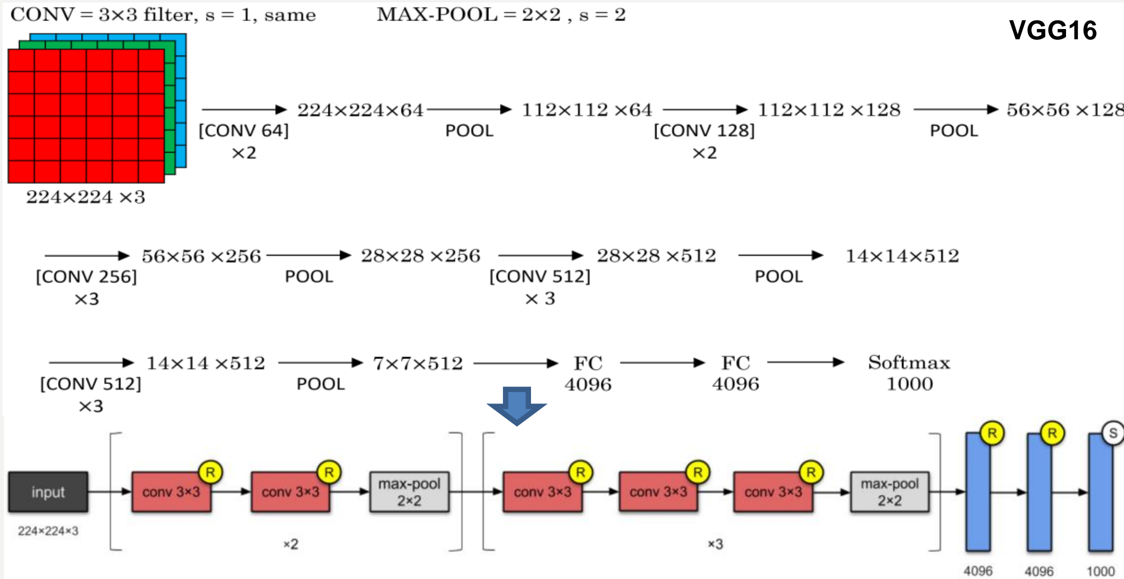
This net has 60M parameters.



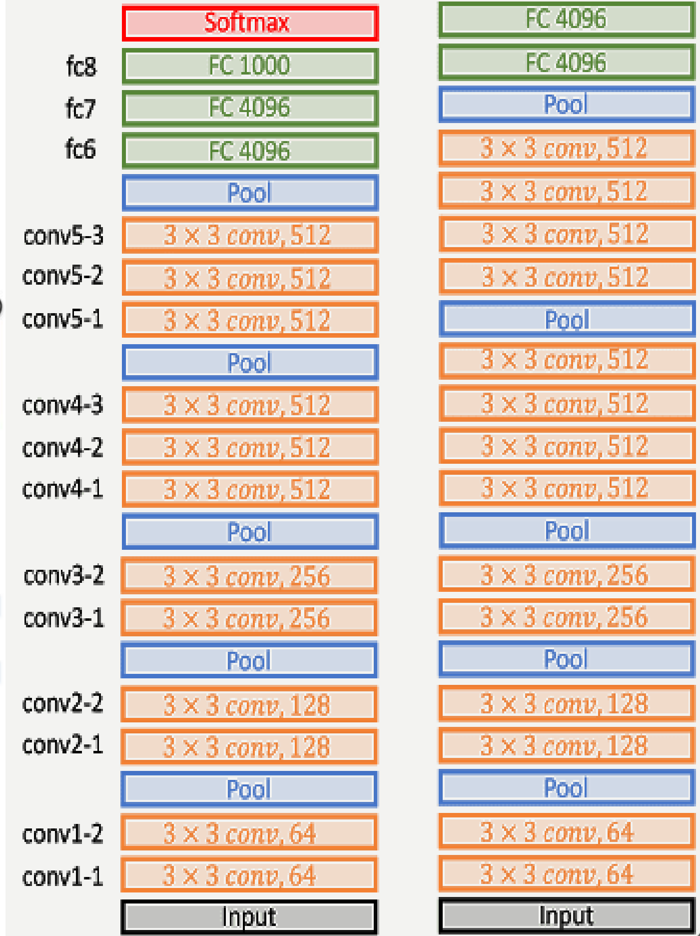
It was the first to implement Rectified Linear Units (ReLUs) as activation functions.

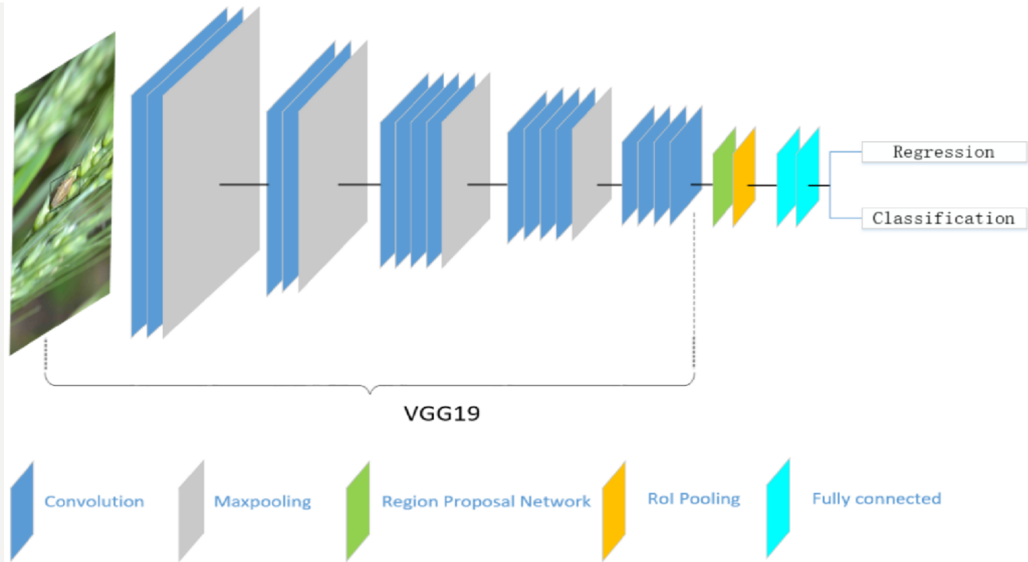
VGG-16 and VGG-19 (2014)

[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]:



This net has 138M parameters.





ResNets

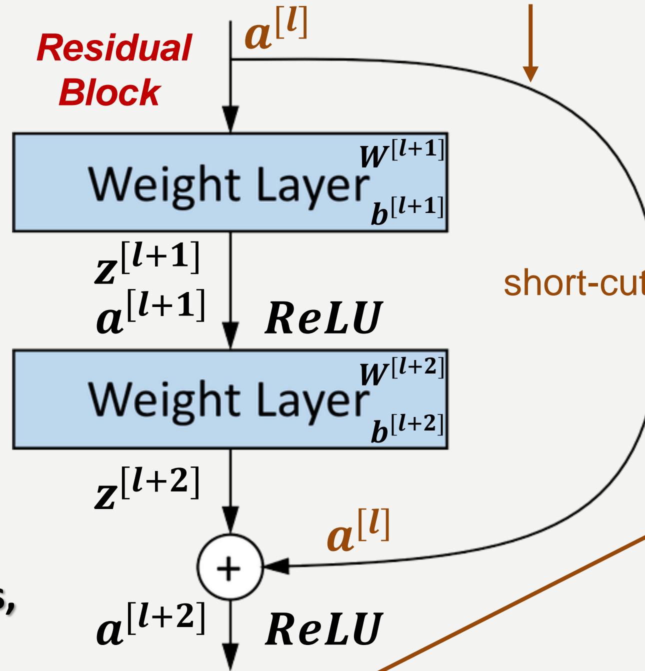
[He at al., 2015, Deep residual networks for image recognition]:

*ResNets are constructed from the stacked **residual blocks** that regularize the non-linear processing using **short-cut (identity, skip connection) connections**:*

$$z^{[l+1]} = W^{[l+1]} \cdot a^{[l]} + b^{[l+1]}$$
$$a^{[l+1]} = ReLU\left(z^{[l+1]}\right)$$

$$z^{[l+2]} = W^{[l+2]} \cdot a^{[l+1]} + b^{[l+2]}$$
$$a^{[l+2]} = ReLU\left(z^{[l+2]} + a^{[l]}\right)$$

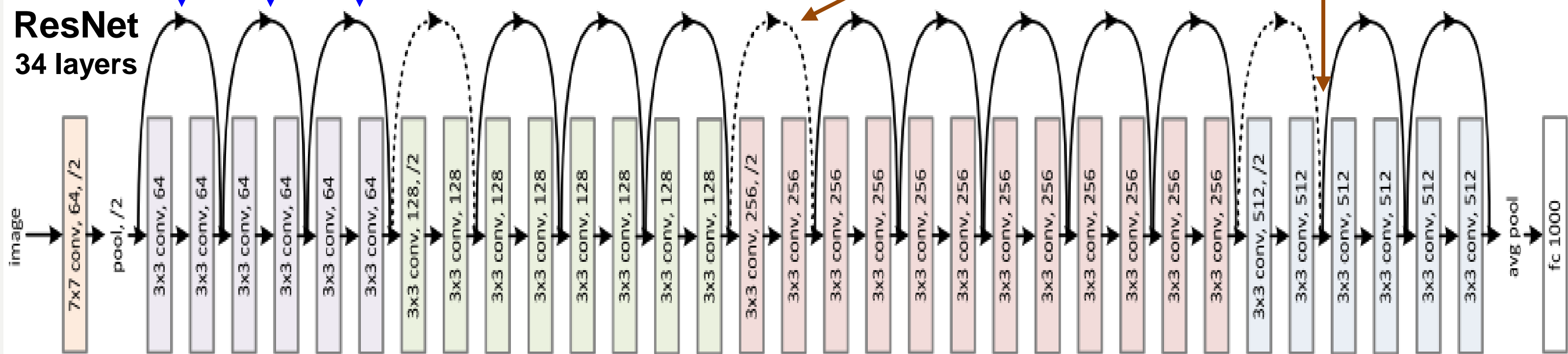
$a^{[l]}$ and $z^{[l+2]}$ must have the same dimensions, so in ResNets, we use the same convolutions:



ResNets allow us to construct much deeper architectures because residual blocks avoid overfitting.

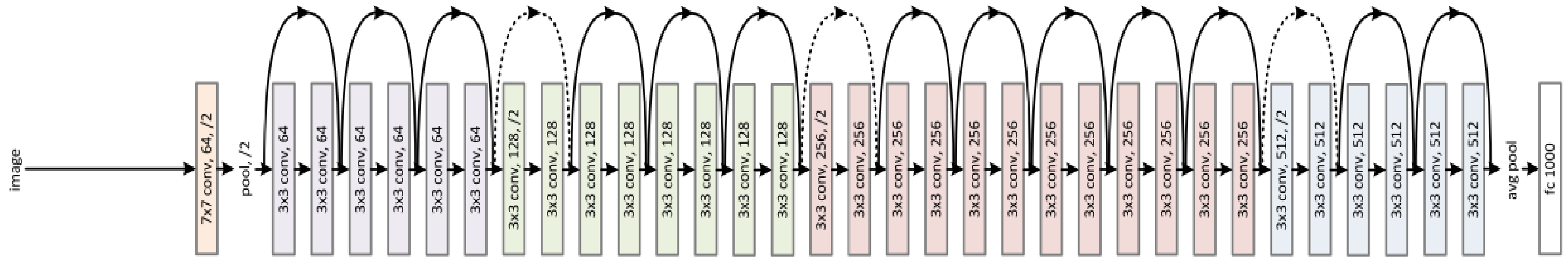
If we want to use different dimensions of $a^{[l]}$ and $z^{[l+2]}$, we must use extra weight matrix W_s to transform:

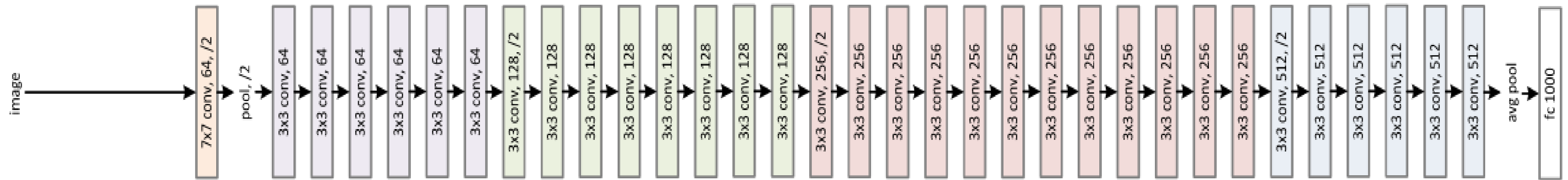
$$a^{[l+2]} = ReLU\left(z^{[l+2]} + W_s^{[l+2]} \cdot a^{[l]}\right)$$

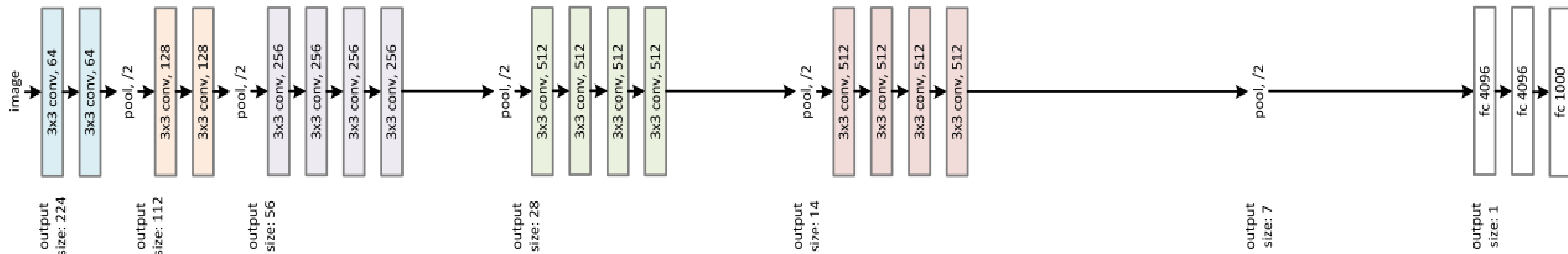


Comparison of ResNet to PlainNet and VGG-19

[He at al., 2015, Deep residual networks for image recognition]:







ResNets are constructed from the stacked residual blocks that regularize the non-linear processing using short-cut (identity, skip connection) connections.

ResNet-50 (2015)

This net has 26M parameters!

It used skip connections the first time, designed much deeper CNNs (up to 152 layers) without compromise with generalization, and was among the first to use batch normalization.

Paper: Deep Residual Learning for Image Recognition, Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Microsoft

Published in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Inception Module

Inception modules allow to use various convolutions (filters) at the same time:

1×1

3×3

5×5

MAX-POOL

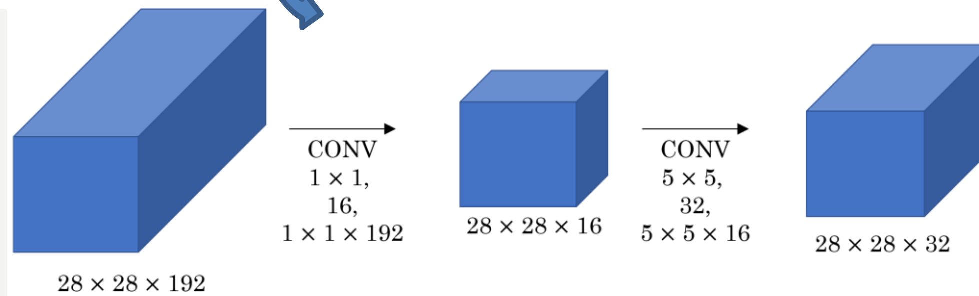
$28 \times 28 \times 192$

CONV 5×5, same, 32

$28 \times 28 \times 192$ → $28 \times 28 \times 32$

28, 28, 32, 32, 128, 64

Computational cost of this convolution is 28x28x32 x 5x5x192 ≈ 120M operations.



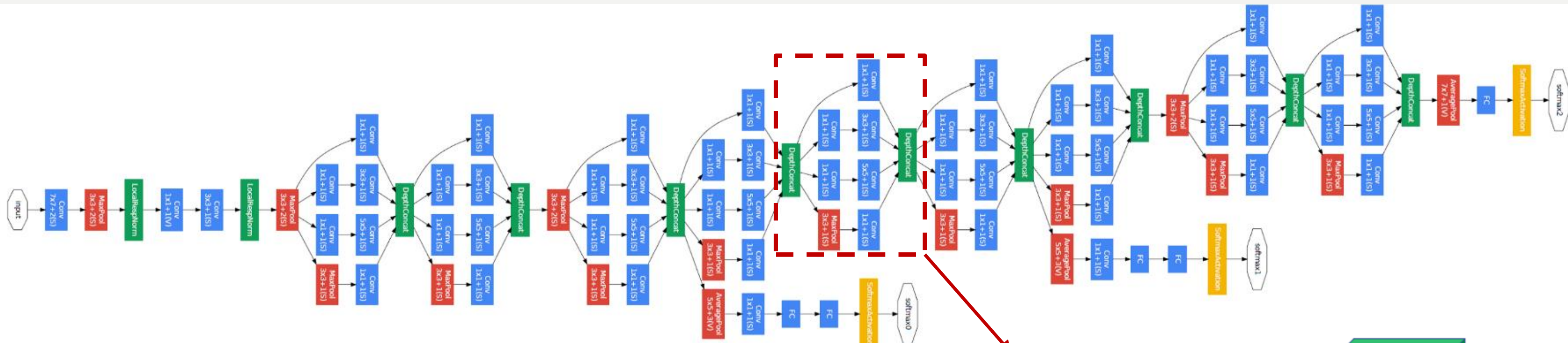
Using 1x1 convolutions, we can reduce the number or multiplications 10 times:

(28x28x16 x 1x1x192) + (28x28x16 x 1x1x192) ≈ 12.4M operations

[Szegedy et al. 2014. Going deeper with convolutions]

Inception Networks (2014)

Building an inception network from inception modules:



Previous Activation

1 × 1 CONV
28x28x64

1 × 1 CONV
1x1x96

3 × 3 CONV
28x28x128

1 × 1 CONV
1x1x96

5 × 5 CONV
28x28x32

MAXPOOL
3 × 3,s = 1
same
28x28x192

1 × 1 CONV
28x28x32

Channel Concat

28
28
32
32
128
64

Inception-v1 (2014)

Paper: Going Deeper with Convolutions,
Authors: Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich.
Google, University of Michigan, University of North Carolina
Published in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

This net has 5M parameters.

It has parallel towers of convolutions with different filters, uses 1x1 convolutions, adding nonlinearity, and two auxiliary classifiers to provide additional regularization.

input
224×224×3
Stem
inception
×2
max-pool 3×3
inception
inception
×3
inception
max-pool 3×3
inception
×2
avg-pool 7×7
4096
1000
avg-pool 5×5
conv 1×1
1024
1000
avg-pool 5×5
conv 1×1
1024
1000

Modules/Blocks

Modules (groups of convolutional, pooling and merge operations), in **yellow, green,** or **orange**.
The operations that make up these modules will also be shown.

Module A = conv 3×3 → avg-pool 2×2
Module A

Module B = conv 11×11 → global avg-pool
Module B

Module C = conv 1×1 → max-pool 5×5
Module C

Repeated layers or modules/blocks

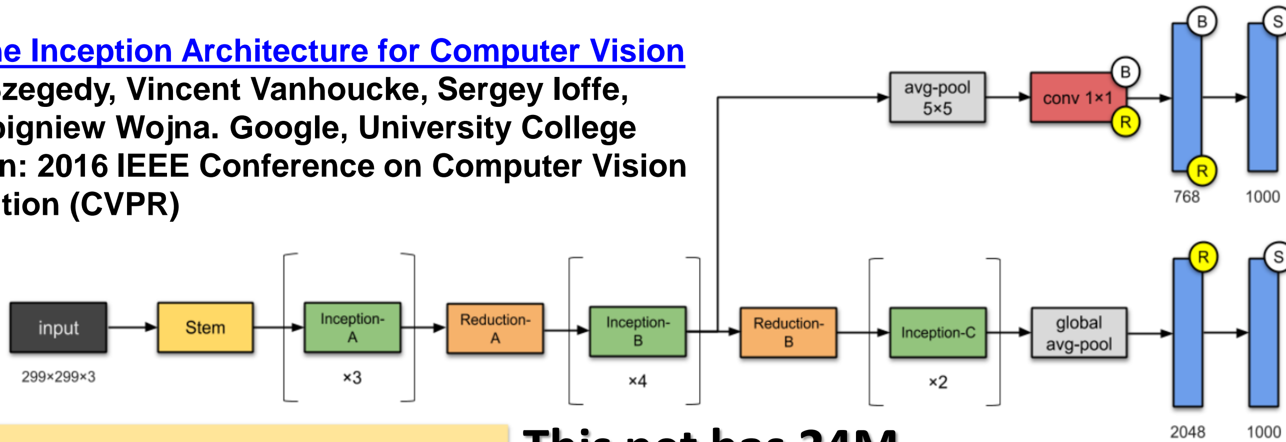
Module B ×2 = Module B → Module B

conv 7×7 → max-pool 3×3 → conv 1×1 → conv 3×3 → max-pool 3×3
Stem

conv 1×1 → conv 5×5
conv 1×1 → conv 3×3
max-pool 3×3 → conv 1×1
conv 1×1
concat
Inception module

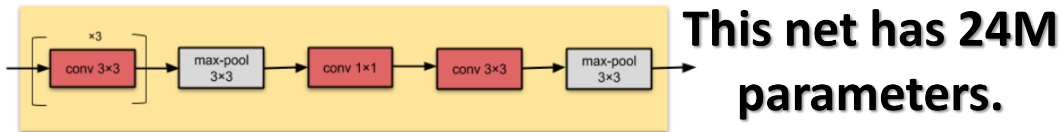
Inception-v3 (2015)

Paper: Rethinking the Inception Architecture for Computer Vision
Authors: Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. Google, University College London, Published in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

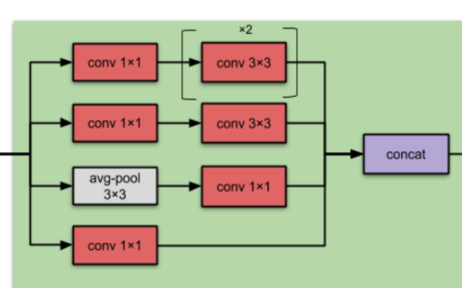


It factorizes $n \times n$ convolutions into asymmetric convolutions: $1 \times n$ and $n \times 1$ convolutions, 5×5 convolution to two 3×3 convolutions, and replaces 7×7 to a series of 3×3 convolutions

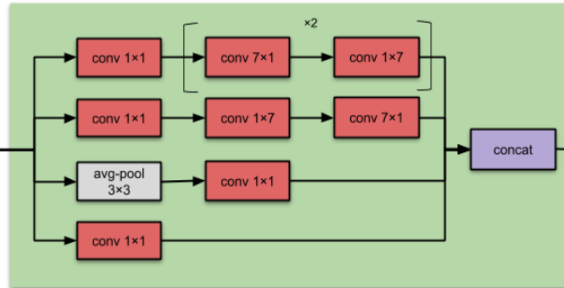
This net has 24M parameters.



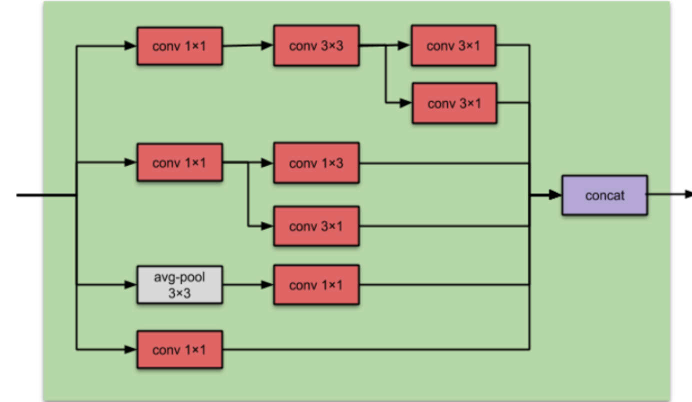
Stem



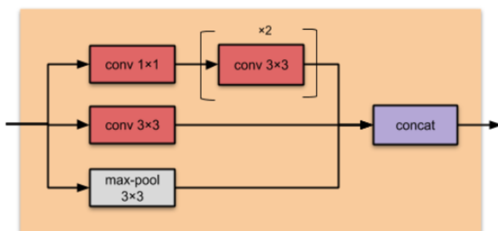
Inception-A



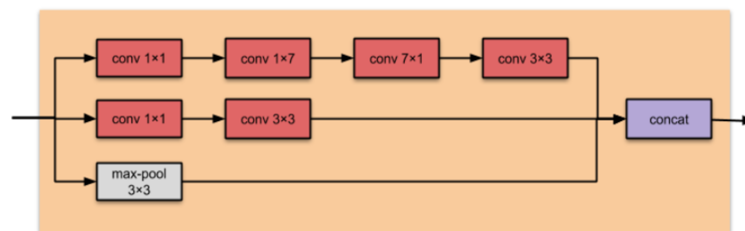
Inception-B



Inception-C



Reduction-A

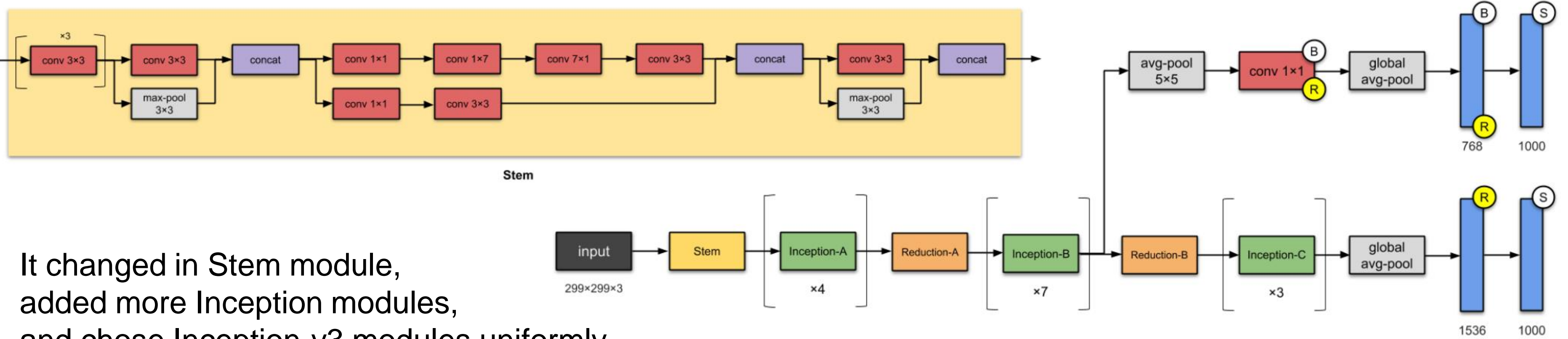


Reduction-B

Inception-v4 (2016)

Paper: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning

Authors: Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi. Google.



It changed in Stem module,
added more Inception modules,
and chose Inception-v3 modules uniformly,
i.e. used the same number of filters for every module.

Inception-A

Inception-B

Inception-C

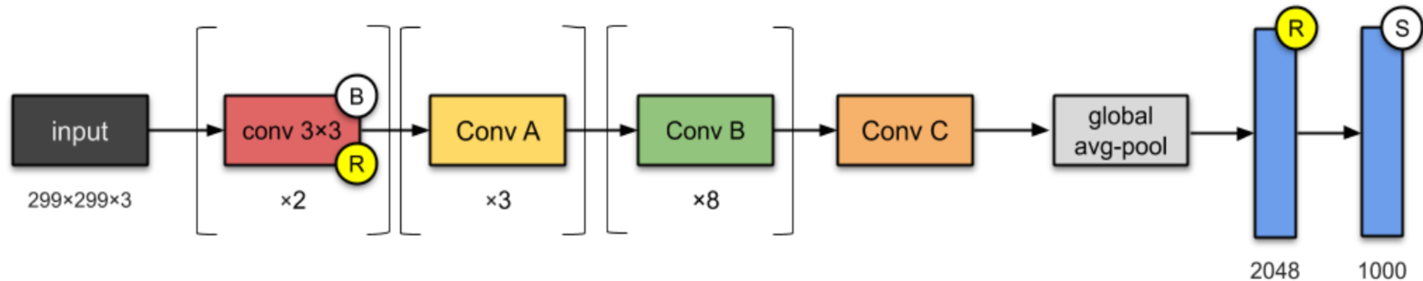
Reduction-A

Reduction-B

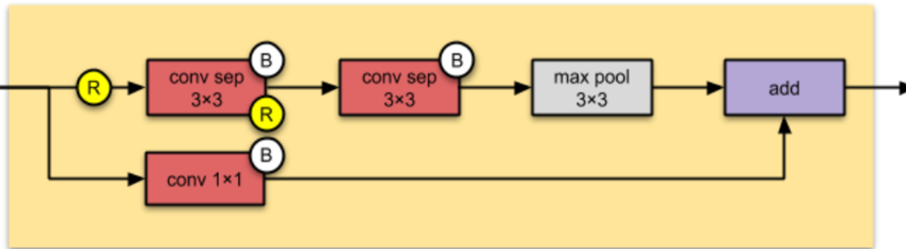
This net has 43M parameters.

Xception (2016)

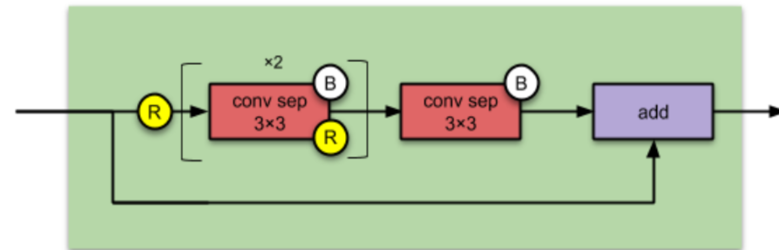
Xception is an adaptation from Inception, where the Inception modules have been replaced with depth-wise separable convolutions.



This net has 23M parameters.

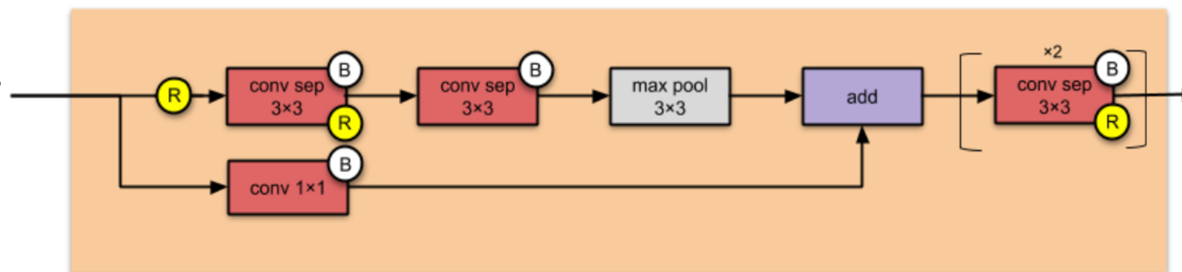


Conv A



Conv B

Paper: [Xception: Deep Learning with Depthwise Separable Convolutions]
Authors: François Chollet. Google.
Published in: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)



Conv C

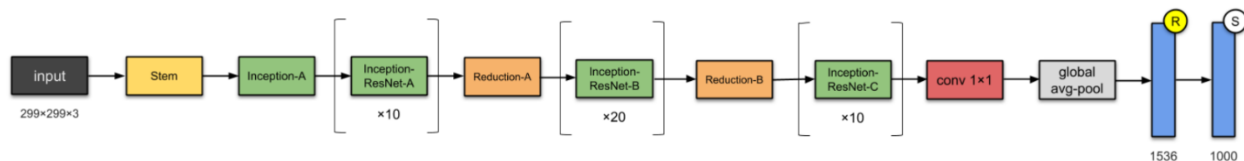
Cross-channel correlations were captured by 1×1 convolutions, and spatial correlations within each channel were captured via the regular 3×3 or 5×5 convolutions.

Inception ResNet-v2 (2016)

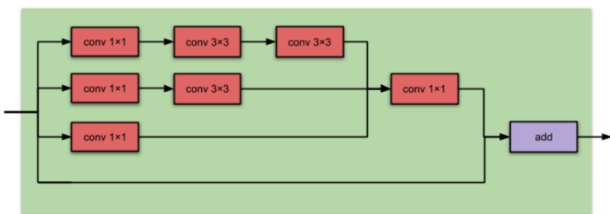
Paper: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning,

Authors: Christian Szegedy, Sergey Loffe, Vincent Vanhoucke, Alex Alemi. Google.

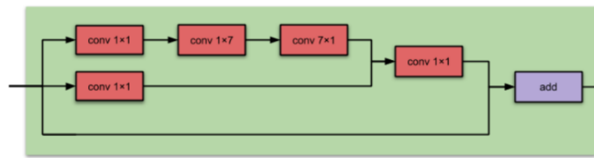
Published in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence



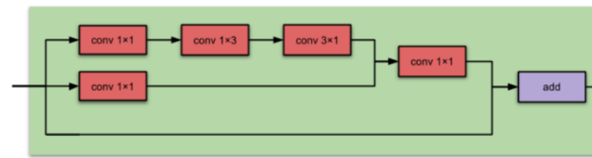
This net has 56M parameters.



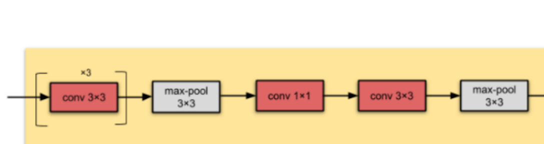
Inception-ResNet-A



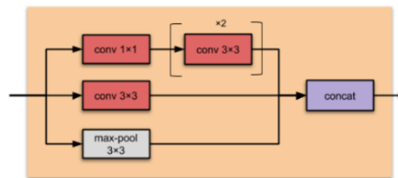
Inception-ResNet-B



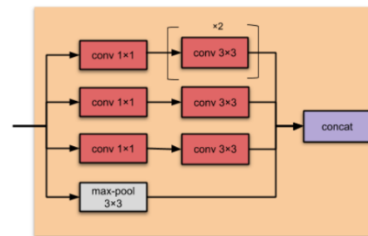
Inception-ResNet-C



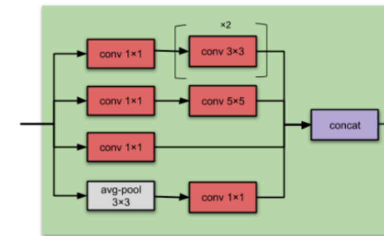
Stem



Reduction-A



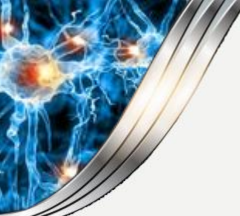
Reduction-B



Inception-A

This solution:

- converts Inception modules to Residual Inception blocks.
- adds more Inception modules.
- adds a new type of Inception module (Inception-A) after the Stem module.

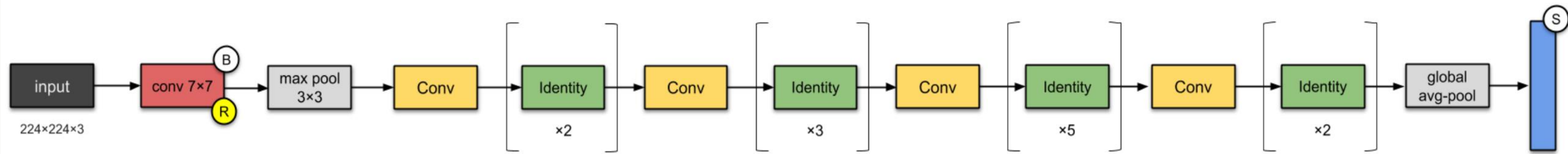


ResNeXt-50 (2017)

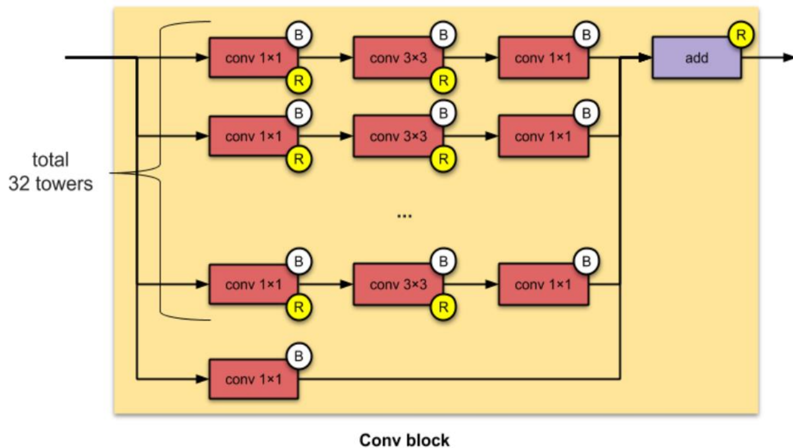
Paper: Aggregated Residual Transformations for Deep Neural Networks

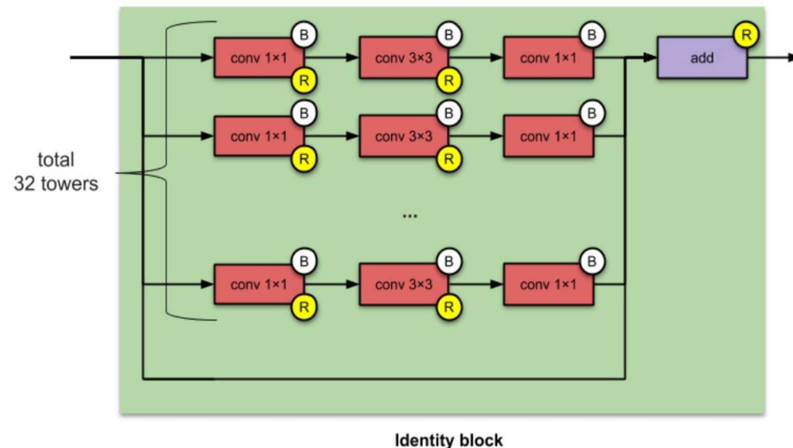
Authors: Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, Kaiming He. University of California San Diego, Facebook Research

Published in: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)



This net has 25M parameters.





It scales up the number of parallel towers ("cardinality") within a module.

[https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d]

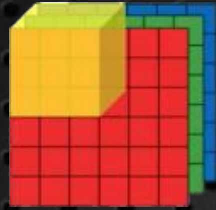
GitHub sources

It is not necessary to implement all these networks from scratch,
but you can use the original sources available on GitHub repositories:

1. Find the source at GitHub.
2. Copy the source at GitHub repository.
3. Clone it in your computer:
   > git clone https://github.com/
4. Go to the repository, e.g.: cd deep-residual-networks
5. Go to the prototxt/more and look at the structure of the chosen network.

When implementing selected types of networks, we often use open-source implementations available on GitHub and adapt them to our tasks.

In the same way, we copy implementations with trained parameters when we want to use transfer learning, i.e. reusing the already trained models to different tasks which use similar sets of features that can be reused.



Handwritten Digits Classification

An example of the CNN model implementation using the MNIST dataset.

MNIST Classification in Jupyter Notebook

Now, let's try to create and train a simple Convolutional Neural Network (CNN) to tackle with a handwritten digit classification problem using MNIST dataset:

Each image in the MNIST dataset is 28x28 pixels and contains a centred, grayscale digit form 0 to 9. Our goal is to classify these images to one of the ten classes using ten output neurons of the CNN network.

MNIST Classification in Jupyter Notebook

Let's import libraries, frameworks, and setting of the parameters:

In [1]:

```python
'''Trains a simple ConvNet on the MNIST dataset. It gets over 99.60% test accuracy after 48 epochs
(but there is still a margin for hyperparameter tuning). Training can take an hour or so!'''

# Import libraries
from __future__ import print_function
import numpy as np
import math
from math import ceil
import tensorflow as tf
import os
import seaborn as sns
import matplotlib.pyplot as plt  # library for plotting math functions
import pandas as pd
import keras   # Import keras framework with various functions, models and structures
from keras.datasets import mnist # gets MNIST dataset from repository
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report

# Set parameters for plots
%matplotlib inline
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

print ("TensorFlow version: " + tf.__version__)
```

```
TensorFlow version: 2.1.0
```

MNIST Classification in Jupyter Notebook

Set hyperparameters and the method for presenting test results:

In [2]:
```
LABELS= ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# Define the confusion matrix for the results
def show_confusion_matrix(validations, predictions, num_classes):
    matrix = metrics.confusion_matrix(validations, predictions)
    plt.figure(figsize=(num_classes, num_classes))
    hm = sns.heatmap(matrix,
                cmap='coolwarm',
                linecolor='white',
                linewidths=1,
                xticklabels=LABELS,
                yticklabels=LABELS,
                annot=True,
                fmt='d')
    plt.yticks(rotation = 0)  # Don't rotate (vertically) the y-axis labels
    hm.invert_yaxis() # Invert the labels of the y-axis
    hm.set_ylim(0, len(matrix))
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()
```

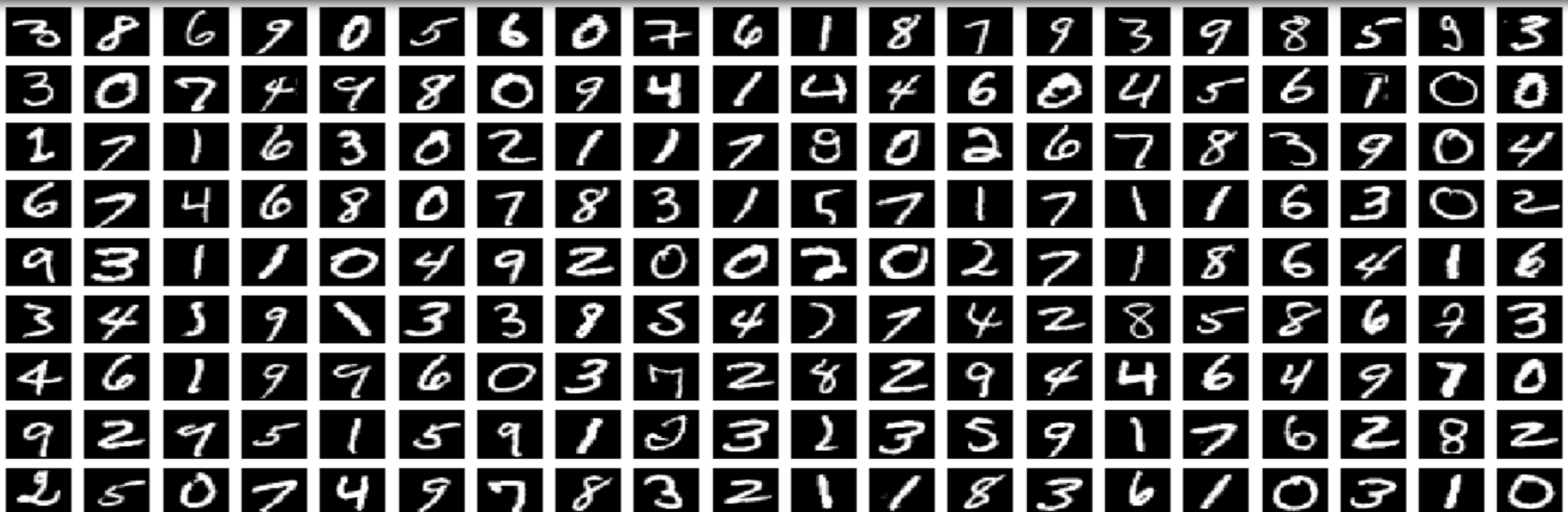
In [3]:
```
# Define hyperparameters
batch_size = 512   # size of mini-baches
num_classes = 10   # number of classes/digits: 0, 1, 2, ..., 9
epochs = 3         # how many times all traing examples will be used to train the model

# Input image dimensions
img_rows, img_cols = 28, 28

# Split the data between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data() # 60000 training and 10000 testing example
```

MNIST Classification in Jupyter Notebook

Look at sample MNIST training examples (handwritten digits):

In [4]:

```
# Show a few sample digits from the training set
plt.rcParams['figure.figsize'] = (2.5, 2.5) # set default size of plots
col1 = 10
row1 = 1
fig = plt.figure(figsize=(col1, row1))
for index in range(0, col1*row1):
    fig.add_subplot(row1, col1, index + 1)
    plt.axis('off')
    plt.imshow(x_train[index]) # index of the sample picture
    plt.title("Class " + str(y_train[index]))
plt.show()

# Show a few sample digits from the training set
plt.rcParams['figure.figsize'] = (1.0, 1.0) # set default size of plots
col2 = 20
row2 = 10
fig = plt.figure(figsize=(col2, row2))
for index in range(col1*row1, col1*row1 + col2*row2):
    fig.add_subplot(row2, col2, index - col1*row1 + 1)
    plt.axis('off')
    plt.imshow(x_train[index]) # index of the sample picture
plt.show()
```



jupyter

MNIST Classification in Jupyter Notebook

Load training data, changing the shapes of the matrices storing training and testing data, transform the input data from [0, 255] to [0.0, 1.0] range, and convert numerical class names into categories:

In [5]:

```
# According to the different formats reshape training and testing data
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# Transform training and testing data and show their shapes
x_train = x_train.astype('float32')   # Copy this array and cast it to a specified type
x_test = x_test.astype('float32')     # Copy this array and cast it to a specified type
x_train /= 255   # Transfrom the training data from the range of 0 and 255 to the range of 0 and 1
x_test /= 255    # Transfrom the testing data from the range of 0 and 255 to the range of 0 and 1
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors (integers) to binary class matrices using as specific
y_train = keras.utils.to_categorical(y_train, num_classes) # y_train - a converted class vector int
y_test = keras.utils.to_categorical(y_test, num_classes) # y_test - a converted class vector into c
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

MNIST Classification in Jupyter Notebook

Build a neural network structure (a computational model):

In [6]:

```
# Define the sequential Keras model composed of a few layers
model = Sequential()   # establishes the type of the network model
# Conv2D - creates a convolutional layer (https://keras.io/layers/convolutional/#conv2d) with
# filters - specified number of convolutional filters
# kernel_size - defines the frame (sliding window) size where the convolutional filter is implement
# activation - sets the activation function for this layers, here ReLU
# input_shape - defines the shape of the input matrix (vector), here input_shape = (1, img_rows, in
model.add(Conv2D(filters=32, kernel_size=(3, 3),activation='relu', input_shape=input_shape))
# model.add(Conv2D(32, (3, 3), activation='relu')) - shoter way of the above code
# MaxPooling2D pools the max value from the frame (sliding window) of 2 x 2 size
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.20)) # Implements the drop out with the probability of 0.20
model.add(Conv2D(64, (3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128,(3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.30))
model.add(Conv2D(256,(3, 3), activation='relu',padding='same'))
#model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.40))
model.add(Conv2D(512,(3, 3), activation='relu',padding='same'))
#model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.50))
# Finish the convolutional model and flatten the layer which does not affect the batch size.
model.add(Flatten())
# Use a dense layer (MLP) consisting of 256 neurons with relu activation functions
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(num_classes, activation='softmax'))
```

MNIST Classification in Jupyter Notebook

Compile the model using optimizer, augment data using generator, and train it:

In [8]:

```
# Compile the model using optimizer
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),  # choose the optimizer
              metrics=['acc']) # List of metrics to be evaluated by the model during training and t

# Learning rate reduction durint the training process: https://keras.io/callbacks/#reducelronplatea
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc', # quantity to be monitored (val_loss
                                            factor=0.5, # factor by which the learning rate will be
                                            patience=5, # number of epochs that produced the monito
                                            verbose=1,  # 0: quiet, 1: update messages.
                                            min_lr=0.001) # lower bound on the learning rate

# Augmentation of training data. It generates batches of tensor image data with real-time data augm
datagen = ImageDataGenerator(
        rotation_range=5, # rotate images in degrees up to the given degrees
        zoom_range=0.2, # zoom images
        width_shift_range=0.15,  # shift images horizontally
        height_shift_range=0.15)  # shift images vertically
# Computes the internal data stats related to the data-dependent transformations, based on an array
datagen.fit(x_train) # Fits the data generator to the sample data x_train.

# Simple train the model, validate, evaluate, and present scores
'''history = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,  # no of training epochs
          verbose=1,  # 0 = silent, 1 = progress bar, 2 = one line per epoch
          validation_data=(x_test, y_test),
          validation_split=0.2,  # cross-validation split 1/5
          shuffle=True) # method of how to shuffle training and validation data '''

# Advanced train the model, validate, evaluate, and present scores: https://keras.io/models/model/#
history = model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                              epochs=epochs,  # no of training epochs
                              steps_per_epoch=x_train.shape[0]//batch_size, # no of mini-batches
                              validation_data=(x_test, y_test),
                              verbose=1,  # 0 = silent, 1 = progress bar, 2 = one line per epoch
                              callbacks=[learning_rate_reduction])
```

Evaluate the trained model and plot how it convergences on charts:

```
Epoch 1/3
117/117 [==============================] - 239s 2s/step - loss: 1.9395 - acc: 0.2978 - val_loss: 1.0056 - val_acc: 0.6138
Epoch 2/3
117/117 [==============================] - 254s 2s/step - loss: 0.8777 - acc: 0.7117 - val_loss: 0.1801 - val_acc: 0.9456
Epoch 3/3
117/117 [==============================] - 252s 2s/step - loss: 0.3709 - acc: 0.8885 - val_loss: 0.0808 - val_acc: 0.9753
```

Evaluate, score and plot the accuracy and the loss

In [8]:

```
# Evaluate the model and print out the final scores for the test set
score = model.evaluate(x_test, y_test, verbose=0)  # evaluate the model on the test set
print('Test loss:', score[0])       # print out the loss = score[0] (generalization error)
print('Test accuracy:', score[1])  # print out the generalization accuracy = score[1] of the model on test set

# Plot training & validation accuracy values: https://keras.io/visualization/#training-history-visualization
plt.rcParams['figure.figsize'] = (15.0, 5.0) # set default size of plots
plt.plot(history.history['acc']) # The history object gets returned by the fit method of models.
plt.plot(history.history['val_acc'])  # val_accuracy
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')  # OR plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values: https://keras.io/visualization/#training-history-visualizatio
plt.plot(history.history['loss']) # The history object gets returned by the fit method of models.
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc ='upper left')  # OR plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

```
Test loss: 0.080788445275O9063
Test accuracy: 0.9753000140190125
```

MNIST Classification in Jupyter Notebook

Model evaluation, convergence drawing and error charts:

```
Test loss: 0.08078844527509063
Test accuracy: 0.9753000140190125
```

Model accuracy

Here is the presentation of only 3 learning epochs!
We usually train such networks for several dozen epochs, getting better results (accuracy) and smaller errors!

Model loss

Why results on test data are better than on train data?

Mini-batch mode and regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at the testing time, so the model does not change as during training time. That is why the train error is always bigger, which can appear weird in view of classic machine learning models.

MNIST Classification in Jupyter Notebook

Generate summaries of the training and show a confusion matrix:

In [11]:

```
# Use the trained model for predictions of the test data
y_pred_test = model.predict(x_test)

# Take the class with the highest probability from the test predictions as a winning one
max_y_pred_test = np.argmax(y_pred_test, axis=1)
max_y_test = np.argmax(y_test, axis=1)

# Show the confution matrix of the collected results
show_confusion_matrix(max_y_test, max_y_pred_test, num_classes)

# Print classification report
print(classification_report(max_y_test, max_y_pred_test))
```

	precision	recall	f1-score	support
0	0.98	0.99	0.99	980
1	1.00	0.98	0.99	1135
2	0.97	0.98	0.98	1032
3	0.95	0.99	0.97	1010
4	0.99	0.96	0.98	982
5	0.96	0.99	0.97	892
6	0.97	0.98	0.98	958
7	0.99	0.96	0.98	1028
8	0.97	0.97	0.97	974
9	0.95	0.96	0.96	1009
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

MNIST Classification in Jupyter Notebook

Confusion (error) matrix in the form of a heat map for the text data:

Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	3	1	0	5	1	18	1	5	8	967
8	2	0	3	6	3	7	4	1	940	8
7	1	1	17	7	1	0	0	990	1	10
6	6	2	0	1	0	4	941	0	4	0
5	0	0	0	9	0	879	1	0	2	1
4	2	0	1	0	941	0	9	0	2	27
3	0	0	0	1002	0	5	0	0	3	0
2	1	0	1011	16	0	0	1	0	3	0
1	0	1114	5	4	0	1	3	1	7	0
0	968	0	1	0	0	0	8	0	3	0

Colour scale: 0, 200, 400, 600, 800, 1000

MNIST Classification in Jupyter Notebook

Count and filter out incorrectly classified test examples to show them:

In [10]:

```python
# Find out misclassified examples
classcheck = max_y_test - max_y_pred_test  # 0 - when the class is the same, 1 - otherwise
misclassified = np.where(classcheck != 0)[0]
num_misclassified = len(misclassified)

# Print misclassification report
print('Number of misclassified examples: ', str(num_misclassified))
print('Misclassified examples:')
print(misclassified)

# Show misclassified examples:
print('Misclassified images (original class : predicted class):')
plt.rcParams['figure.figsize'] = (2.5, 2.5) # set default size of plots
col = 10
row = 2 * math.ceil(num_misclassified / col)
fig = plt.figure(figsize=(col, row))
for index in range(0,num_misclassified):
    fig.add_subplot(row, col, index + 1 + col*(index//col))
    plt.axis('off')
    plt.imshow(x_test[misclassified[index]].reshape(img_rows, img_cols)) # index of the test sample picture
    plt.title(str(max_y_test[misclassified[index]]) + ":" + str(max_y_pred_test[misclassified[index]]))
plt.show()
```

```
Number of misclassified examples:  247
Misclassified examples:
[  18   62   78  151  160  184  206  241  247  259  264  320  324  376
  412  420  435  479  497  511  542  571  582  619  629  646  674  684
  691  717  726  740  774  810  829  881  916  926  938  947  956 1014
 1039 1050 1107 1112 1114 1119 1156 1182 1226 1228 1232 1247 1273 1279
 1289 1299 1364 1393 1403 1453 1459 1527 1553 1621 1654 1709 1721 1754
 1782 1790 1813 1878 1941 1965 2016 2035 2043 2070 2118 2129 2130 2135
 2148 2182 2189 2237 2266 2293 2387 2447 2454 2462 2535 2597 2607 2654
 2659 2705 2780 2823 2896 2939 2959 2995 3069 3073 3132 3166 3240 3269
 3288 3289 3330 3333 3441 3504 3533 3534 3567 3597 3604 3716 3726 3762
 3767 3780 3808 3811 3906 3926 4001 4007 4013 4015 4063 4065 4078 4137
 4145 4207 4212 4224 4265 4271 4360 4477 4482 4497 4500 4571 4575 4604
 4639 4690 4751 4761 4783 4808 4814 4823 4838 4860 4874 4879 4880 4943
 4956 5159 5176 5183 5209 5642 5654 5749 5835 5842 5858 5887 5888 5903
 5906 5914 5937 6011 6023 6065 6071 6081 6091 6166 6505 6554 6555 6558
 6571 6572 6576 6584 6617 6625 6651 6783 6796 6883 6895 7121 7259 7434
 7473 7812 7899 7915 8081 8094 8115 8236 8243 8245 8316 8382 8408 8469
 8509 8520 8527 9009 9015 9019 9024 9036 9071 9280 9505 9530 9539 9629
 9642 9679 9729 9770 9850 9856 9892 9904 9922]
```

MNIST Classification in Jupyter Notebook

247 out of 10,000 incorrectly classified test patterns:

One might wonder why the network had difficulty in classifying them?

Of course, such a network can be taught further to achieve a smaller error!

This network has been taught only for 3 epochs!

```
Misclassified images (original class : predicted class):
```

3:5 9:5 9:8 9:5 4:6 8:3 9:8 9:5 4:6 6:0

9:8 9:8 0:6 4:9 5:3 4:9 8:9 9:5 4:8 4:9

8:5 4:9 8:2 1:8 2:3 2:6 5:3 7:3 8:9 0:6

7:4 4:9 4:9 7:2 4:8 4:9 4:2 2:8 3:5 8:9

1:3 6:5 7:3 2:3 9:5 4:6 3:8 7:2 7:3 6:8

MNIST Classification in Jupyter Notebook

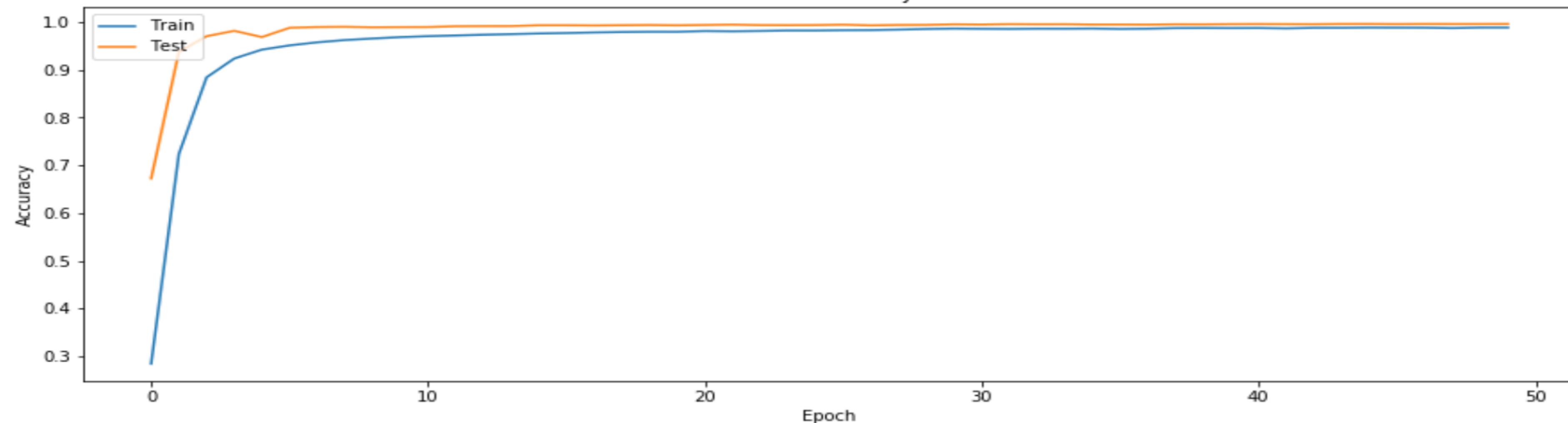
Now, let's try to train the network for 50 epochs:

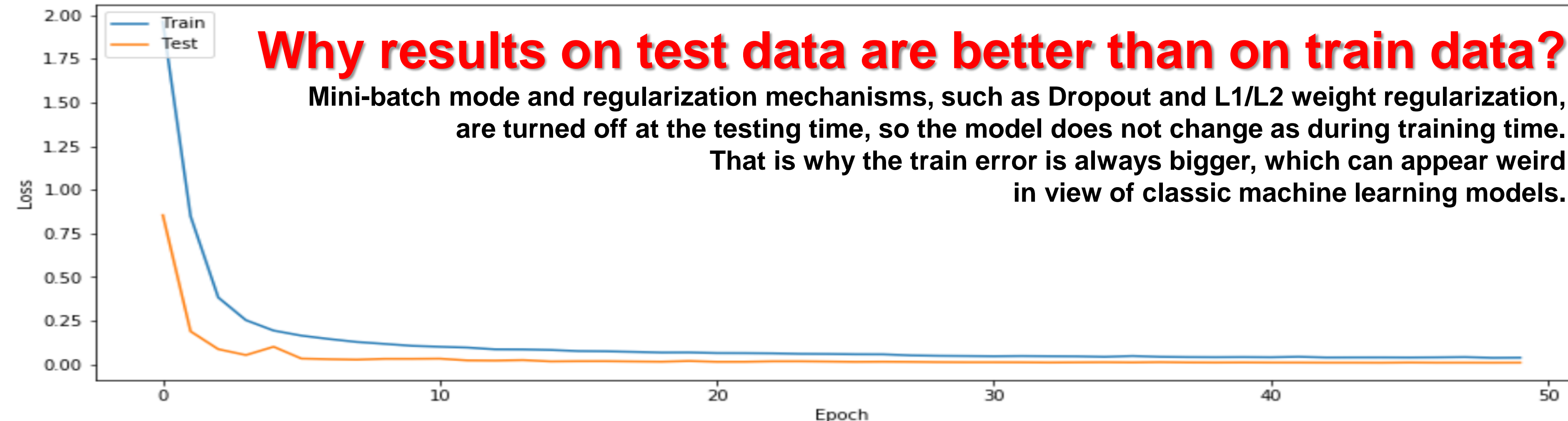
```
Epoch 1/50
117/117 [==============================] - 271s 2s/step - loss: 1.9644 - acc: 0.2841 - val_loss: 0.8554 - val_acc: 0.6723
Epoch 2/50
117/117 [==============================] - 270s 2s/step - loss: 0.8482 - acc: 0.7236 - val_loss: 0.1902 - val_acc: 0.9377
Epoch 3/50
117/117 [==============================] - 391s 3s/step - loss: 0.3834 - acc: 0.8843 - val_loss: 0.0880 - val_acc: 0.9706
Epoch 4/50
117/117 [==============================] - 691s 6s/step - loss: 0.2535 - acc: 0.9239 - val_loss: 0.0543 - val_acc: 0.9819
                                                                                         ⋮                ⋮

Epoch 00037: ReduceLROnPlateau reducing learning rate to 0.25.
Epoch 38/50
117/117 [==============================] - 352s 3s/step - loss: 0.0425 - acc: 0.9877 - val_loss: 0.0122 - val_acc: 0.9956
Epoch 39/50
117/117 [==============================] - 351s 3s/step - loss: 0.0418 - acc: 0.9878 - val_loss: 0.0117 - val_acc: 0.9955
Epoch 40/50
117/117 [==============================] - 351s 3s/step - loss: 0.0425 - acc: 0.9877 - val_loss: 0.0122 - val_acc: 0.9959
Epoch 41/50
117/117 [==============================] - 360s 3s/step - loss: 0.0416 - acc: 0.9879 - val_loss: 0.0116 - val_acc: 0.9961
Epoch 42/50
117/117 [==============================] - 349s 3s/step - loss: 0.0446 - acc: 0.9871 - val_loss: 0.0115 - val_acc: 0.9959
Epoch 43/50
117/117 [==============================] - 353s 3s/step - loss: 0.0401 - acc: 0.9882 - val_loss: 0.0110 - val_acc: 0.9958
Epoch 44/50
117/117 [==============================] - 354s 3s/step - loss: 0.0407 - acc: 0.9883 - val_loss: 0.0110 - val_acc: 0.9963
Epoch 45/50
117/117 [==============================] - 347s 3s/step - loss: 0.0406 - acc: 0.9887 - val_loss: 0.0106 - val_acc: 0.9963
Epoch 46/50
117/117 [==============================] - 353s 3s/step - loss: 0.0403 - acc: 0.9885 - val_loss: 0.0118 - val_acc: 0.9960
Epoch 47/50
117/117 [==============================] - 1063s 9s/step - loss: 0.0414 - acc: 0.9885 - val_loss: 0.0109 - val_acc: 0.9963
Epoch 48/50
117/117 [==============================] - 949s 8s/step - loss: 0.0427 - acc: 0.9877 - val_loss: 0.0111 - val_acc: 0.9962
Epoch 49/50
117/117 [==============================] - 909s 8s/step - loss: 0.0386 - acc: 0.9887 - val_loss: 0.0108 - val_acc: 0.9962

Epoch 00049: ReduceLROnPlateau reducing learning rate to 0.125.
Epoch 50/50
117/117 [==============================] - 891s 8s/step - loss: 0.0393 - acc: 0.9887 - val_loss: 0.0111 - val_acc: 0.9963
```

MNIST Classification in Jupyter Notebook

Graphs of learning convergence (accuracy) and error minimization (loss):

```
Test loss: 0.011101936267607016
Test accuracy: 0.9962999820709229
```



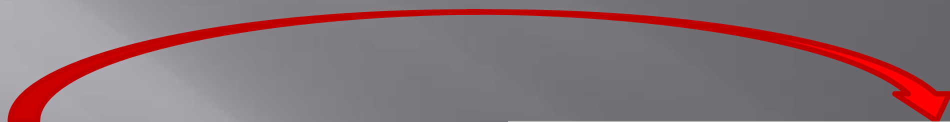


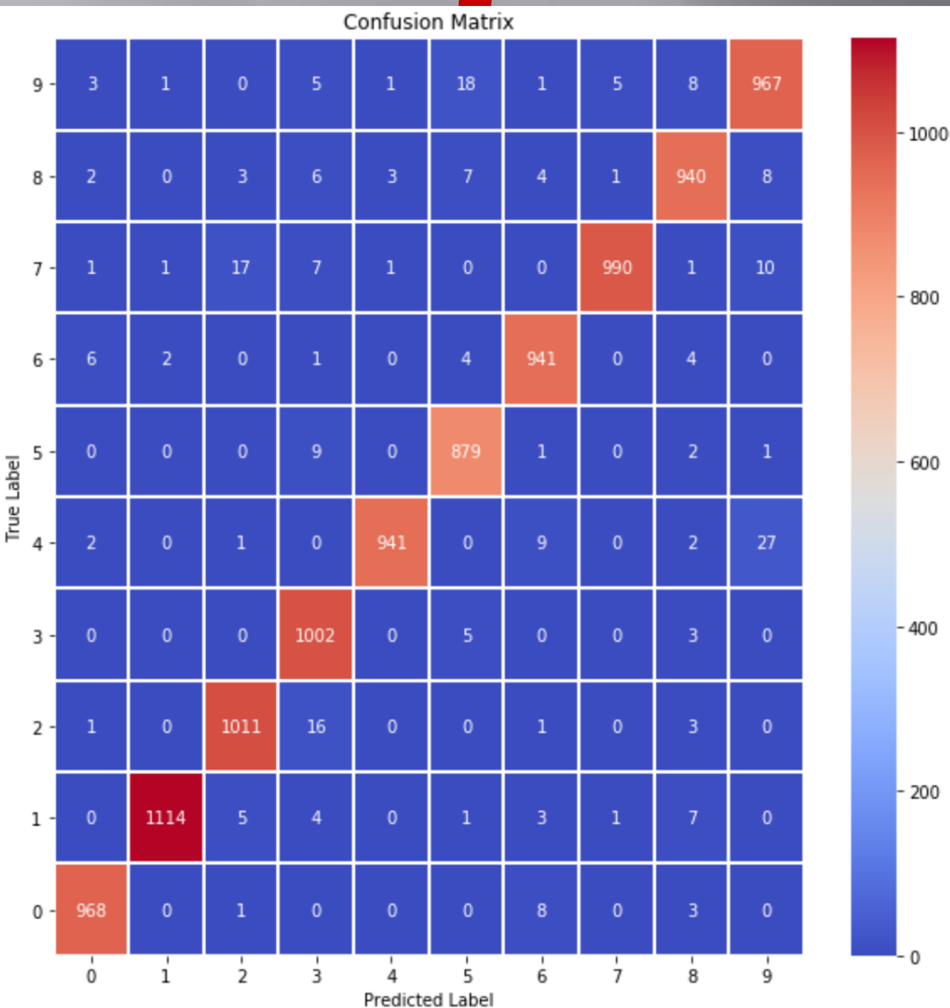
Why results on test data are better than on train data?

Mini-batch mode and regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at the testing time, so the model does not change as during training time. That is why the train error is always bigger, which can appear weird in view of classic machine learning models.

MNIST Classification in Jupyter Notebook

The confusion matrix has also improved: more examples have migrated towards the diagonal (correct classifications) from the other regions:





Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	3	1	0	5	1	18	1	5	8	967
8	2	0	3	6	3	7	4	1	940	8
7	1	1	17	7	1	0	0	990	1	10
6	6	2	0	1	0	4	941	0	4	0
5	0	0	0	9	0	879	1	0	2	1
4	2	0	1	0	941	0	9	0	2	27
3	0	0	0	1002	0	5	0	0	3	0
2	1	0	1011	16	0	0	1	0	3	0
1	0	1114	5	4	0	1	3	1	7	0
0	968	0	1	0	0	0	8	0	3	0

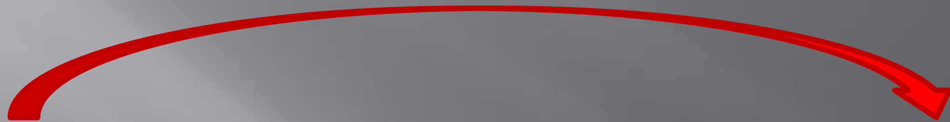
Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	0	0	0	0	5	1	0	2	1	1000
8	0	0	1	1	0	1	0	0	970	1
7	0	2	2	0	0	0	0	1023	0	1
6	3	0	1	0	0	0	953	0	1	0
5	0	0	0	6	0	885	1	0	0	0
4	0	0	0	0	981	0	0	0	0	1
3	0	0	0	1010	0	0	0	0	0	0
2	1	0	1029	0	0	0	0	2	0	0
1	0	1132	0	1	0	0	1	1	0	0
0	980	0	0	0	0	0	0	0	0	0

Colour scale: 0, 200, 400, 600, 800, 1000

MNIST Classification in Jupyter Notebook

The number and the accuracy of correctly classified examples for all individual classes increase have risen:



	precision	recall	f1-score	support
0	0.98	0.99	0.99	980
1	1.00	0.98	0.99	1135
2	0.97	0.98	0.98	1032
3	0.95	0.99	0.97	1010
4	0.99	0.96	0.98	982
5	0.96	0.99	0.97	892
6	0.97	0.98	0.98	958
7	0.99	0.96	0.98	1028
8	0.97	0.97	0.97	974
9	0.95	0.96	0.96	1009
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	1.00	1.00	1.00	1032
3	0.99	1.00	1.00	1010
4	0.99	1.00	1.00	982
5	1.00	0.99	0.99	892
6	1.00	0.99	1.00	958
7	1.00	1.00	1.00	1028
8	1.00	1.00	1.00	974
9	1.00	0.99	0.99	1009
accuracy			1.00	10000
macro avg	1.00	1.00	1.00	10000
weighted avg	1.00	1.00	1.00	10000

However, we can see that the process of network training is not over yet and should be continued for several dozen epochs.

MNIST Classification in Jupyter Notebook

The number of misclassified examples after 50 epochs compared to 3 epochs has dropped from 247 to 37 out of 10,000 test examples, resulting in an error of 0.37%. Here are all misclassified examples:

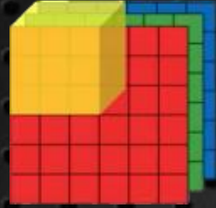
```
Number of misclassified examples:  37
Misclassified examples:
[ 359  445  582  659  674  716  947 1039 1232 1393 1737 1878 1901 2035
 2130 2182 2414 2462 2597 2939 3225 3422 3762 4176 4620 4761 5654 5937
 6558 6571 6576 8316 8376 8408 9530 9642 9729]
Misclassified images (original class : predicted class):
```

9:4 6:0 8:2 2:7 5:3 1:7 8:9 7:1 9:4 5:3

5:3 8:3 9:4 5:3 4:9 1:3 9:4 2:0 5:3 9:5

7:9 6:0 6:8 2:7 6:0 9:4 7:2 5:3 6:2 9:7

7:1 7:2 1:6 8:5 9:8 9:7 5:6



Classification of Images

An example of the CNN model implementation using the CIFAR-10 dataset.

jupyter

CIFAR-10 Classification in Jupyter

Classification of images 32 x 32 pixels to 10 classes (3 learning epochs):

Class [6] Class [9] Class [9] Class [4] Class [1] Class [1] Class [2] Class [7] Class [8] Class [3]

Create the network structure

In [6]:
```python
# Define the sequential Keras model composed of a few layers
model = Sequential()   # establishes the type of the network model
# Conv2D - creates a convolutional layer (https://keras.io/layers/convolutional/#conv2d) with
# filters - specified number of convolutional filters
# kernel_size - defines the frame (sliding window) size where the convolutional filter is implemented
# activation - sets the activation function for this layers, here ReLU
# input_shape - defines the shape of the input matrix (vector), here input_shape = (1, img_rows, img_cols)
model.add(Conv2D(64, kernel_size=(3, 3),activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
# MaxPooling2D pools the max value from the frame (sliding window) of 2 x 2 size
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Implements the drop out with the probability of 0.25
model.add(Conv2D(128, (3, 3), activation='relu',padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(256, (3, 3), activation='relu',padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.35))
# Finish the convolutional model and flatten the layer which does not affect the batch size.
model.add(Flatten())
# Use a dense layer (MLP) consisting of 256 neurons with relu activation functions
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(num_classes, activation='softmax'))
```

CIFAR-10 Classification in Jupyter

Compilation, optimization , data augmentation (generation) and training:

Compile and train the network

In [7]:

```
# Compile the model using optimizer
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['acc']) # List of metrics to be evaluated by the model during training and testing: https://keras.io/m

# Learning rate reduction durint the training process: https://keras.io/callbacks/#reducelronplateau
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc', # quantity to be monitored (val_loss)
                                            factor=0.5, # factor by which the learning rate will be reduced. new_lr = lr * fa
                                            patience=5, # number of epochs that produced the monitored quantity with no impro
                                            verbose=1,  # 0: quiet, 1: update messages.
                                            min_lr=0.001) # lower bound on the learning rate

# Augmentation of training data. It generates batches of tensor image data with real-time data augmentation. The data will be
datagen = ImageDataGenerator(
        rotation_range=10,       # rotate images in degrees up to the given degrees
        width_shift_range=0.1,   # shift images horizontally
        height_shift_range=0.1,  # shift images vertically
        horizontal_flip=True)    # flip images (left<->right)
# Computes the internal data stats related to the data-dependent transformations, based on an array of samples x_train
datagen.fit(x_train)

# Train the model, validate, evaluate, and present scores
history=model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                            epochs=epochs,
                            steps_per_epoch=x_train.shape[0]//batch_size,  # no of mini-batches
                            validation_data=(x_test, y_test),
                            verbose=1,
                            callbacks=[learning_rate_reduction])
```

jupyter

CIFAR-10 Classification in Jupyter

Results of training after three training epochs:

```
Test loss: 2.1507028507232664
Test accuracy: 0.2071000039577484
```

Model accuracy

Train
Test

Accuracy

0.26
0.24
0.22
0.20
0.18
0.16
0.14
0.12

0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

Epoch

Model loss

Train
Test

Loss

2.30
2.25
2.20
2.15
2.10
2.05

0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

Epoch

CIFAR-10 Classification in Jupyter

Confusion (error) matrix after three training epochs:

	precision	recall	f1-score	support
0	0.20	0.62	0.30	1000
1	0.23	0.48	0.32	1000
2	0.00	0.00	0.00	1000
3	0.14	0.14	0.14	1000
4	0.00	0.00	0.00	1000
5	0.18	0.13	0.15	1000
6	0.50	0.00	0.01	1000
7	0.20	0.02	0.04	1000
8	0.21	0.37	0.27	1000
9	0.23	0.31	0.27	1000
accuracy			0.21	10000
macro avg	0.19	0.21	0.15	10000
weighted avg	0.19	0.21	0.15	10000

```
Number of misclassified examples:  7929
Misclassified examples:
[   0    3    4 ... 9994 9995 9999]
```

Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
9	174	158	0	9	0	10	0	2	337	310
8	400	101	0	4	0	27	0	0	366	102
7	254	203	0	71	0	43	0	24	117	288
6	177	204	0	295	0	112	4	35	95	78
5	338	221	0	121	0	127	0	21	103	69
4	245	157	0	241	0	101	4	17	136	99
3	335	202	0	140	0	153	0	10	75	85
2	363	233	0	108	0	109	0	10	105	72
1	159	483	0	8	0	18	0	2	194	136
0	617	94	0	3	0	10	0	0	183	93

Color scale: 0, 150, 300, 450, 600

We usually train such networks for minimum a few dozens of epochs to get satisfying results.

CIFAR-10 Classification in Jupyter

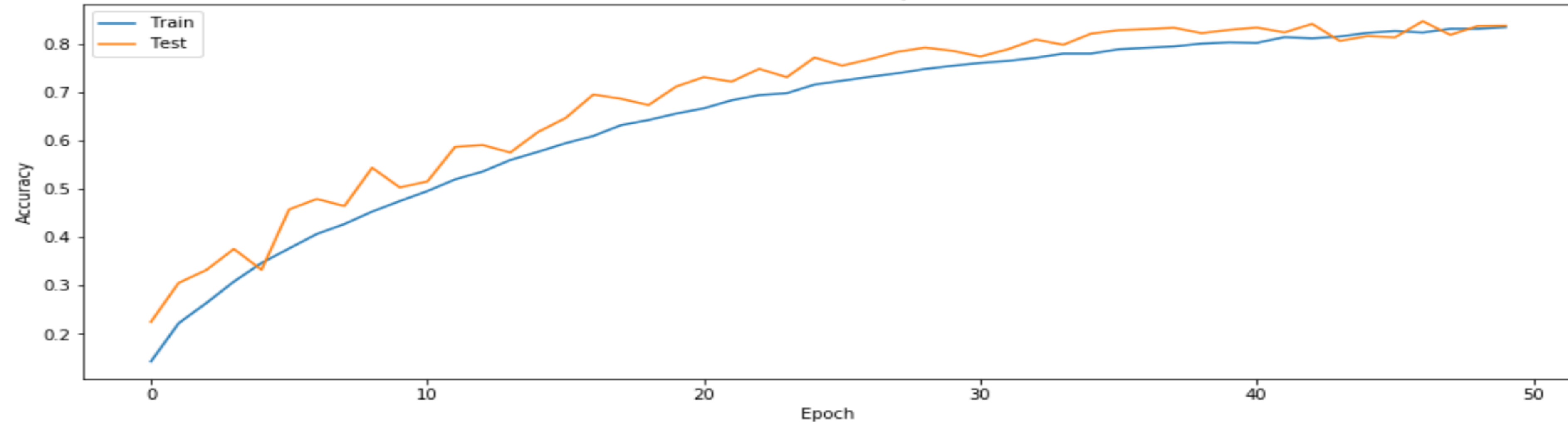
Let's train the network longer (50 epochs, a few hours) and as you can see the error (val_loss) systematically decreases, and the accuracy (val_acc) increases:

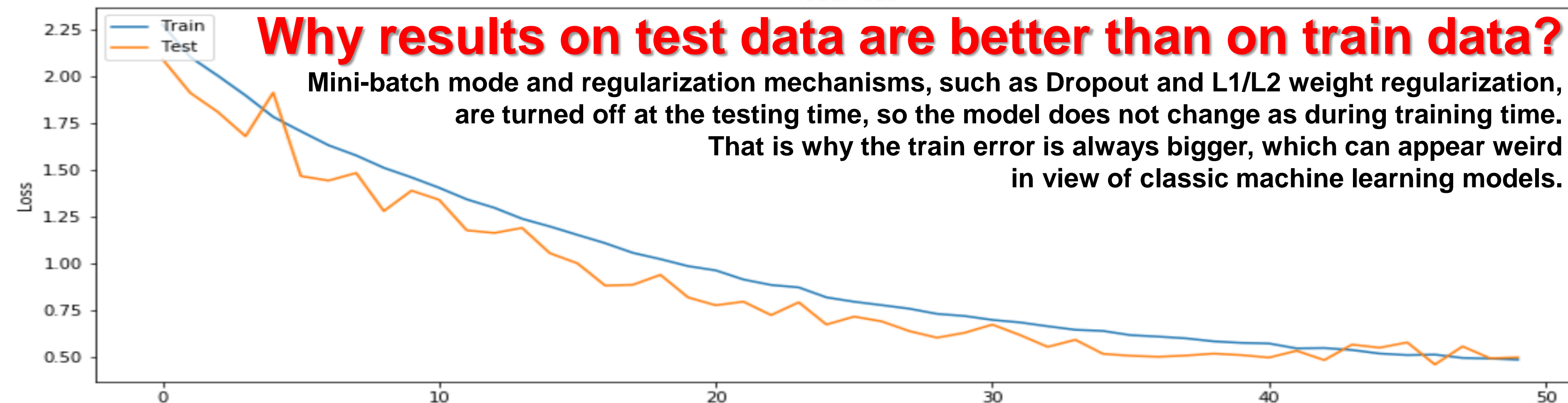
```
Epoch 1/50
97/97 [==============================] - 955s 10s/step - loss: 2.2744 - acc: 0.1426 - val_loss: 2.0892 - val_acc: 0.2247

                                                                                                    ⋮                       ⋮

Epoch 36/50
97/97 [==============================] - 751s 8s/step - loss: 0.6174 - acc: 0.7896 - val_loss: 0.5071 - val_acc: 0.8291
Epoch 37/50
97/97 [==============================] - 746s 8s/step - loss: 0.6093 - acc: 0.7926 - val_loss: 0.5017 - val_acc: 0.8312
Epoch 38/50
97/97 [==============================] - 842s 9s/step - loss: 0.5998 - acc: 0.7955 - val_loss: 0.5083 - val_acc: 0.8342
Epoch 39/50
97/97 [==============================] - 825s 9s/step - loss: 0.5840 - acc: 0.8012 - val_loss: 0.5187 - val_acc: 0.8230
Epoch 40/50
97/97 [==============================] - 784s 8s/step - loss: 0.5759 - acc: 0.8040 - val_loss: 0.5108 - val_acc: 0.8297
Epoch 41/50
97/97 [==============================] - 750s 8s/step - loss: 0.5727 - acc: 0.8028 - val_loss: 0.4975 - val_acc: 0.8346
Epoch 42/50
97/97 [==============================] - 746s 8s/step - loss: 0.5466 - acc: 0.8147 - val_loss: 0.5339 - val_acc: 0.8244
Epoch 43/50
97/97 [==============================] - 737s 8s/step - loss: 0.5483 - acc: 0.8123 - val_loss: 0.4840 - val_acc: 0.8422
Epoch 44/50
97/97 [==============================] - 746s 8s/step - loss: 0.5380 - acc: 0.8161 - val_loss: 0.5666 - val_acc: 0.8069
Epoch 45/50
97/97 [==============================] - 732s 8s/step - loss: 0.5195 - acc: 0.8235 - val_loss: 0.5502 - val_acc: 0.8169
Epoch 46/50
97/97 [==============================] - 688s 7s/step - loss: 0.5108 - acc: 0.8273 - val_loss: 0.5784 - val_acc: 0.8143
Epoch 47/50
97/97 [==============================] - 292s 3s/step - loss: 0.5134 - acc: 0.8242 - val_loss: 0.4603 - val_acc: 0.8477
Epoch 48/50
97/97 [==============================] - 296s 3s/step - loss: 0.4951 - acc: 0.8319 - val_loss: 0.5570 - val_acc: 0.8194
Epoch 49/50
97/97 [==============================] - 282s 3s/step - loss: 0.4917 - acc: 0.8320 - val_loss: 0.4934 - val_acc: 0.8380
Epoch 50/50
97/97 [==============================] - 280s 3s/step - loss: 0.4857 - acc: 0.8353 - val_loss: 0.4985 - val_acc: 0.8385
```

CIFAR-10 Classification in Jupyter

The charts of accuracy and loss show the right convergence process:

```
Test loss: 0.4984995872974396
Test accuracy: 0.8385000228881836
```

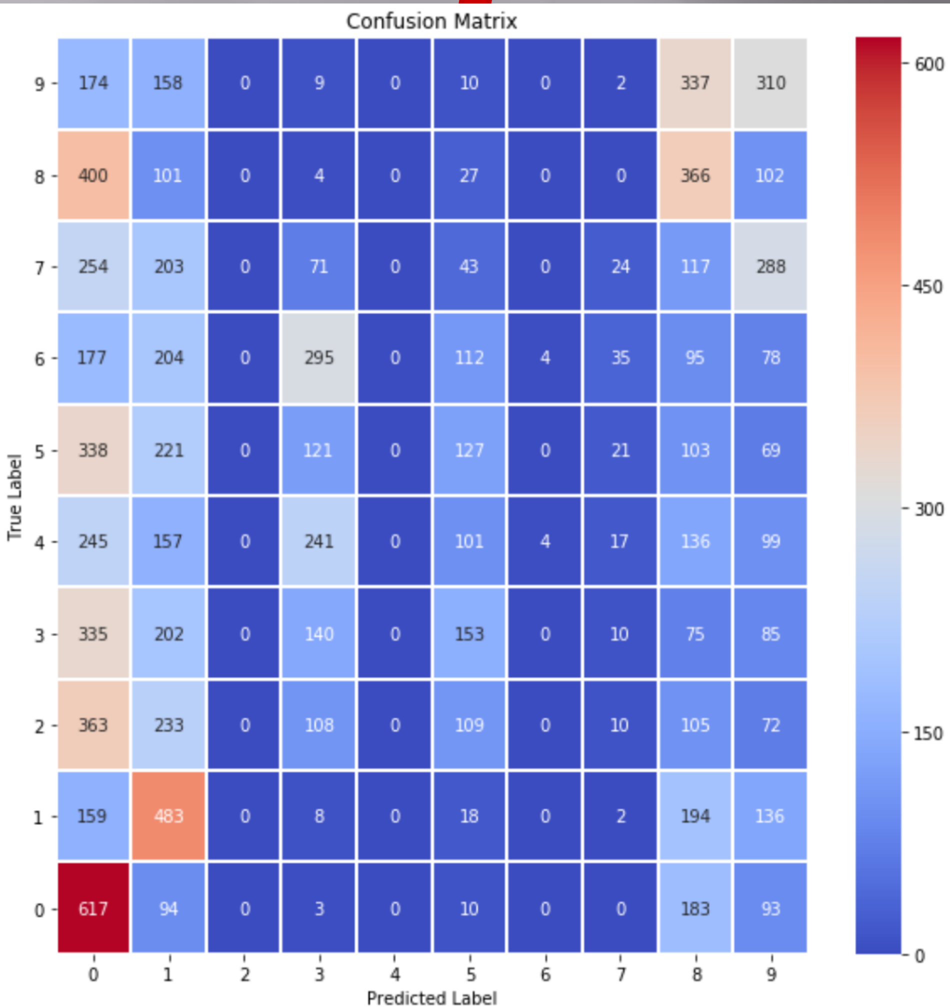


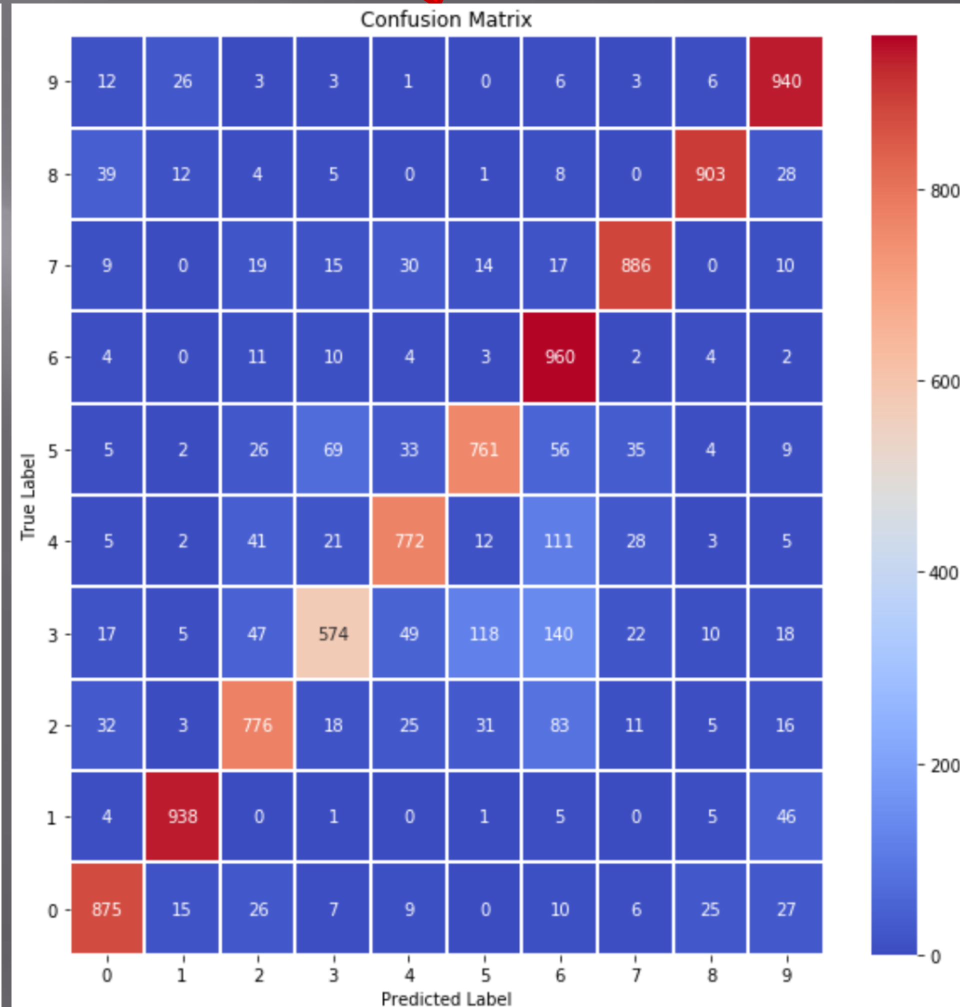


jupyter

CIFAR-10 Classification in Jupyter

The confusion matrix has also improved: more examples have migrated towards the diagonal (correct classifications) from the other regions:





CIFAR-10 Classification in Jupyter

The number and the accuracy of correctly classified examples for all individual classes have increased significantly:

	precision	recall	f1-score	support
0	0.20	0.62	0.30	1000
1	0.23	0.48	0.32	1000
2	0.00	0.00	0.00	1000
3	0.14	0.14	0.14	1000
4	0.00	0.00	0.00	1000
5	0.18	0.13	0.15	1000
6	0.50	0.00	0.01	1000
7	0.20	0.02	0.04	1000
8	0.21	0.37	0.27	1000
9	0.23	0.31	0.27	1000
accuracy			0.21	10000
macro avg	0.19	0.21	0.15	10000
weighted avg	0.19	0.21	0.15	10000

	precision	recall	f1-score	support
0	0.87	0.88	0.87	1000
1	0.94	0.94	0.94	1000
2	0.81	0.78	0.79	1000
3	0.79	0.57	0.67	1000
4	0.84	0.77	0.80	1000
5	0.81	0.76	0.78	1000
6	0.69	0.96	0.80	1000
7	0.89	0.89	0.89	1000
8	0.94	0.90	0.92	1000
9	0.85	0.94	0.89	1000
accuracy			0.84	10000
macro avg	0.84	0.84	0.84	10000
weighted avg	0.84	0.84	0.84	10000

However, we can see that the process of network training is not over yet and should be continued for several dozen epochs.

CIFAR-10 Classification in Jupyter

Examples of misclassifications after 50 training epochs for a test set of 10,000 examples: The number of misclassifications decreased from 7929 after 3 epochs to 1615 after 50 epochs.

```
Number of misclassified examples:  7929
Misclassified examples:
[   0    3    4 ... 9994 9995 9999]
```

```
Number of misclassified examples:  1615
Misclassified examples:
[   9   15   24 ... 9982 9985 9996]
```

We can see that in the case of this training set, the convolution network should be taught much longer (16.15% of incorrect classifications remain) or the structure or the hyperparameters of the model should be changed.

CIFAR-10 Classification in Jupyter

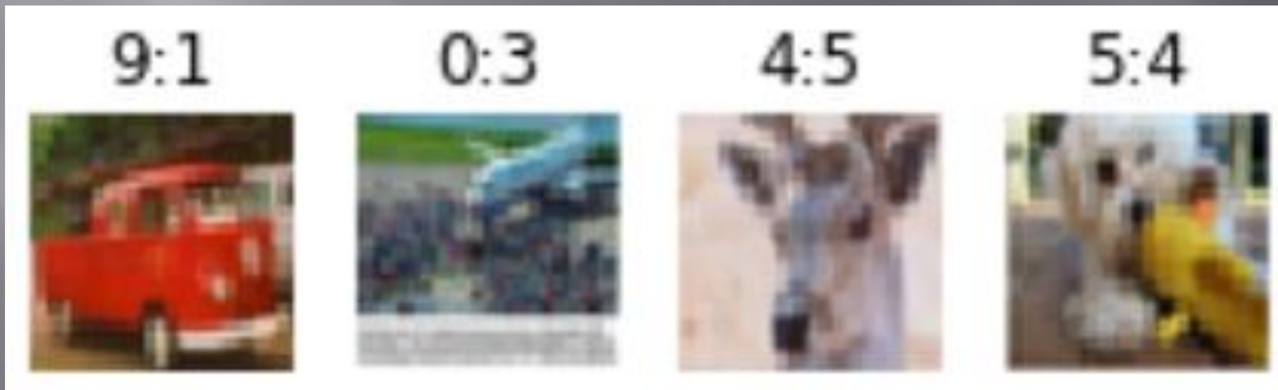
Samples of misclassified examples:

1:9	8:6	5:4	4:6	2:1	1:9	0:2	7:3	4:6	3:5
3:6	5:7	3:6	3:5	4:6	2:6	0:2	3:6	9:8	7:2
2:6	8:0	4:2	4:6	4:3	5:6	2:3	0:8	7:4	3:4
7:3	3:2	0:8	7:3	0:8	8:0	3:5	3:2	3:6	8:1
1:9	6:2	8:0	0:2	7:4	3:2	3:6	7:4	3:5	8:0

Class	Label
airplane	0
automobile	1
bird	2
cat	3
deer	4
dog	5
frog	6
horse	7
ship	8
truck	9

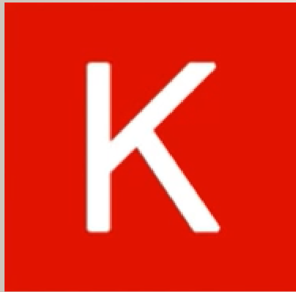
CIFAR-10 Classification in Jupyter

Samples of misclassified examples:

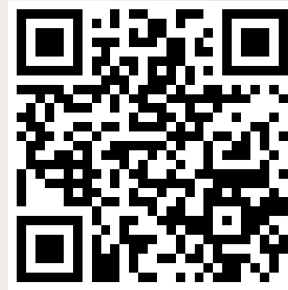


airplane	0
automobile	1
bird	2
cat	3
deer	4
dog	5
frog	6
horse	7
ship	8
truck	9

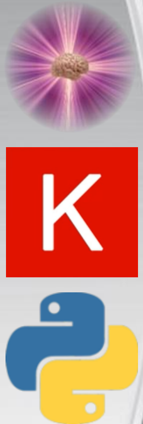
Let's start with powerful computations!



- ✓ Questions?
- ✓ Remarks?
- ✓ Wishes?

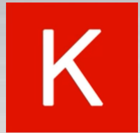


BIBLIOGRAPY



1. Francois Chollet, “Deep learning with Python”, Manning Publications Co., 2018.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, MIT Press, 2016, ISBN 978-1-59327-741-3.
3. Home page for this course: http://home.agh.edu.pl/~horzyk/lectures/ahdydci.php
4. Nikola K. Kasabov, Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
5. Holk Cruse, Neural Networks as Cybernetic Systems, 2nd and revised edition
6. R. Rojas, Neural Networks, Springer-Verlag, Berlin, 1996.
8. Convolutional Neural Network (Stanford)
9. Visualizing and Understanding Convolutional Networks, Zeiler, Fergus, ECCV 2014.

BIBLIOGRAPY



10. https://victorzhou.com/blog/keras-cnn-tutorial/
11. https://github.com/keras-team/keras/tree/master/examples
12. https://medium.com/@margaretmz/anaconda-jupyter-notebook-tensorflow-and-keras-b91f381405f8
13. https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html
14. http://coursera.org/specializations/tensorflow-in-practice
15. https://udacity.com/course/intro-to-tensorflow-for-deep-learning
16. https://www.youtube.com/watch?v=XNKeayZW4dY
17. Heatmaps: https://towardsdatascience.com/formatting-tips-for-correlation-heatmaps-in-seaborn-4478ef15d87f
18. MNIST example: https://medium.com/datadriveninvestor/image-processing-for-mnist-using-keras-f9a1021f6ef0

BIBLIOGRAPY

19. IBM: https://www.ibm.com/developerworks/library/ba-data-becomes-knowledge-1/index.html
20. NVIDIA: https://developer.nvidia.com/discover/convolutional-neural-network
21. JUPYTER: https://jupyter.org/
22. https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d