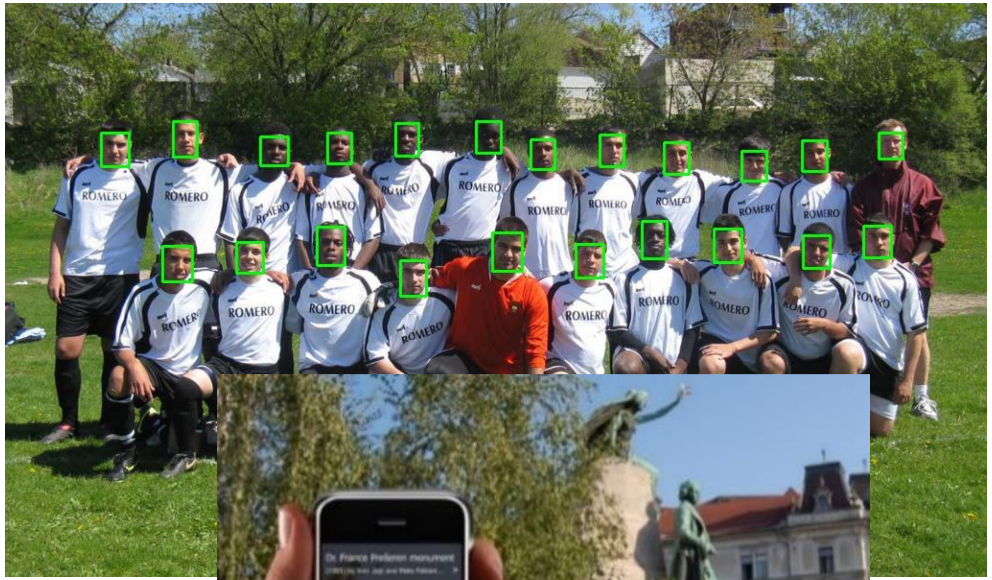
Development of intelligent systems (RInS)

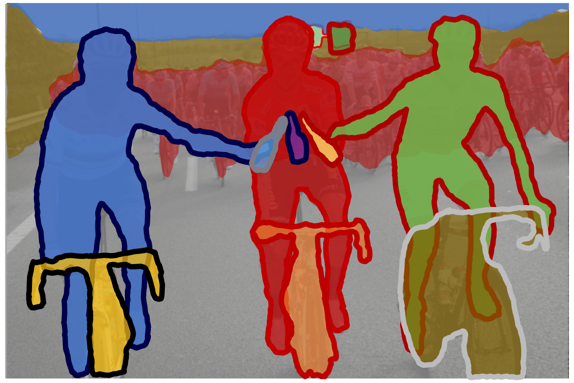
Object detection with Convolutional Neural Networks

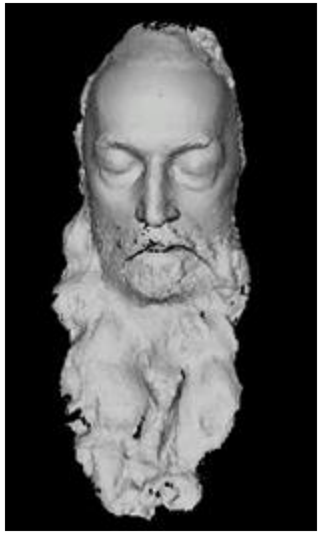
Danijel Skočaj
University of Ljubljana
Faculty of Computer and Information Science

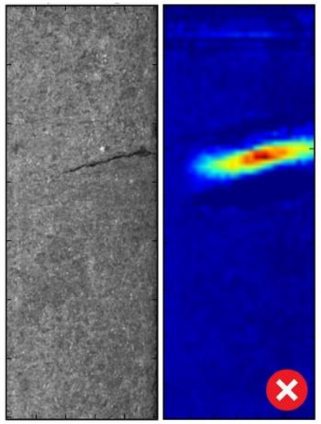
Academic year: 2023/24

Computer vision



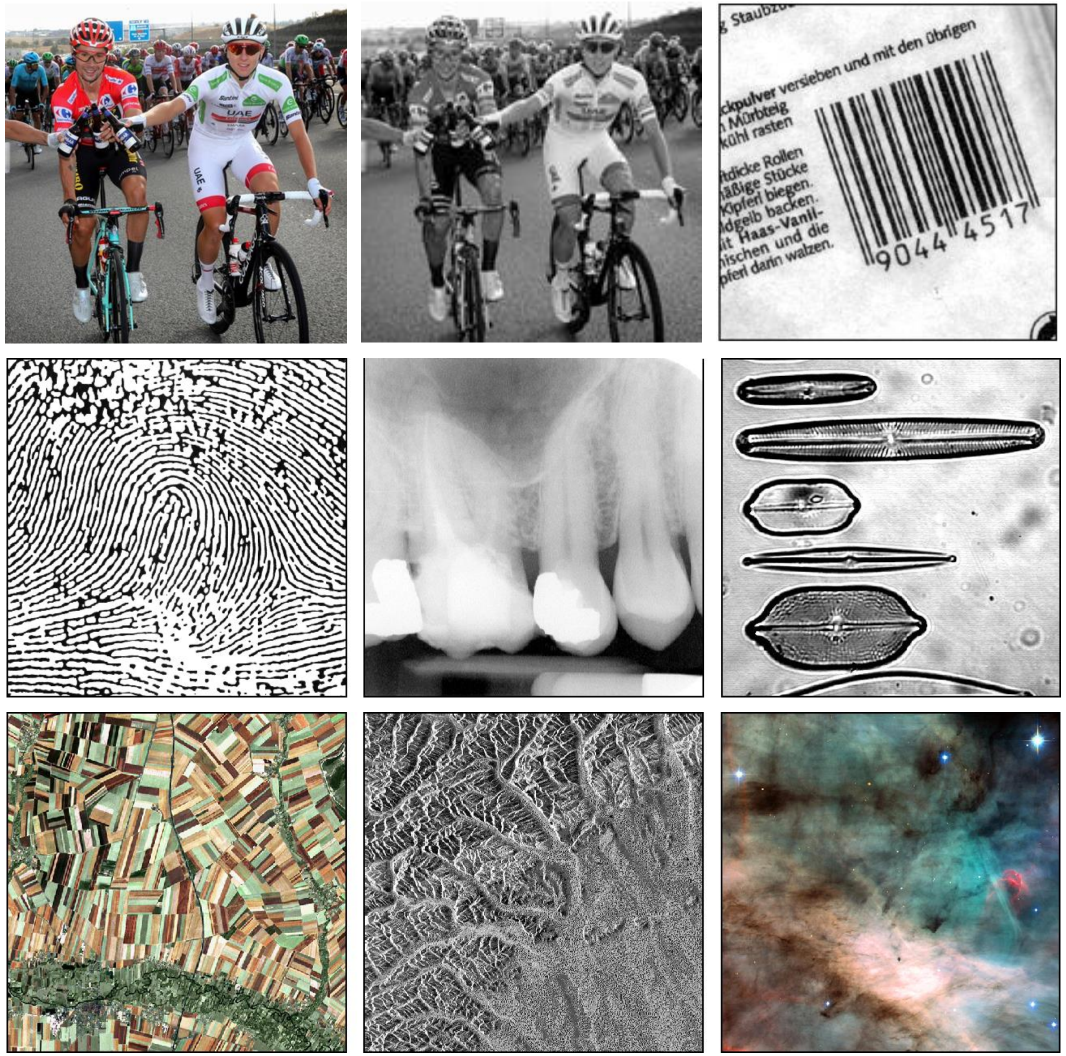






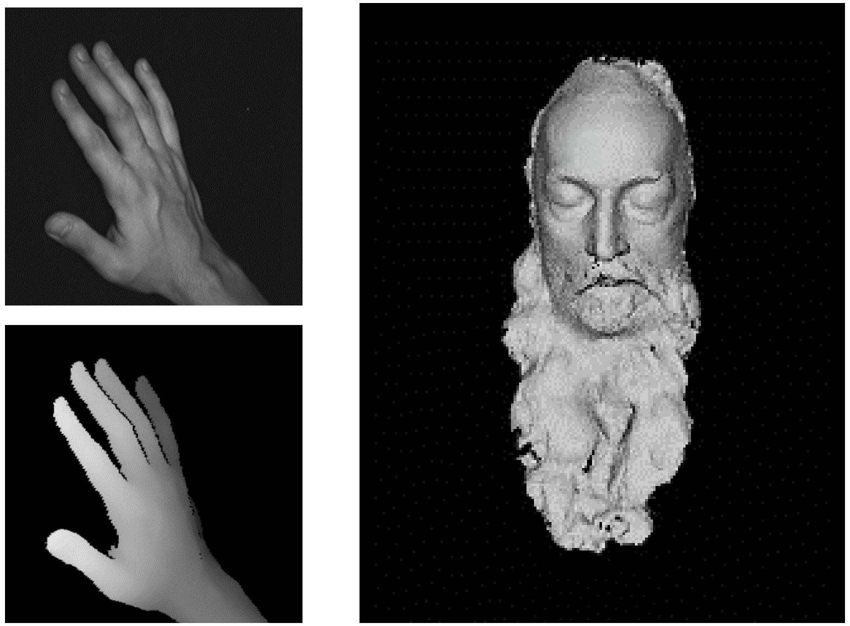
Visual information
Computer vision tasks

Visual information



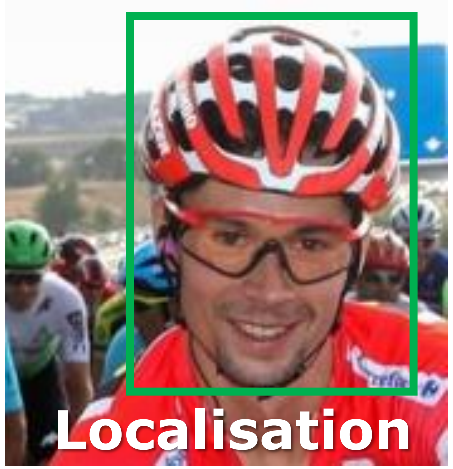
Images

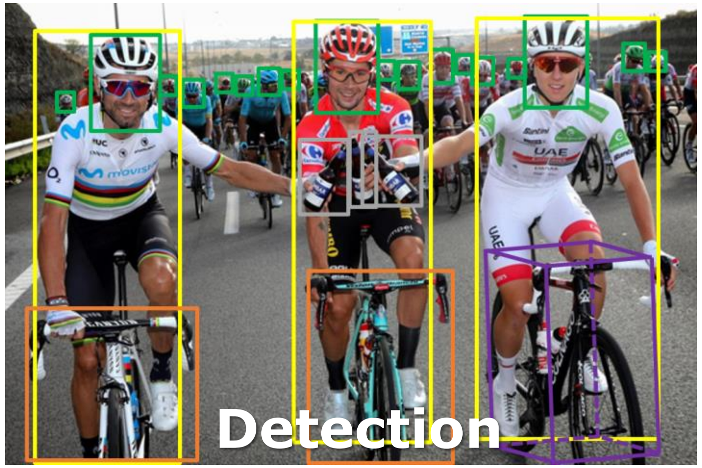
Video

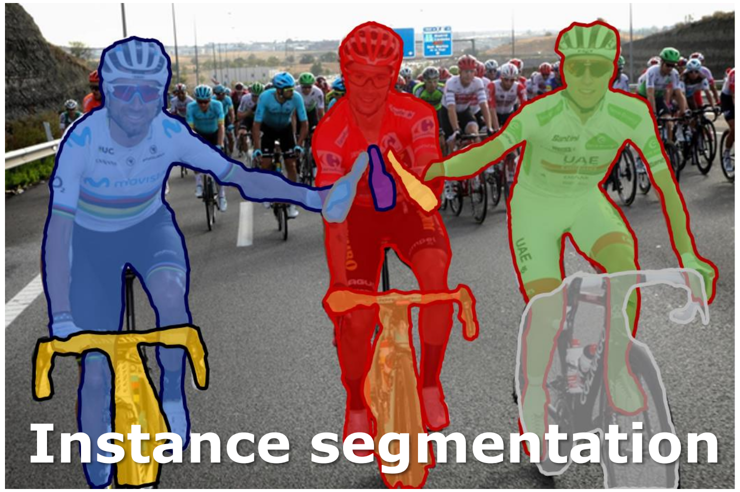


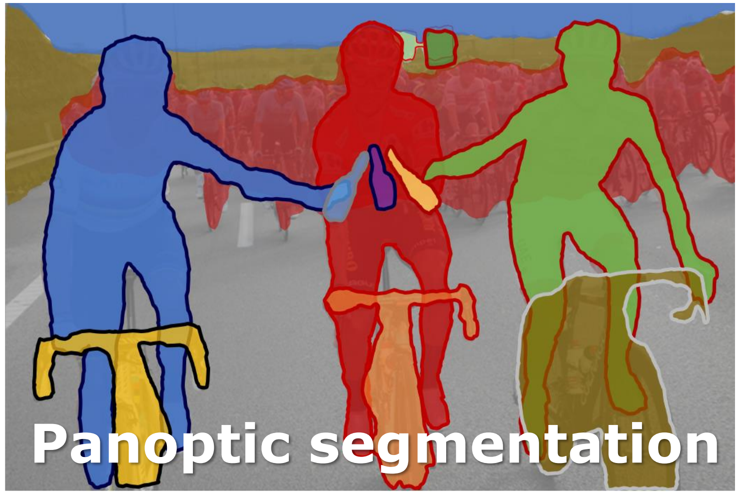
3D

Main computer vision tasks



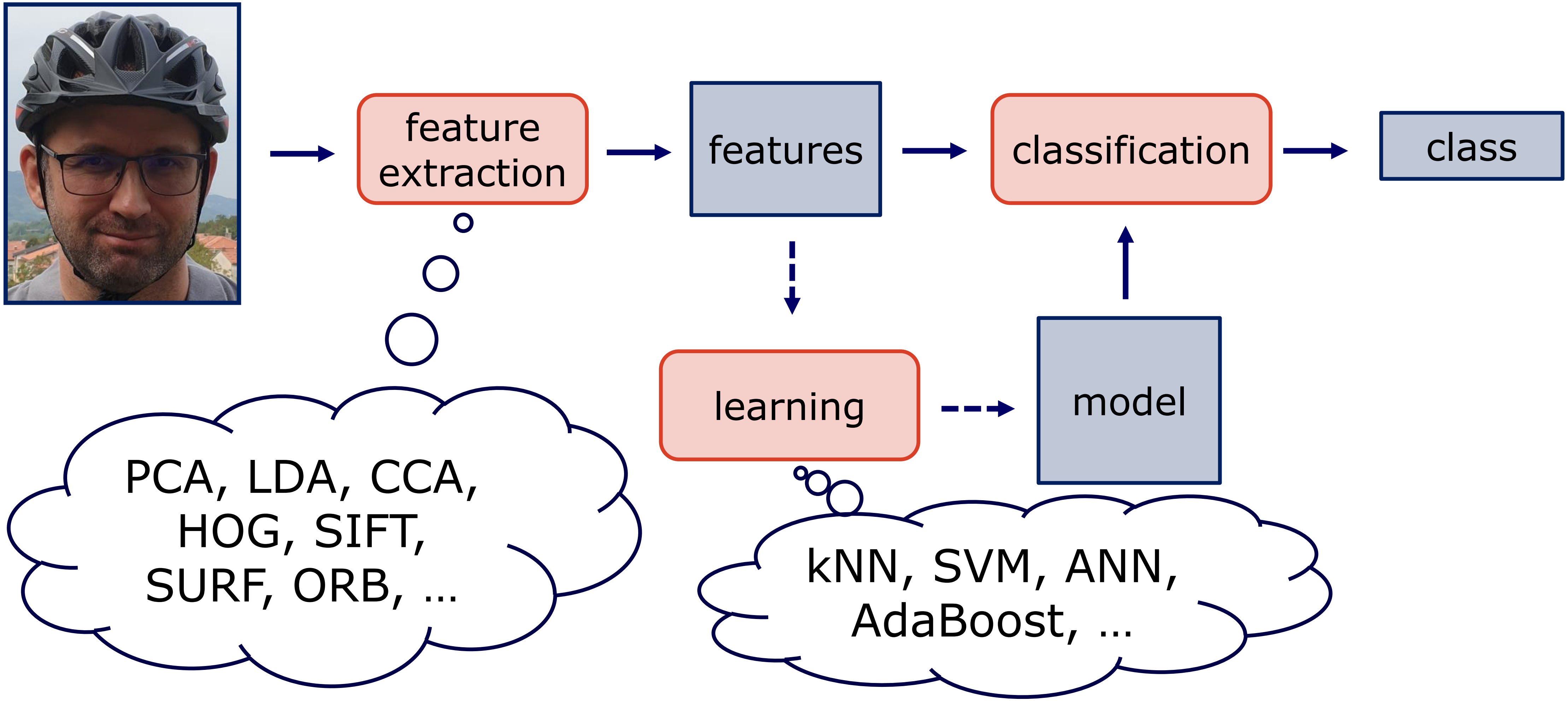






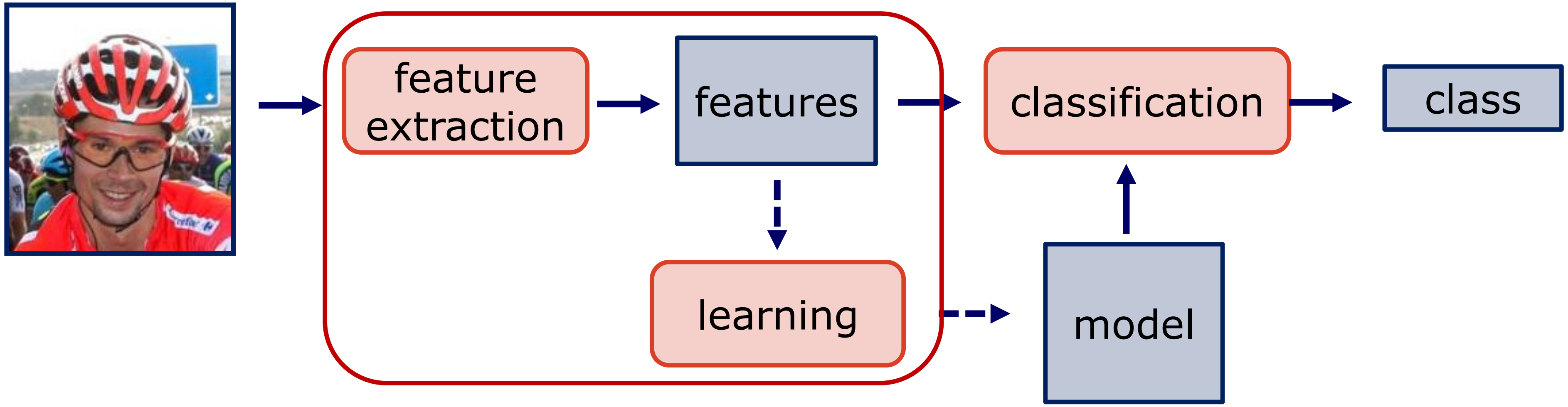
Machine learning in computer vision

- Conventional approach

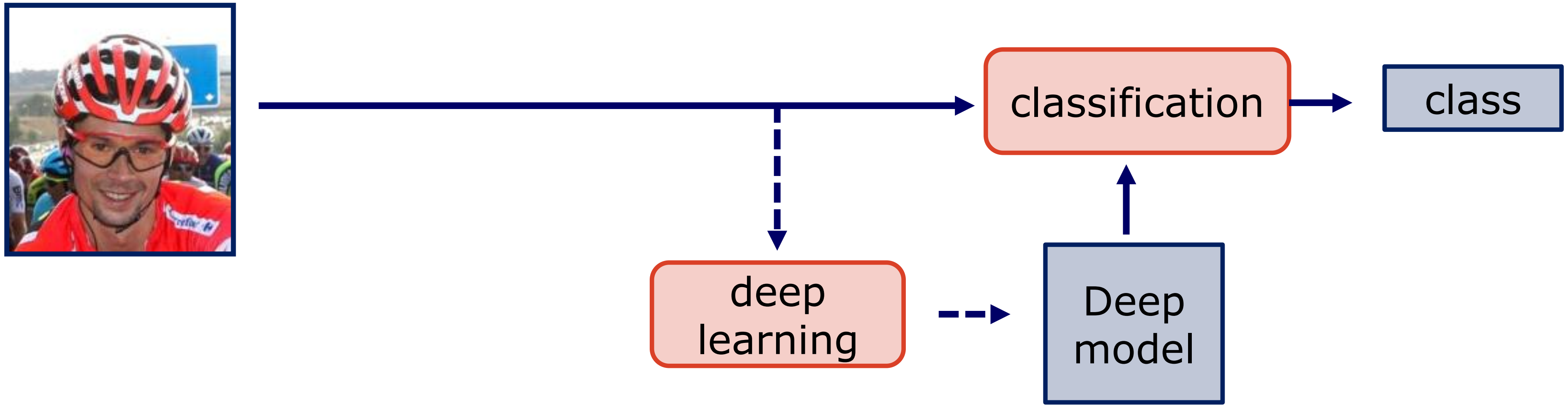


Deep learning in computer vision

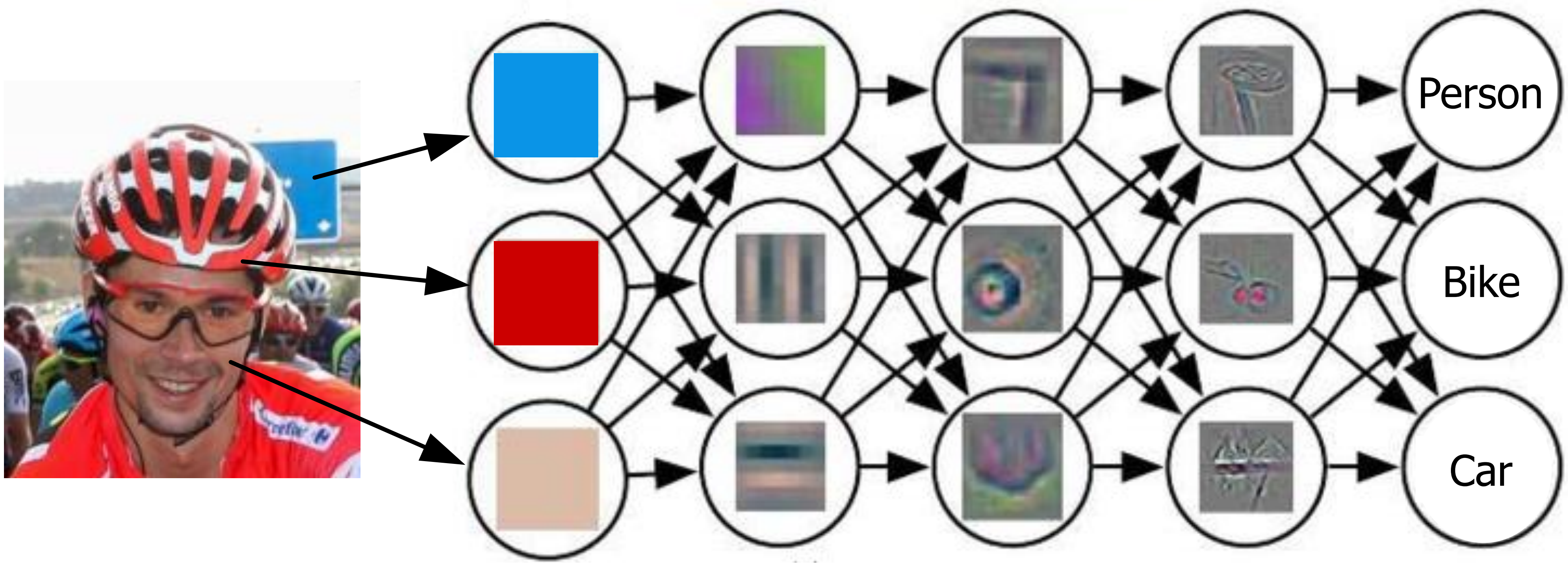
- Conventional machine learning approach in computer vision



- Deep learing approach

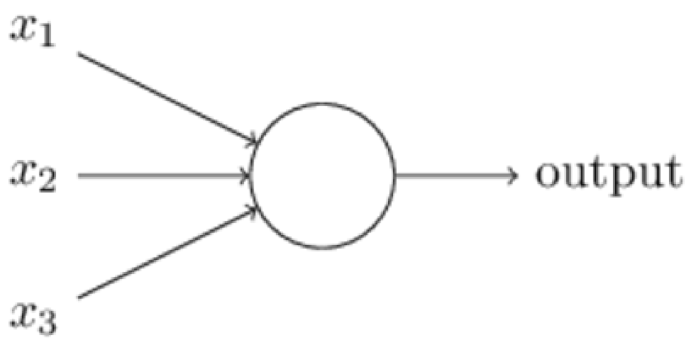


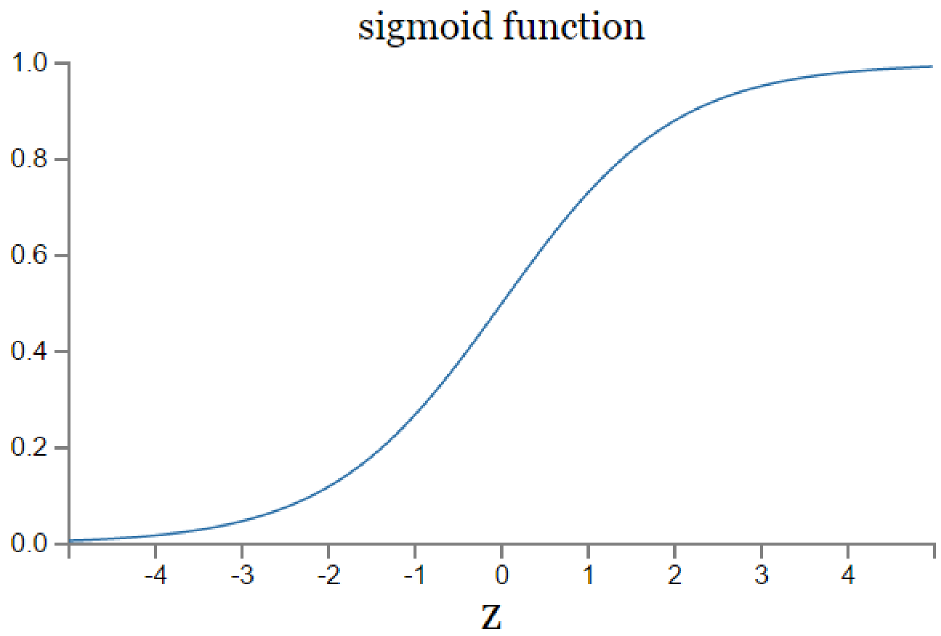
Deep learning – the main concept



Sigmoid neurons

- Real inputs and outputs from interval [0,1]





- Activation function: sigmoid function

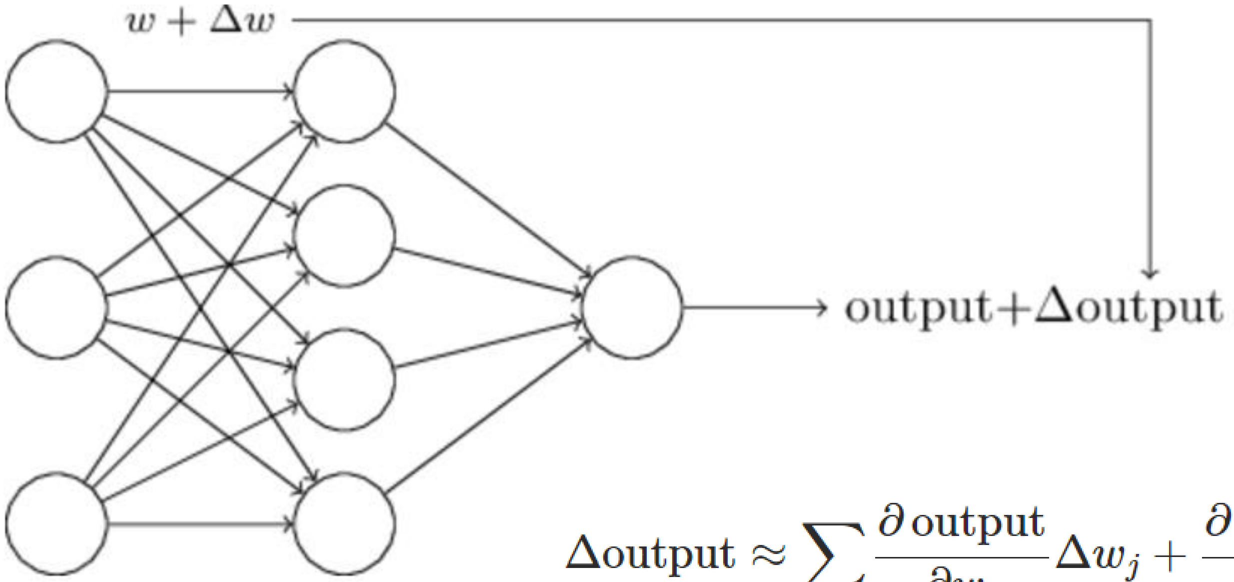
- *output* = $\dfrac{1}{1+\exp(-\sum_j w_j x_j - b)}$

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

$$\sigma(w \cdot x + b)$$

Sigmoid neurons

- Small changes in weights and biases causes small change in output

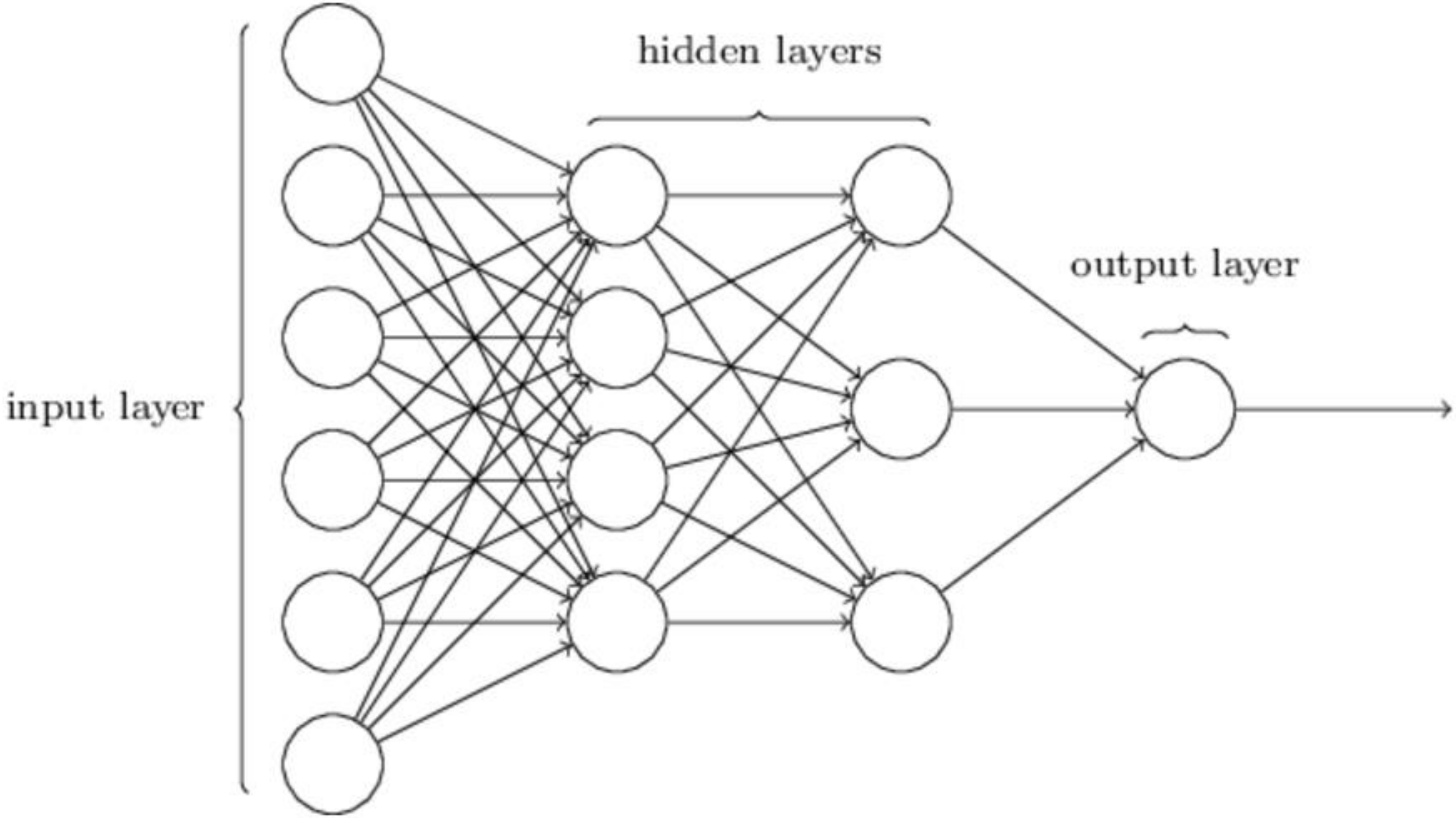


$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b$$

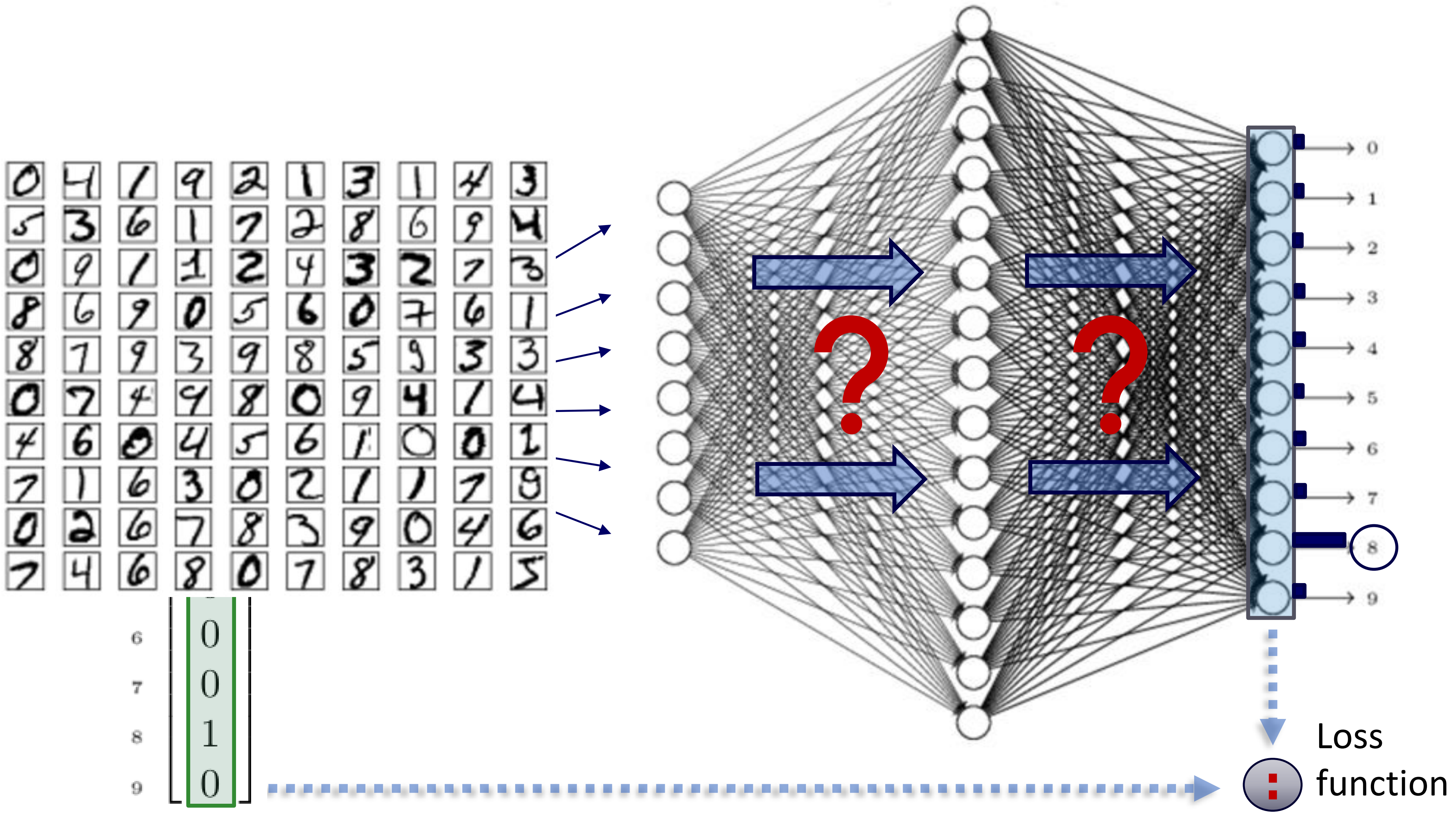
- Enables learning!

Feedfoward neural networks

- Network architecture:



Inference and training



Training!

Example code: Feedforward

- Code from http://neuralnetworksanddeeplearning.com/
  or https://github.com/mnielsen/neural-networks-and-deep-learning
- or https://github.com/MichalDanielDobrzanski/DeepLearningPython35 (for Python 3)

Nielsen, 2015

```
class Network(object):

     def __init__(self, sizes):
         self.num_layers = len(sizes)
         self.sizes = sizes
         self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
         self.weights = [np.random.randn(y, x)
                         for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a


def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

```
net = network.Network([784, 30, 10])
net.SGD(training_data, 5, 10, 3.0, test_data=test_data)
In [55]: x,y=test_data[0]

In [56]: net.feedforward(x)
Out[56]:
array([[  1.83408119e-03],
       [  5.94472468e-08],
       [  1.84785949e-03],
       [  6.85718810e-04],
       [  1.41399919e-05],
       [  5.40491233e-06],
       [  4.74332685e-09],
       [  9.97920007e-01],
       [  8.19370561e-05],
       [  6.65086583e-05]])

In [57]: y
Out[57]: 7
```

Loss function

- Given:

$y\Big($  $\Big) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ for all training images

- Loss function: $C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$
  - (mean sqare error – quadratic loss function)

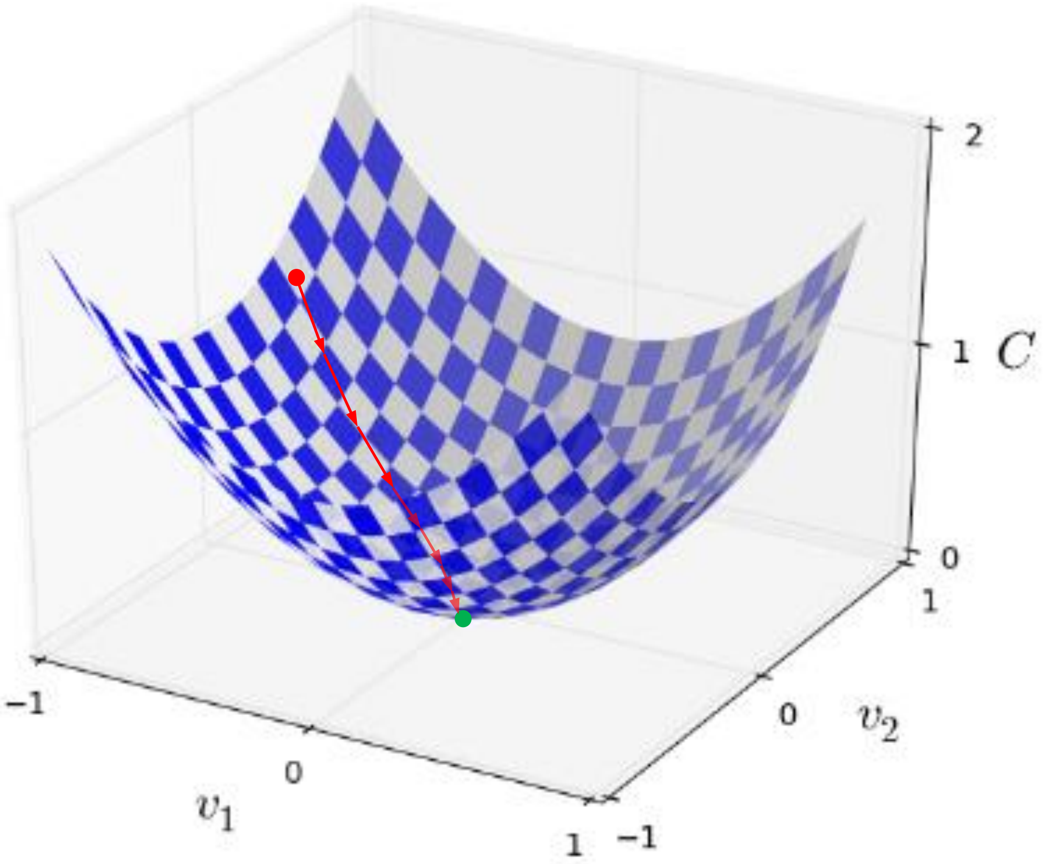
- Find weigths w and biases b that for given input x produce output a that minimizes Loss function C

Gradient descend

- Find minimum of $C(v_1, v_2)$
- Change of C: $\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 = \nabla C \cdot \Delta v \quad = -\eta\|\nabla C\|^2$
- Gradient of C: $\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- *Change* v *in the opposite direction of the gradient:* $\Delta v = -\eta\nabla C$

Learning rate

- Algorithm:
  - Initialize v
  - Until stopping criterium riched
    - Apply udate rule $v \to v' = v - \eta\nabla C.$



Gradient descend in neural networks

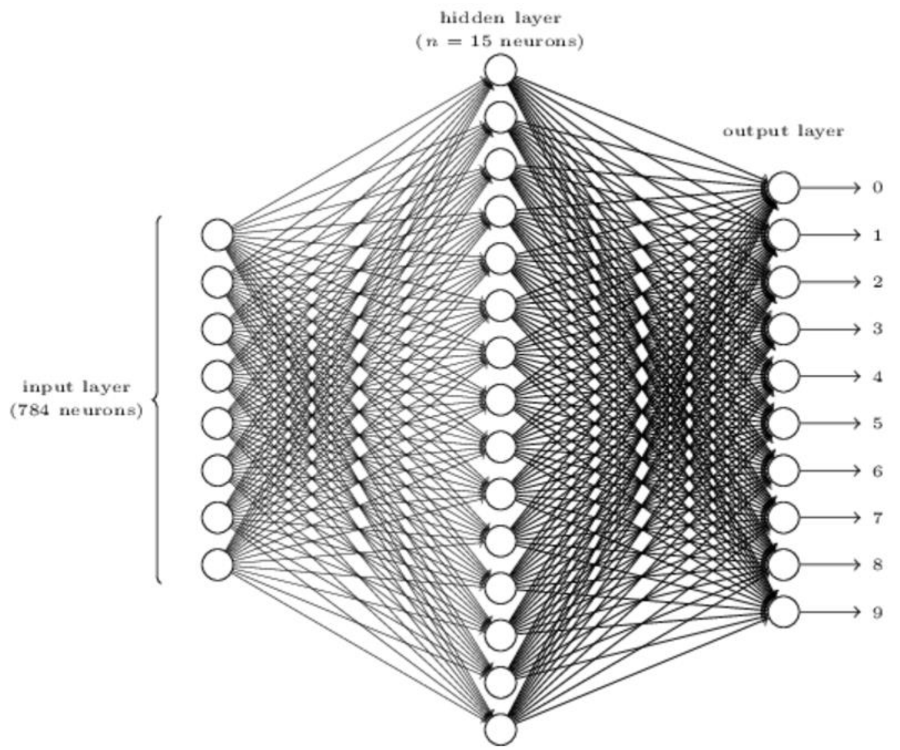
- Loss function $C(w,b)$

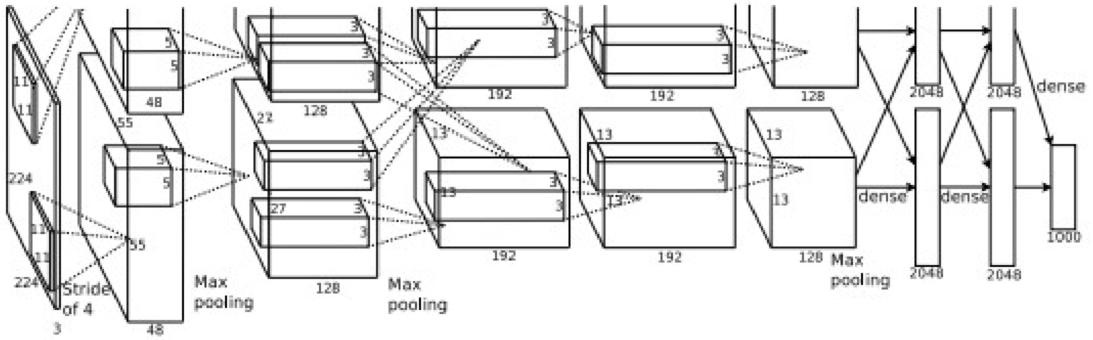
- Update rules:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

- Consider all training samples
- Very many parameters
  => computationaly very expensive

- Use Stochastic gradient descend instead





Example code: SGD

```python
def SGD(self, training_data, epochs, mini_batch_size, eta):
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
```

```python
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

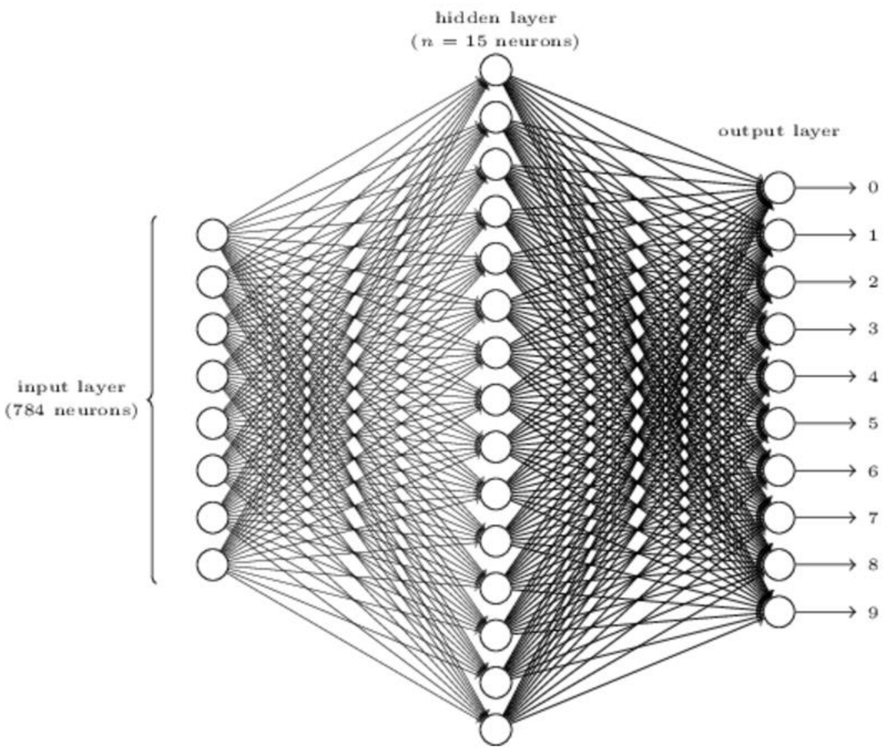
$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

Backpropagation

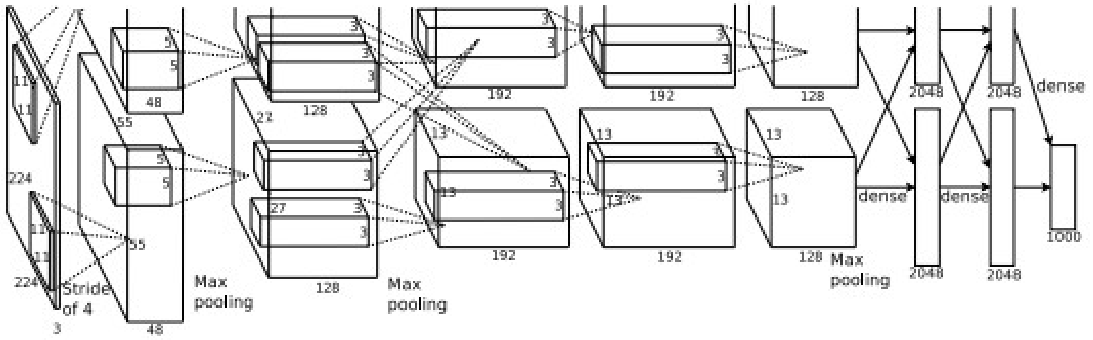
- All we need is gradient of loss function ∇C
  - Rate of change of C wrt. to change in any weigt
  - Rate of change of C wrt. to change in any bias

$$\frac{\partial C}{\partial b_j^l} \qquad \frac{\partial C}{\partial w_{jk}^l}$$



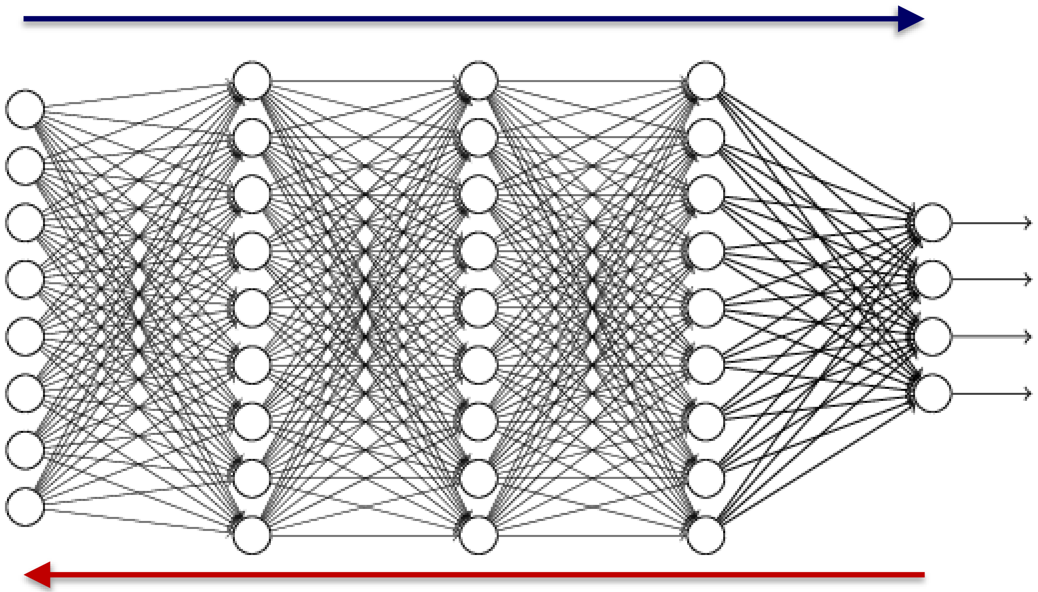
- How to compute gradient?
  - Numericaly
    - Simple, approximate, extremely slow ☹
  - Analyticaly for entire C
    - Fast, exact, nontractable ☹
  - Chain individual parts of network
    - Fast, exact, doable ☺

Backpropagation!



Main principle

- We need the gradient of the Loss function ∇C $\frac{\partial C}{\partial b_j^l}$ $\frac{\partial C}{\partial w_{jk}^l}$
- Two phases:
  - Forward pass; propagation: the input sample is propagated through the network and the error at the final layer is obtained



  - Backward pass; weight update: the error is backpropagated to the individual levels, the contribution of the individual neuron to the error is calculated and the weights are updated accordingly

Learning strategy

- To obtain the gradient of the Loss function ∇C : $\frac{\partial C}{\partial b_j^l}$ $\frac{\partial C}{\partial w_{jk}^l}$
  - For every neuron in the network calculate the error of this neuron

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

    - This error propagates through the network causing the final error
    - Backpropagate the final error to get all δ_j^l
  - Obtain all $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial w_{jk}^l}$ from δ_j^l

Equations of backpropagation

Nielsen, 2015

- BP1: Error in the output layer:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \qquad \delta^L = \nabla_a C \odot \sigma'(z^L)$$

- BP2: Error in terms of the error in the next layer:

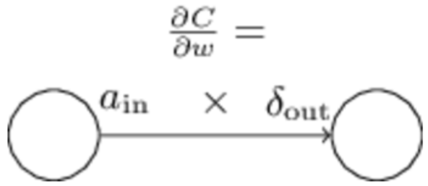
$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \qquad \delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

- BP3: Rate of change of the cost wrt. to any bias:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \frac{\partial C}{\partial b} = \delta$$

- BP4: Rate of change of the cost wrt. to any weight:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \qquad \frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}$$



Backpropagation and SGD

For a number of **epochs**

Until all training images are used

Select a **mini-batch** of m training samples

For each training sample x in the mini-batch

Input: set the corresponding activation $a^{x,1}$

Feedforward: for each $l = 2, 3, \ldots, L$

compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$

Output error: compute $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$

Backpropagation: for each $l = L-1, L-2, \ldots, 2$

compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$

Gradient descend: for each $l = L, L-1, \ldots, 2$ and x update:

$$w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$$

$$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

Example code: Backpropagation

```
def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
```

```
def cost_derivative(self, output_activations, y):
    return (output_activations-y)
```

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

Activation and loss functions

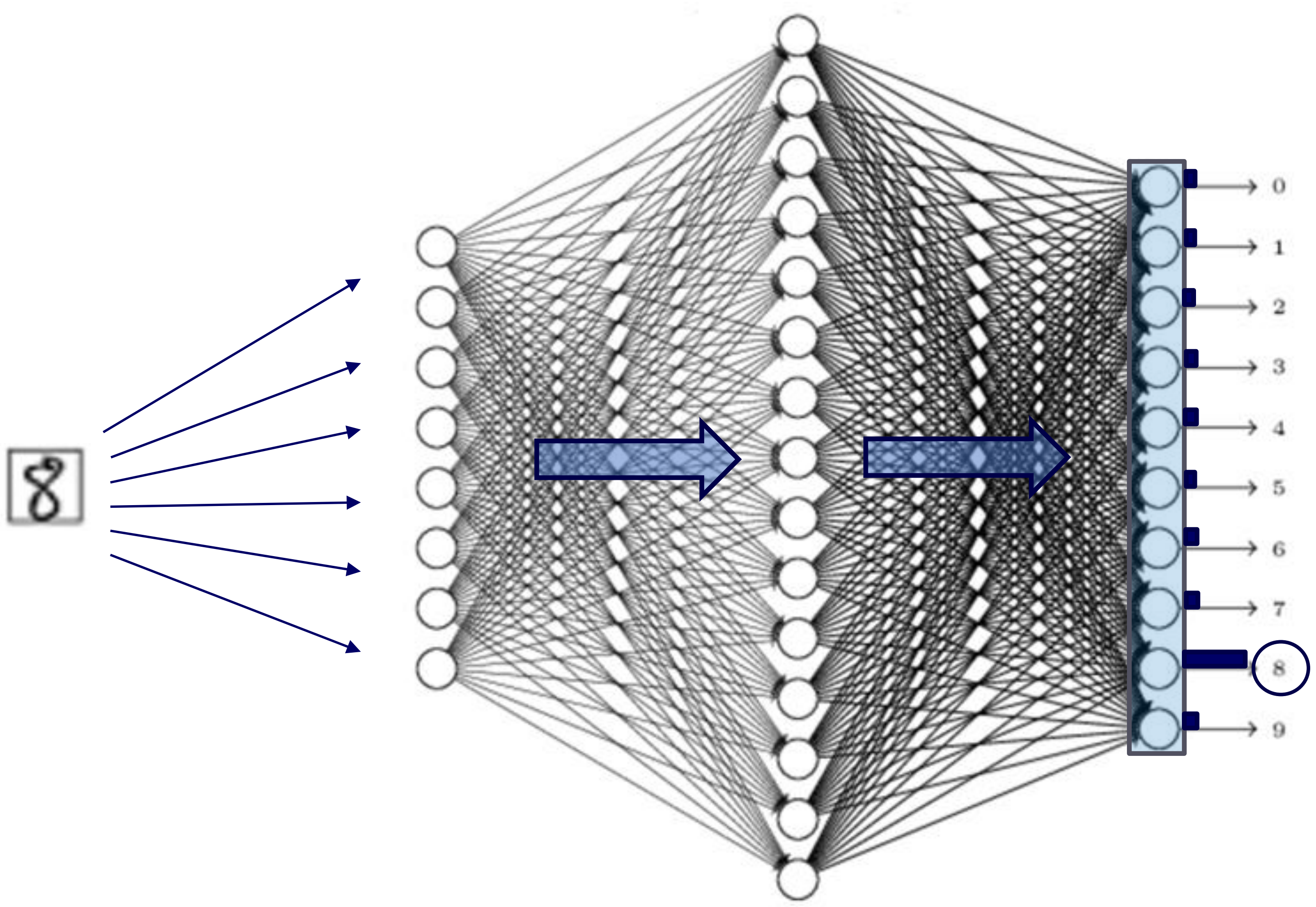
Activation function	Loss function
Linear $$a_j^L = z_j^L$$	Quadratic $$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$
Sigmoid $$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$	Cross-entropy $$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$
Softmax $$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$	Categorical Cross-entropy $$C = -\frac{1}{n} \sum_x \sum_j y_j \ln a_j^L$$
Other	Custom

Activation functions

Method	Papers
ReLU	8096
Sigmoid Activation	5363
GELU Gaussian Error Linear Units (GELUs)	5285
Tanh Activation	4936
Leaky ReLU	915
GLU Language Modeling with Gated Convolutional Networks	372
Swish Searching for Activation Functions	254
Softplus	204
Mish	183
SELU Self-Normalizing Neural Networks	178
PReLU Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification	86
ReLU6 MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications	58
Hard Swish Searching for MobileNetV3	54
Maxout Maxout Networks	45
ELU Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)	34

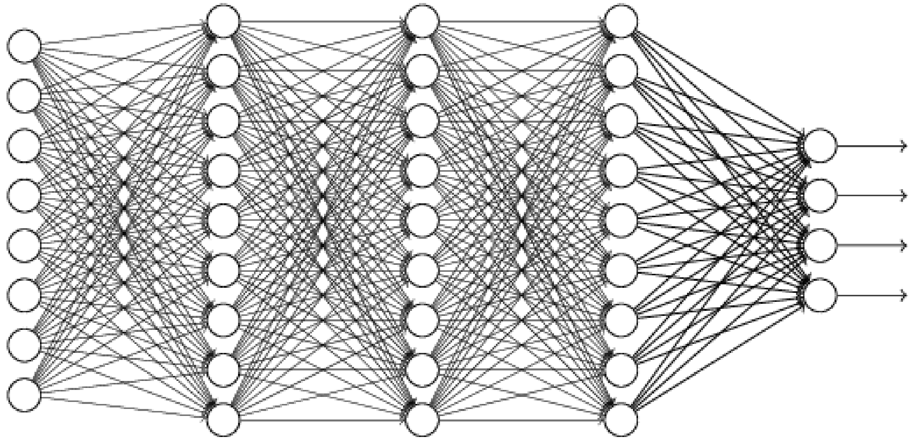
[https://paperswithcode.com]

Classification with Feedforward neural networks

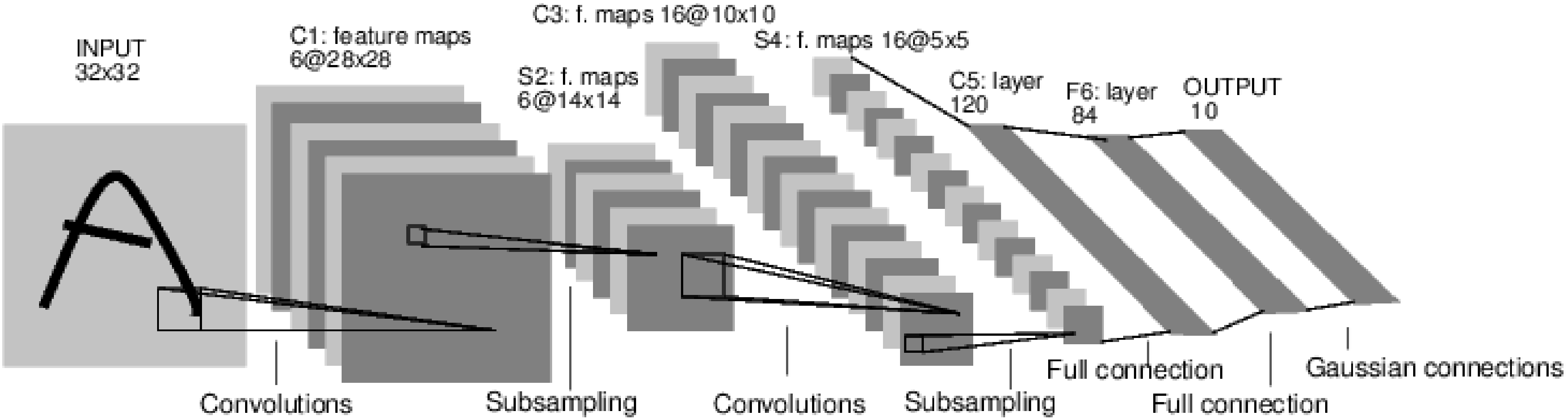


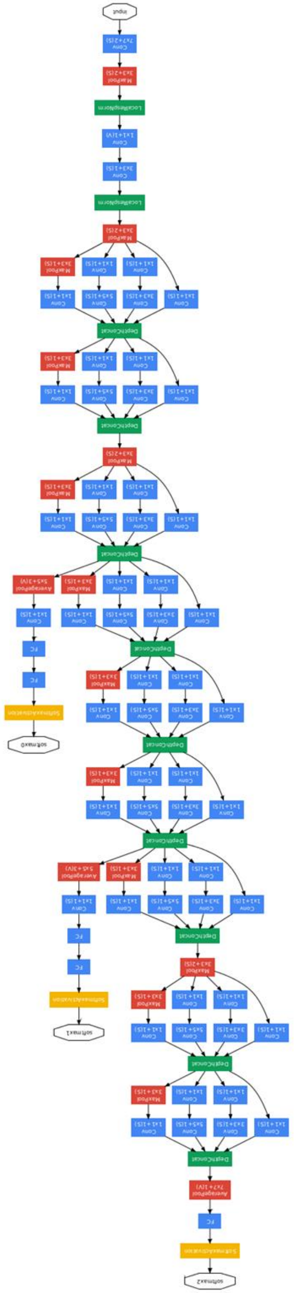
Convolutional neural networks

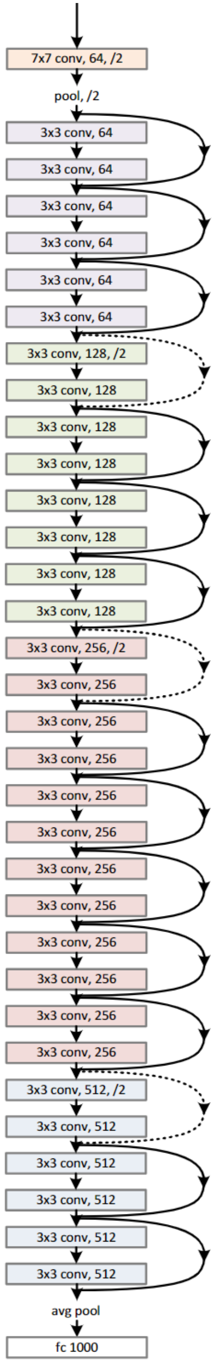
- From feedforward fully-connected neural networks …



- … to convolutional neural networks

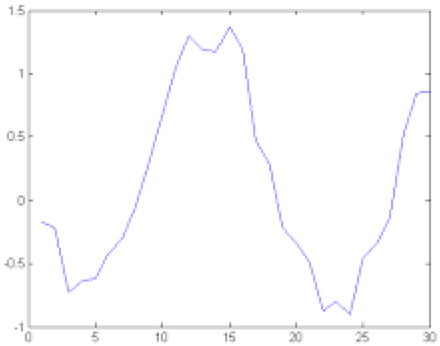


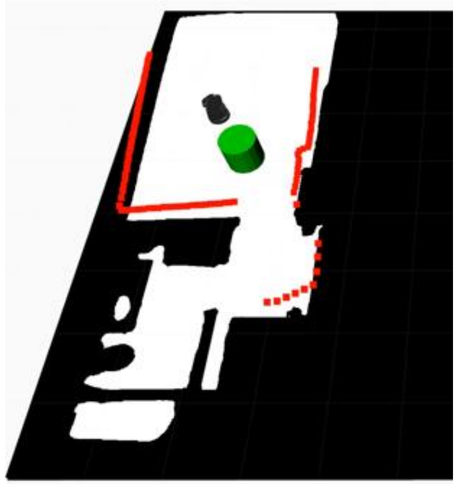


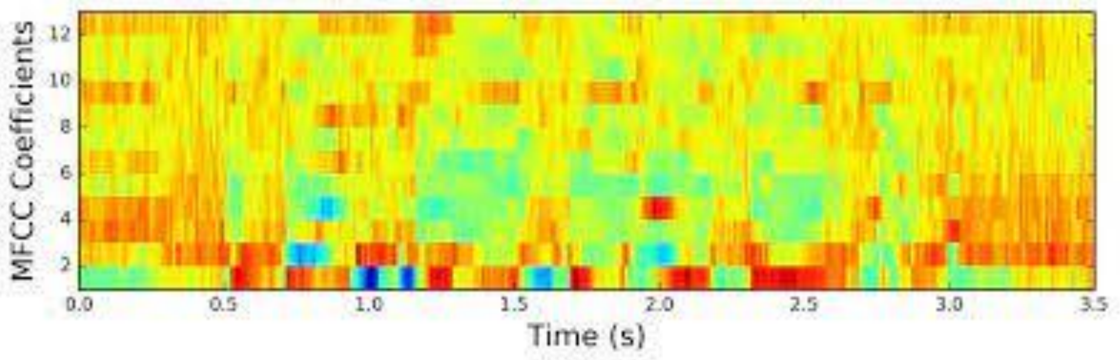


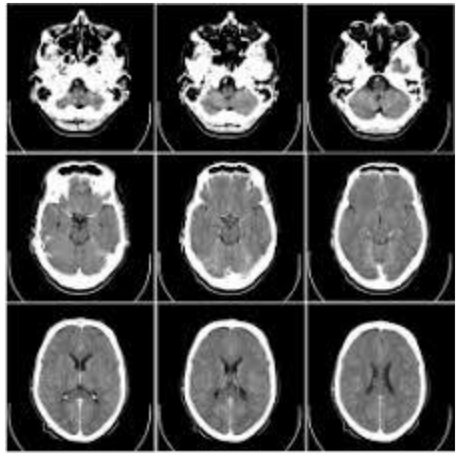
Convolutional neural networks

- Data in vectors, matrices, tensors
- Neigbourhood, spatial arrangement
- 2D: Images,time-fequency representations



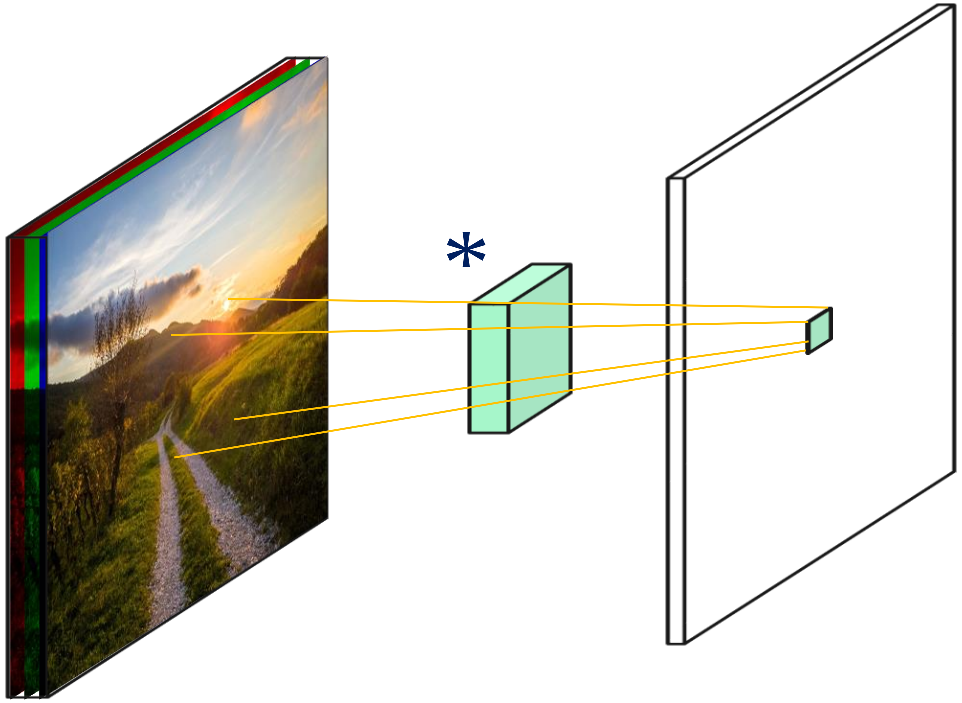




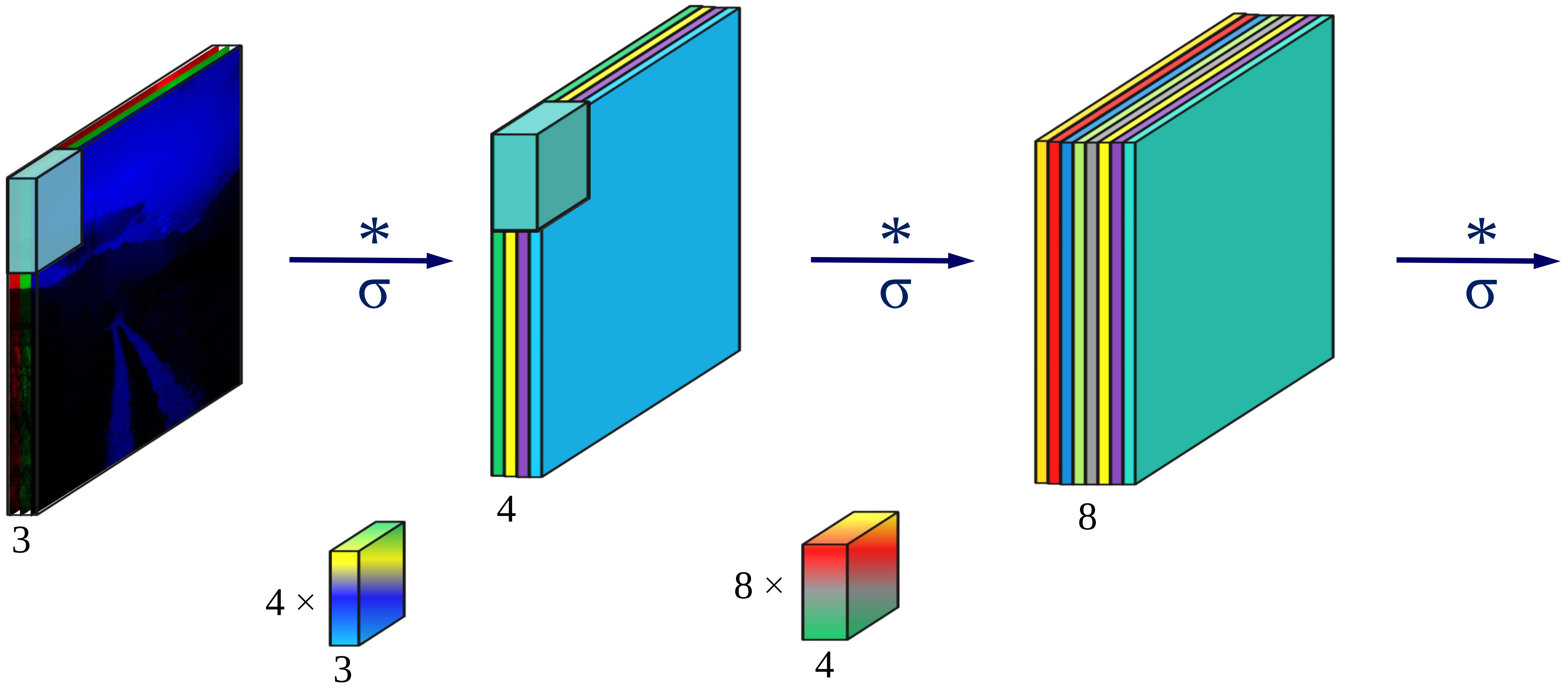


- 1D: sequential signals, text, audio, speech, time series,…
- 3D: volumetric images, video, 3D grids

Convolution layer

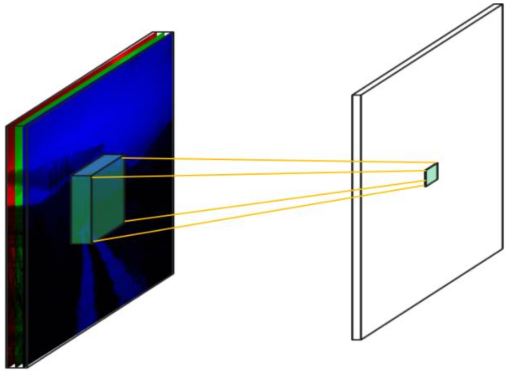


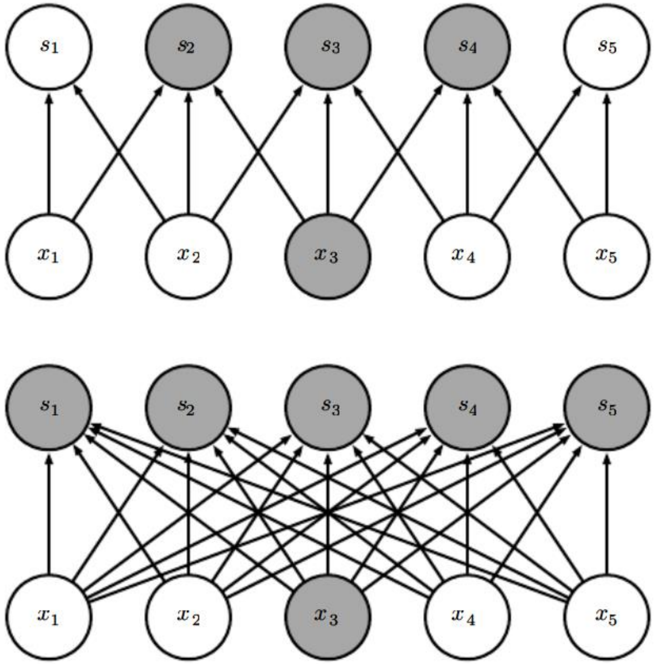
Convolution layer



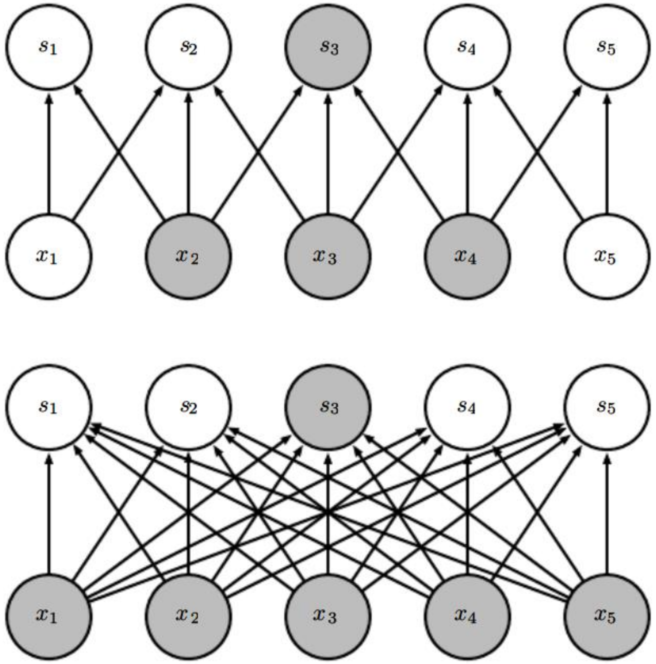
Sparse connectivity

- Local connectivity – neurons are only locally connected (**receptive field**)
  - Reduces memory requirements
  - Improves statistical efficiency
  - Requires fewer operations

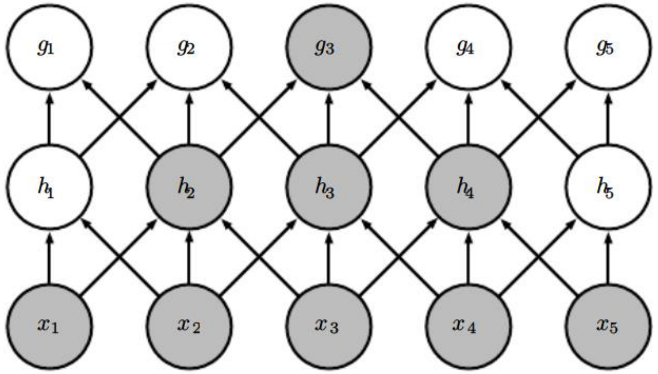




from below



from above

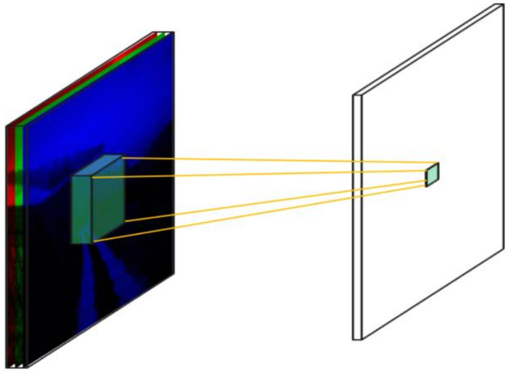


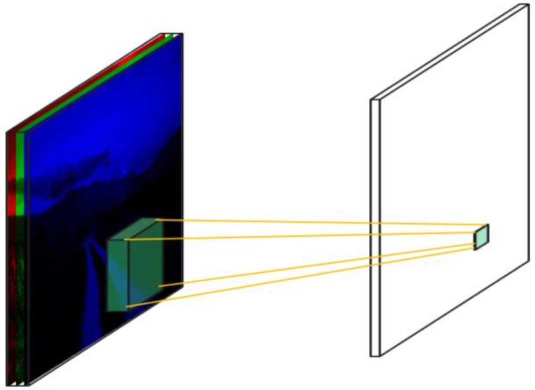
The receptive field of the units in the deeper layers is large
=> Indirect connections!

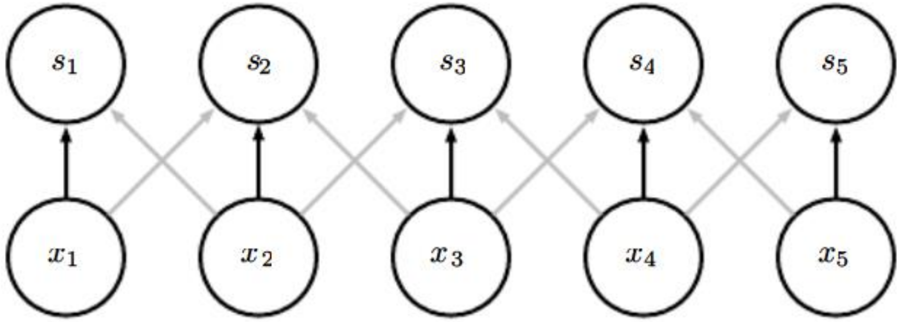
Parameter sharing

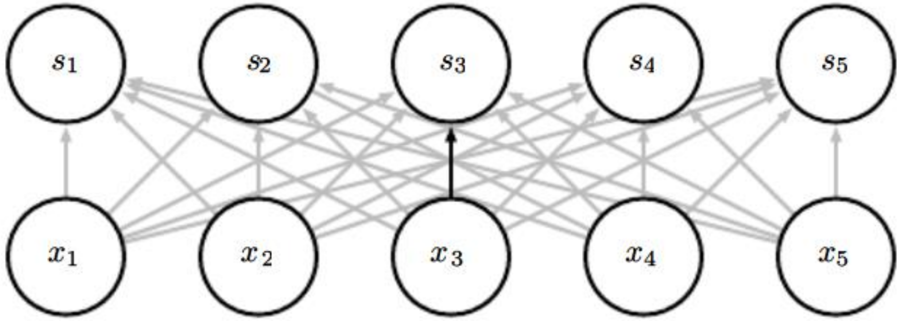
- **Neurons share weights!**
  - Tied weights
- Every element of the kernel is used at every position of the input
- All the neurons at the same level detect the same feature (everywhere in the input)
- Greatly reduces the number of parameters!

- **Equivariance to translation**
  - Shift, convolution = convolution, shift
  - Object moves => representation moves



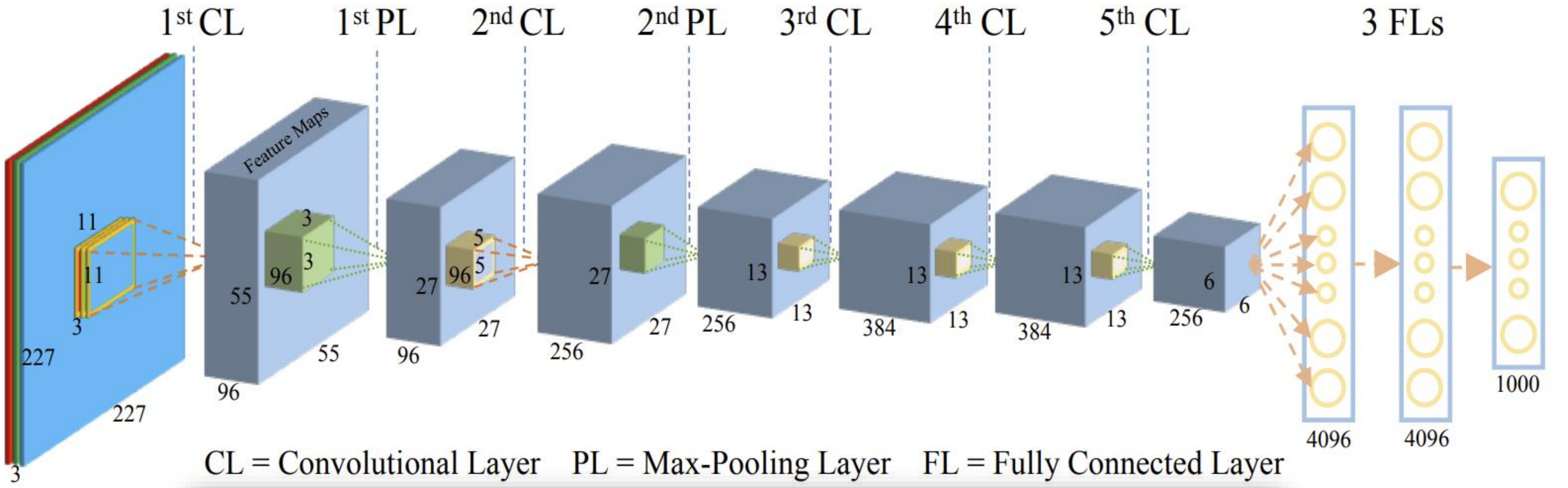




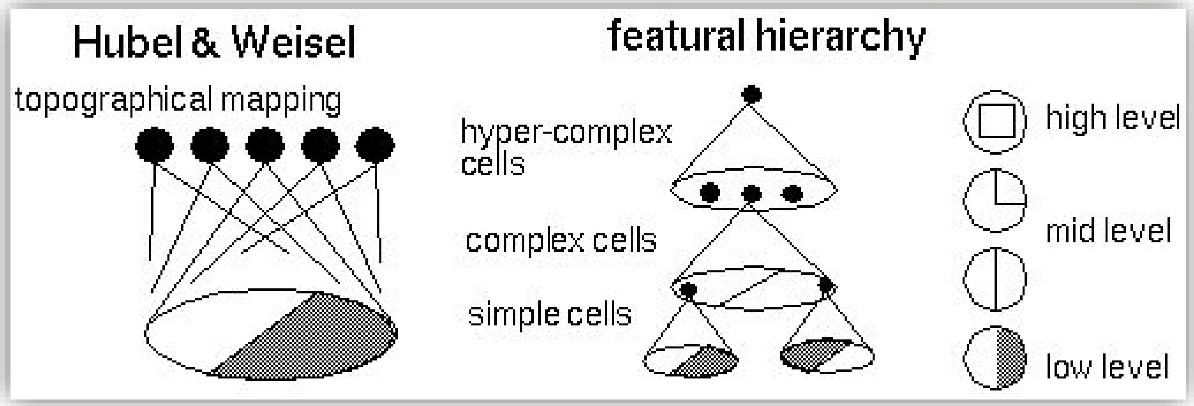


Convolutional neural network

- Hierarchical representation
- Increasingly larger effective receptive field



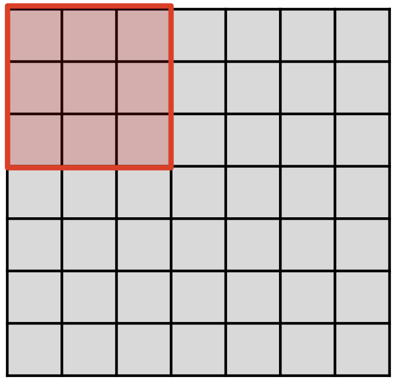
[Qin et al., 2018]



[Hubel & Wiesel, 1961]

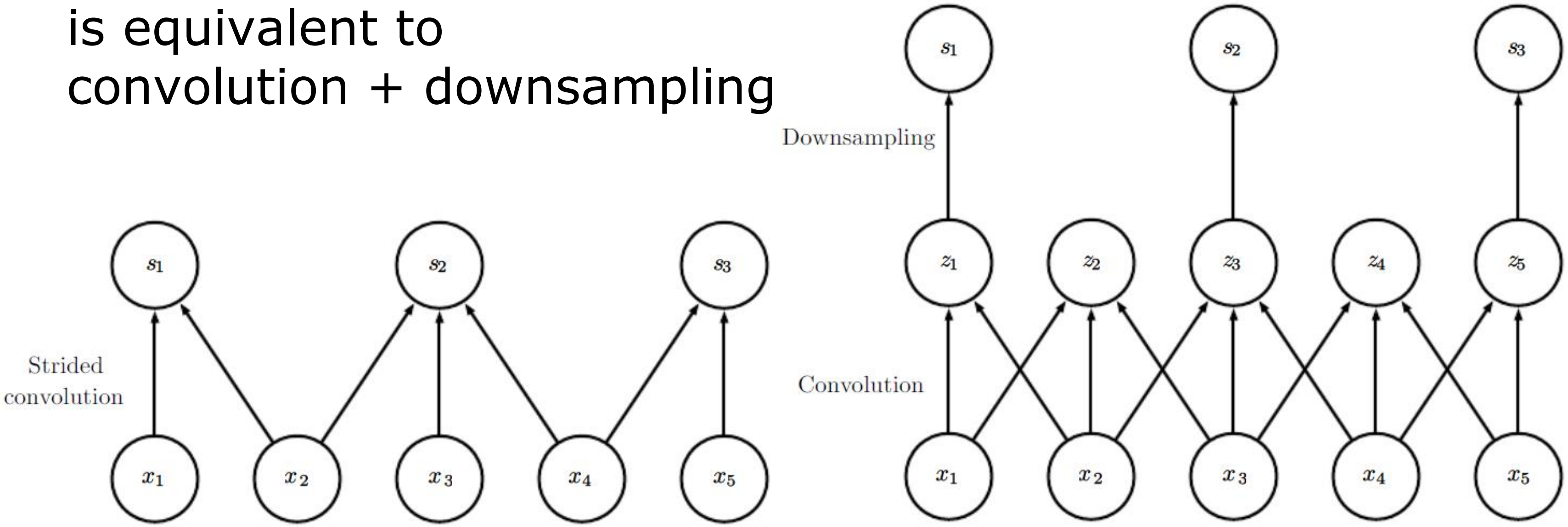
Stride

- Step for convolution filter



Stride=1
Stride=2

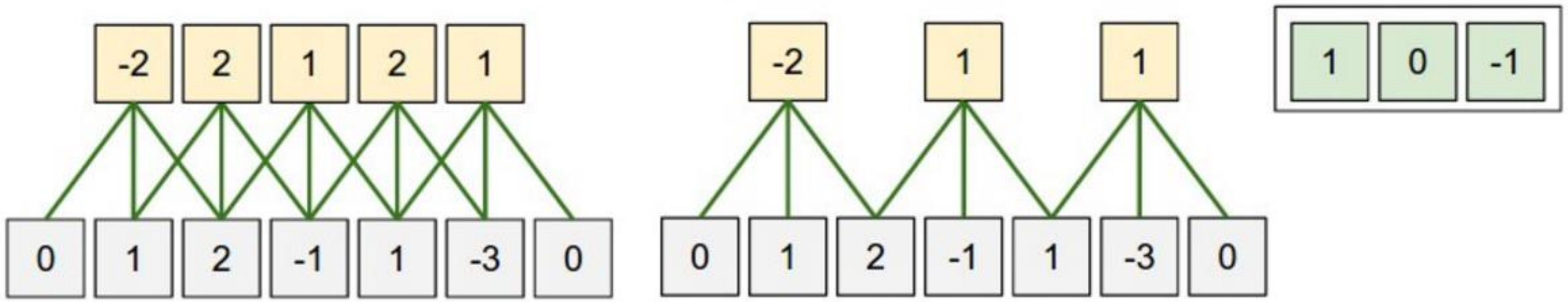
Convolution with stride>1
is equivalent to
convolution + downsampling



- Output size:

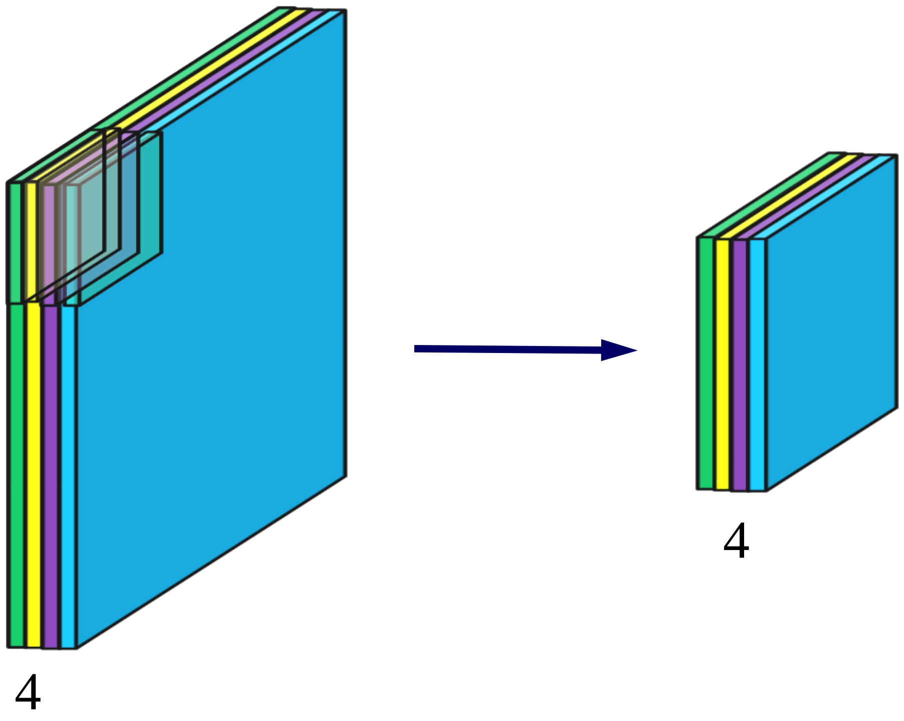
$$\frac{N-F}{S}+1$$

- Example:

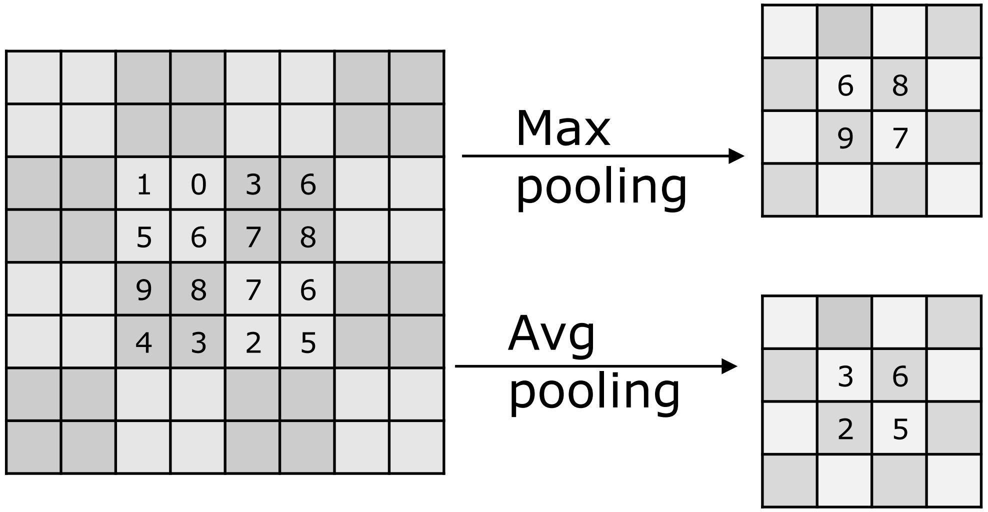


Pooling layer

- Downsampling – reduces the volume size (width and height)
- Process each activation map independently – keeps the volume depth unchanged



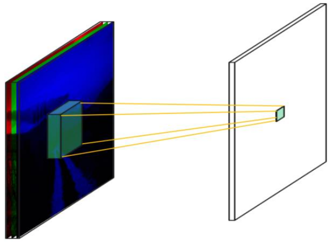
- Example with
  - F=2
  - S=2



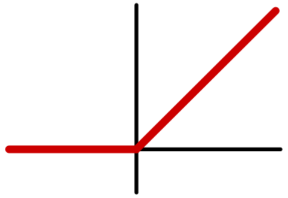
CNN layers

- Layers used to build ConvNets:
  - INPUT:
    raw pixel values

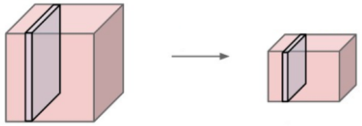
  - CONV:
    convolutional layer



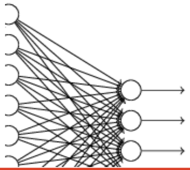
  - (BN: batch normalisation)
  - (ReLU:)
    introducing nonlinearity

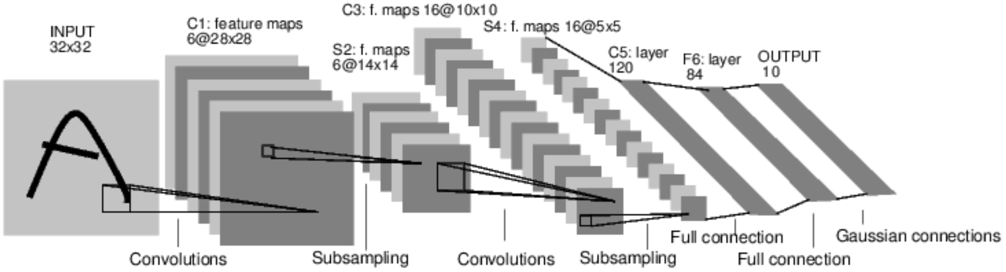


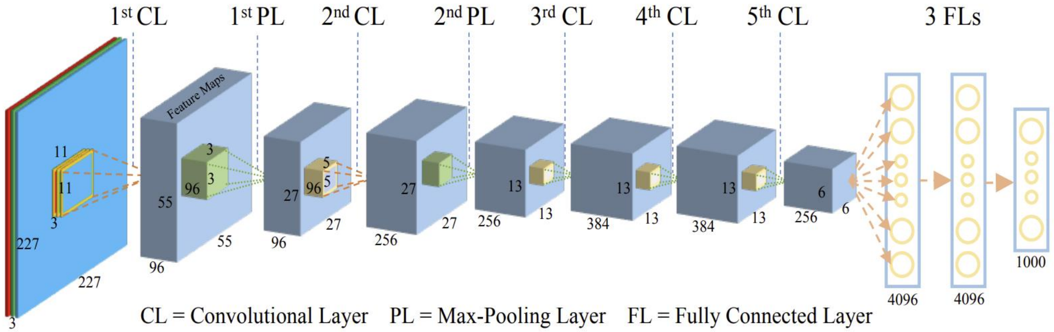
  - POOL:
    downsampling

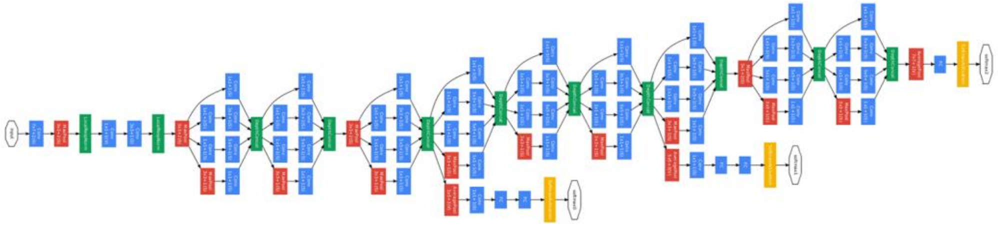


  - FC:
    for computing class scores
  - SoftMax







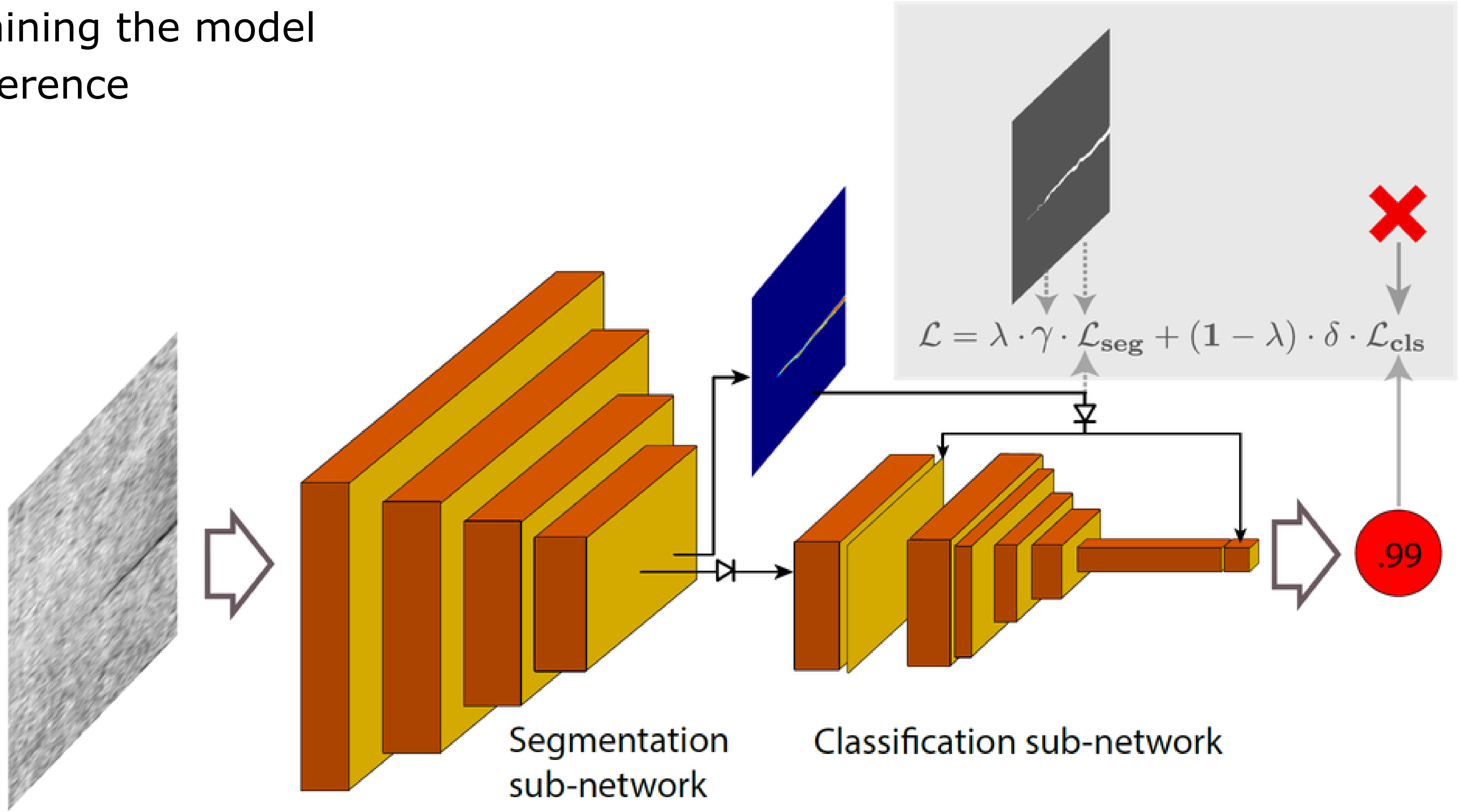


Typical solution

Korak 1: Zajem podatkov

Network architecture

- Training the model
- Inference



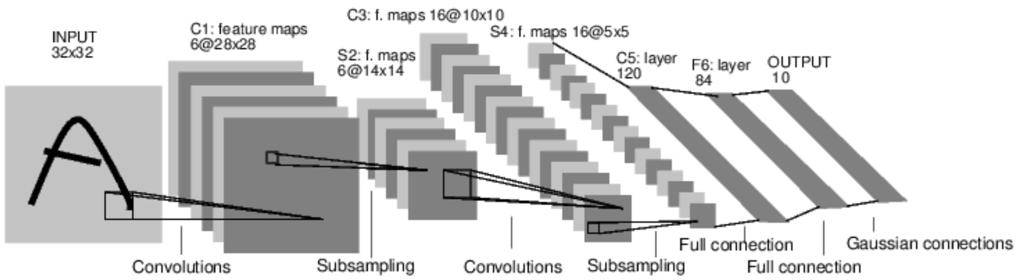
Example implementation in TensorFlow

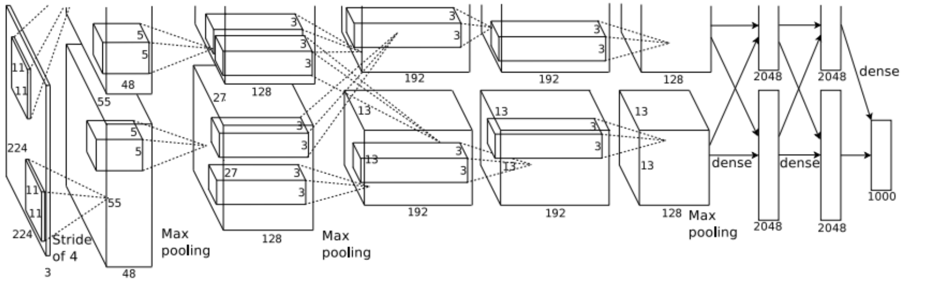
```python
with variable_scope.variable_scope(scope, 'SegDecNet', [inputs]) as sc:
    end_points_collection = sc.original_name_scope + '_end_points'
    # Collect outputs for conv2d, max_pool2d
    with arg_scope([layers.conv2d, layers.fully_connected, layers_lib.max_pool2d, layers.batch_norm],
                   outputs_collections=end_points_collection):
        # Apply specific parameters to all conv2d layers (to use batch norm and relu - relu is by default)
        with arg_scope([layers.conv2d, layers.fully_connected],
                       weights_initializer= lambda shape,dtype=tf.float32, partition_info=None: tf.random_normal(shape, mean=0,stddev=0.01, dtype=dtype),
                       biases_initializer=None,
                       normalizer_fn=layers.batch_norm,
                       normalizer_params={'center': True,'scale': True, 'decay': self.BATCHNORM_MOVING_AVERAGE_DECAY,'epsilon': 0.001}):

            net = layers_lib.repeat(inputs, 2, layers.conv2d, 32, [5, 5], scope='conv1')
            net = layers_lib.max_pool2d(net, [2, 2], scope='pool1')
            net = layers_lib.repeat(net, 3, layers.conv2d, 64, [5, 5], scope='conv2')
            net = layers_lib.max_pool2d(net, [2, 2], scope='pool2')
            net = layers_lib.repeat(net, 4, layers.conv2d, 64, [5, 5], scope='conv3')
            net = layers_lib.max_pool2d(net, [2, 2], scope='pool3')
            net = layers.conv2d(net, 1024, [15, 15], padding='SAME', scope='conv4')
            net_prob_mat = layers.conv2d(net, 1, [1, 1], scope='conv5', activation_fn=None)

            with tf.name_scope('decision'):
                net_prob_mat = tf.nn.relu(net_prob_mat)
                decision_net = tf.concat([net, net_prob_mat],axis=3)
                decision_net = layers_lib.max_pool2d(decision_net, [2, 2], scope='decision/pool4')
                decision_net = layers.conv2d(decision_net, 8, [5, 5], padding='SAME', scope='decision/conv6')
                decision_net = layers_lib.max_pool2d(decision_net, [2, 2], scope='decision/pool5')
                decision_net = layers.conv2d(decision_net, 16, [5, 5], padding='SAME', scope='decision/conv7')
                decision_net = layers_lib.max_pool2d(decision_net, [2, 2], scope='decision/pool6')
                decision_net = layers.conv2d(decision_net, 32, [5, 5], scope='decision/conv8')

                with tf.name_scope('decision/global_avg_pool'):
                    avg_decision_net = keras.layers.GlobalAveragePooling2D()(decision_net)
                with tf.name_scope('decision/global_max_pool'):
                    max_decision_net = keras.layers.GlobalMaxPooling2D()(decision_net)
                with tf.name_scope('decision/global_avg_pool'):
                    avg_prob_net = keras.layers.GlobalAveragePooling2D()(net_prob_mat)
                with tf.name_scope('decision/global_max_pool'):
                    max_prob_net = keras.layers.GlobalMaxPooling2D()(net_prob_mat)

                # adding avg_prob_net and max_prob_net may not be needed, but it doesen't hurt
                decision_net = tf.concat([avg_decision_net, max_decision_net, avg_prob_net, max_prob_net], axis=1)
                decision_net = layers.fully_connected(decision_net, 1, scope='decision/FC9',normalizer_fn=None,
                                                      biases_initializer=tf.constant_initializer(0), activation_fn=None)
return decision_net
```

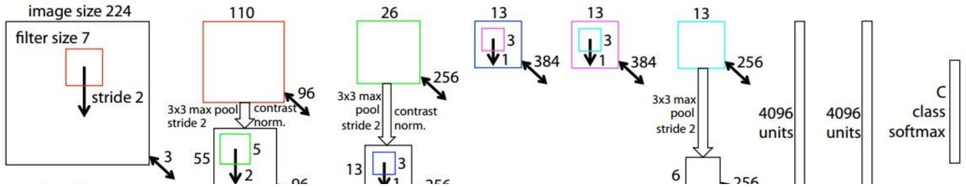
Segmentation network

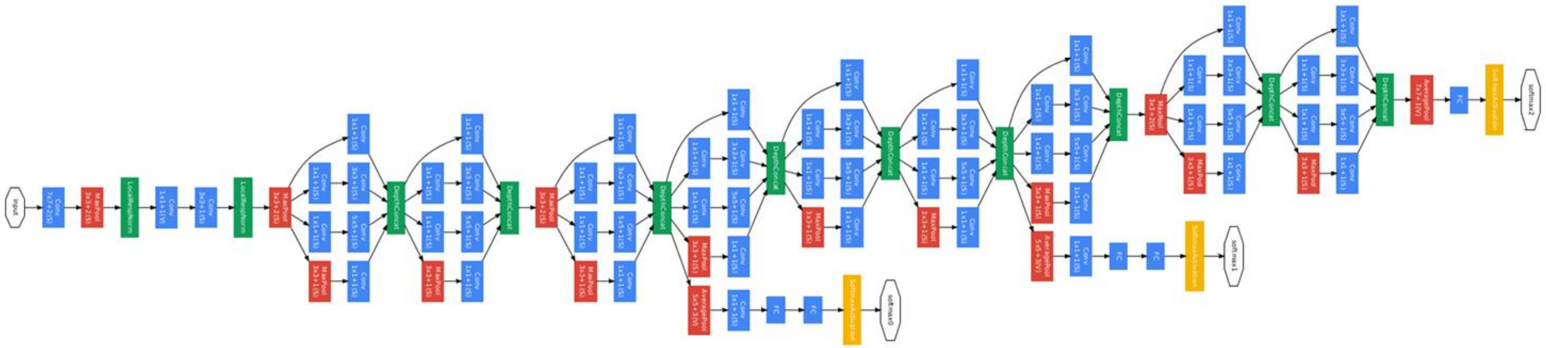
Classification network

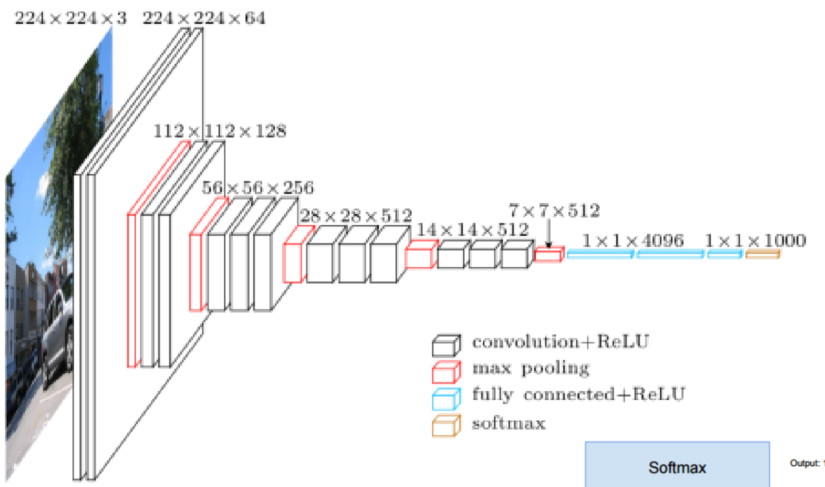
Backbone architectures

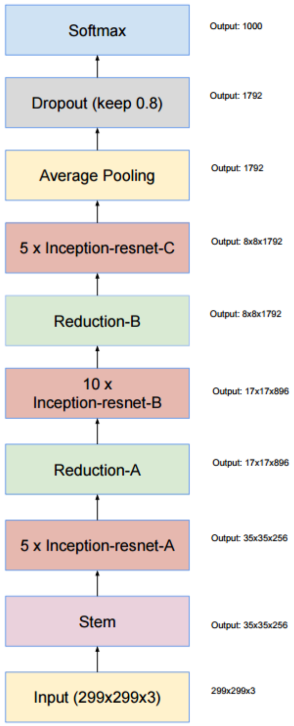


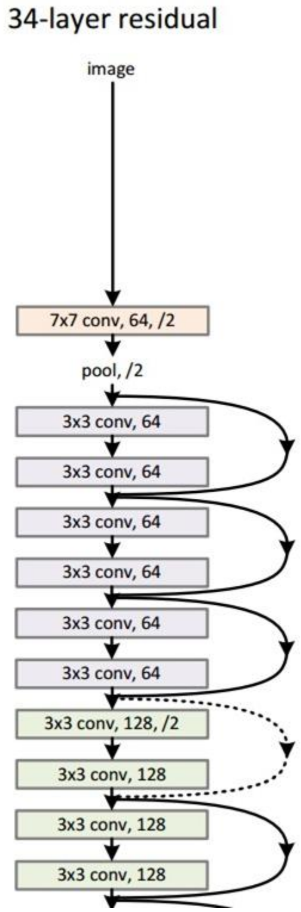








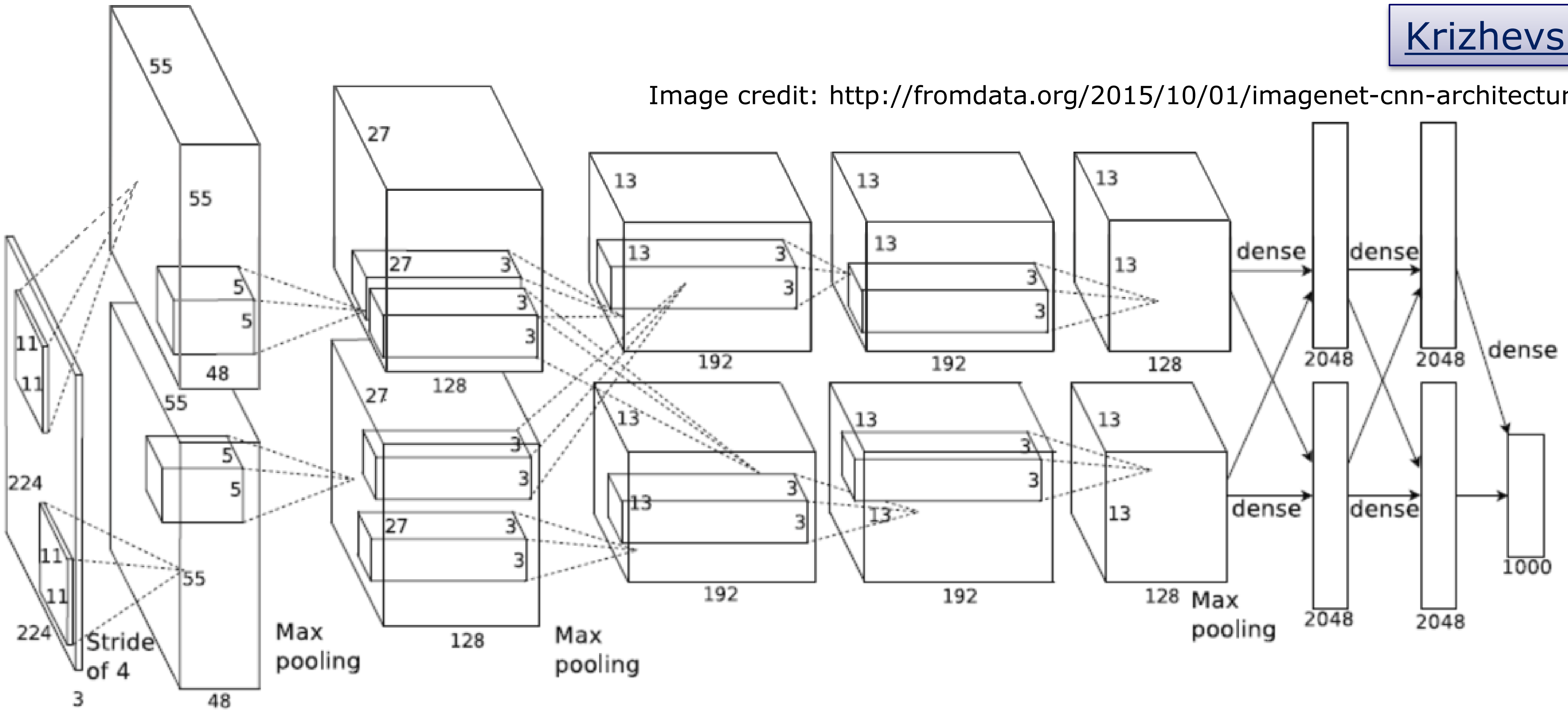




AlexNet

[Krizhevsky, 2012]

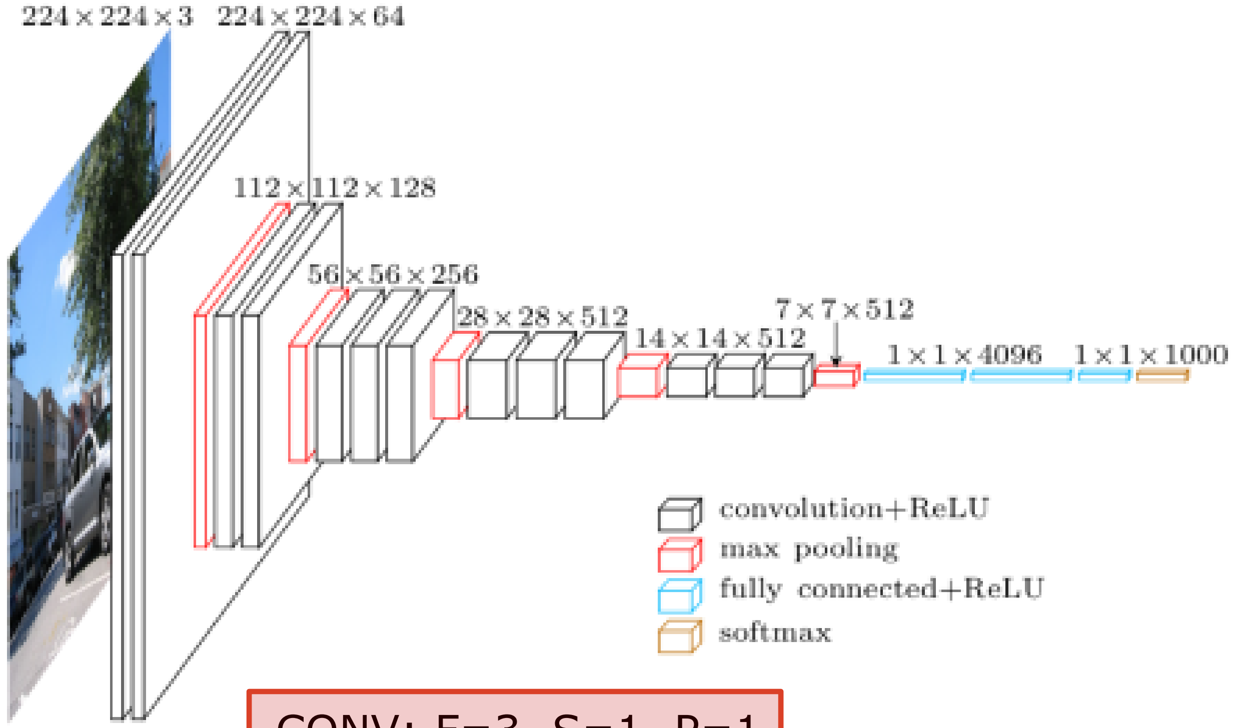
Image credit: http://fromdata.org/2015/10/01/imagenet-cnn-architecture-image/



CONV1	POOL	CONV2	POOL	CONV3	CONV4	CONV5	POOL	FC6	FC7	FC8
F=11	F=3	F=5	F=3	F=3	F=3	F=3	F=3	4096	4096	1000
S=4	S=2	S=1	S=2	S=1	S=1	S=1	S=2			
		P=2		P=1	P=1	P=1				

- ReLU, data augmentation, Dropout, Momentum, L2 regularisation

VGG



CONV: F=3, S=1, P=1
POOL: F=2, S=2

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

- Classical CNN backbone shape
- VGG16, VGG19

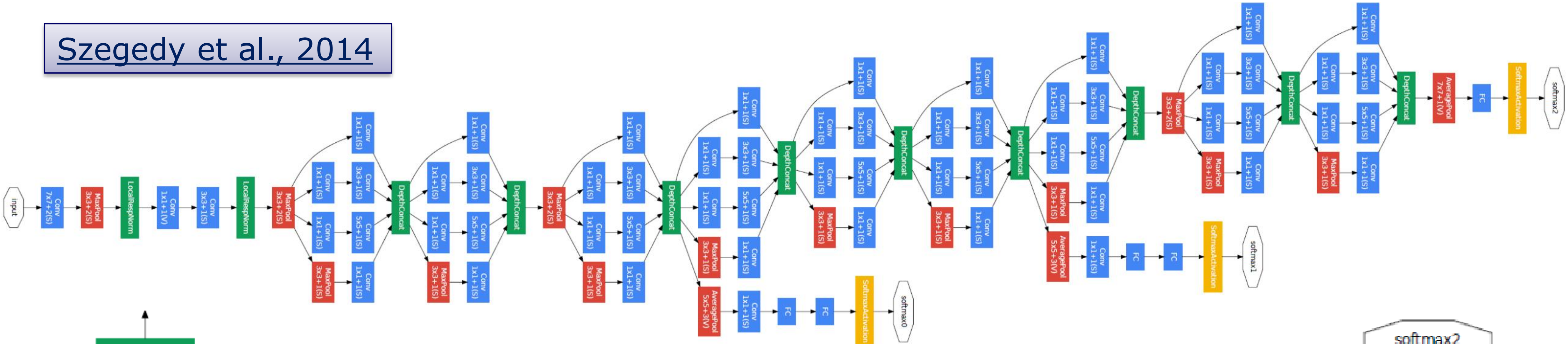
[Simonyan & Zisserman, 2014]

Table 2: **Number of parameters** (in millions).

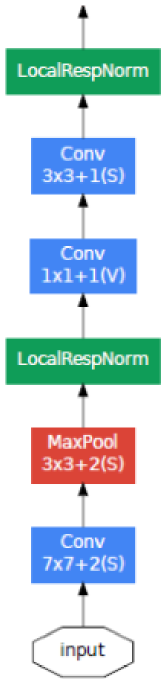
Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

GoogLeNet / Inception

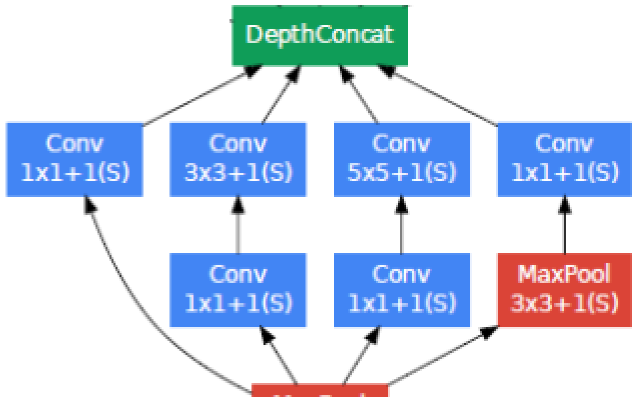
Szegedy et al., 2014



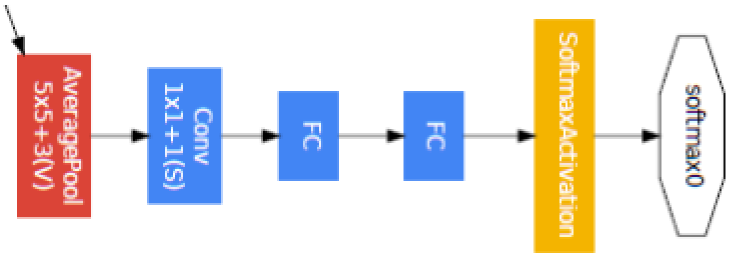
Stem network

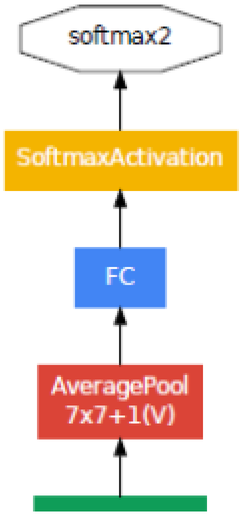


Inception module



Auxiliary output





Classifier output

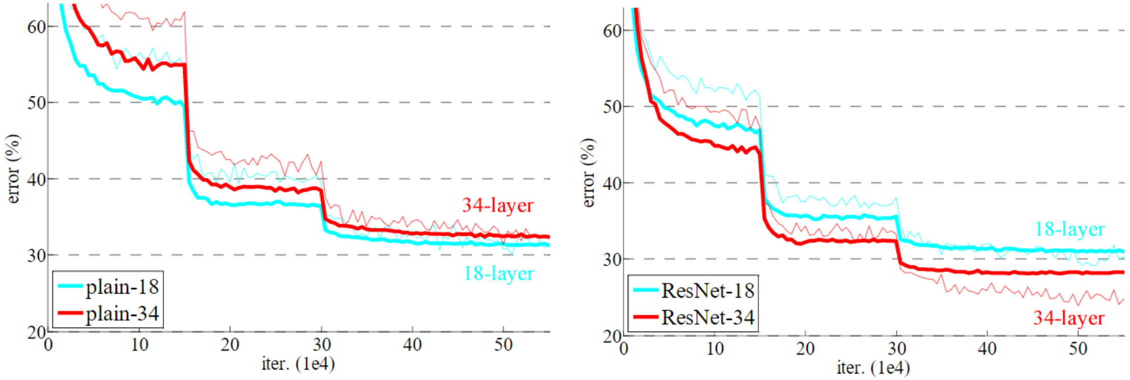
ResNet

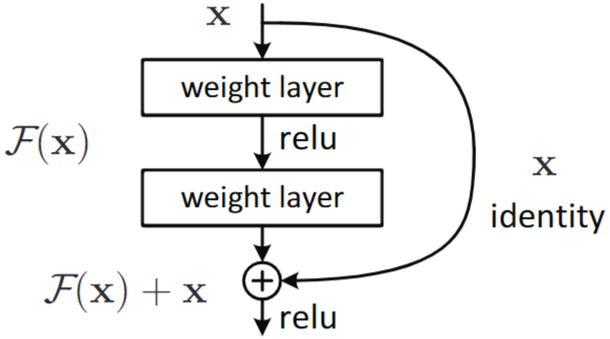
- Going deeper!
- Plain deep networks do not work

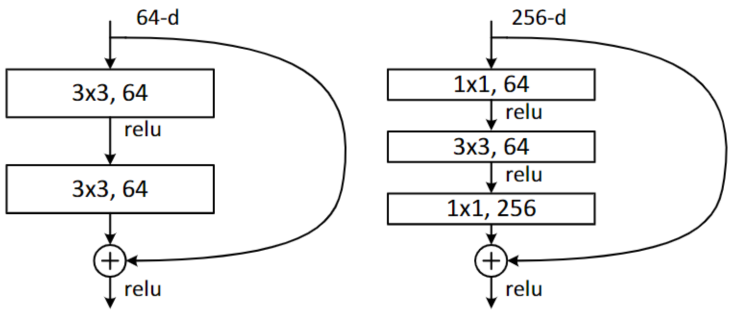
- Shortcut connections!
  - Figth vanishing gradient problem
- Learn residual functions

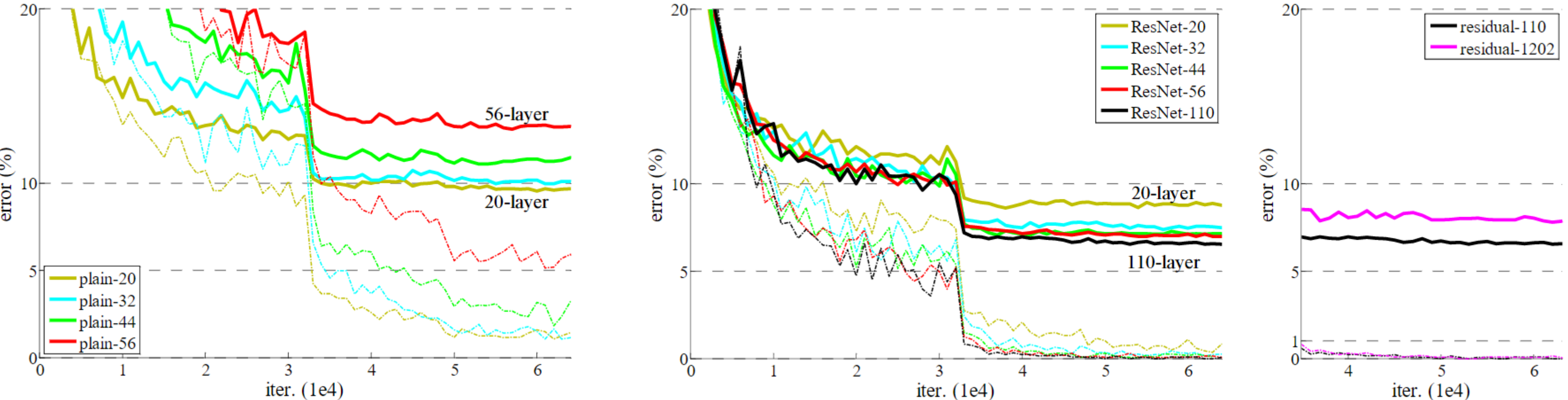
$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}$$

- Bottleneck building blocks
- Very deep networks:
  - 152, 101, 50, 34, 18



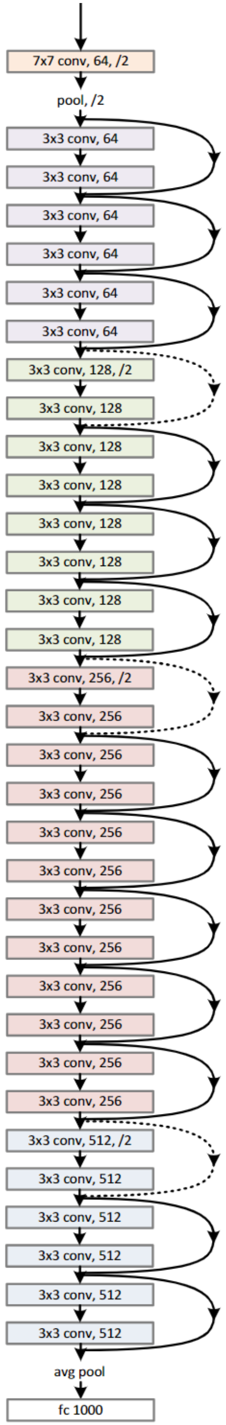






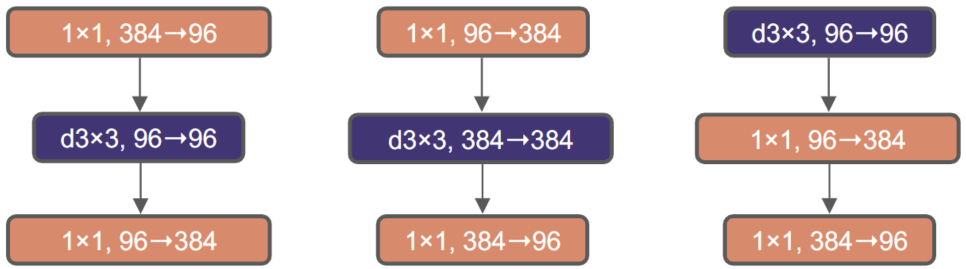
	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

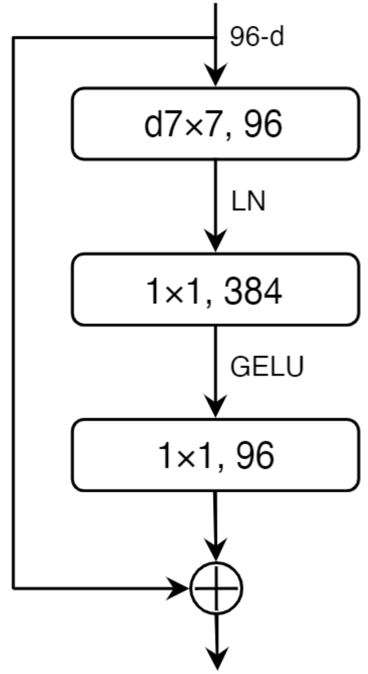
He et al., 2015

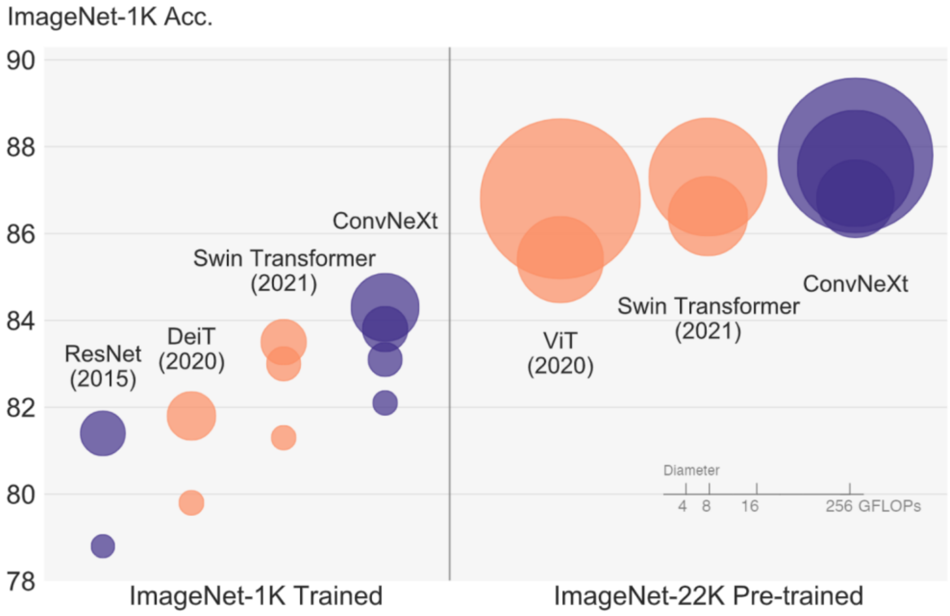


ConvNext

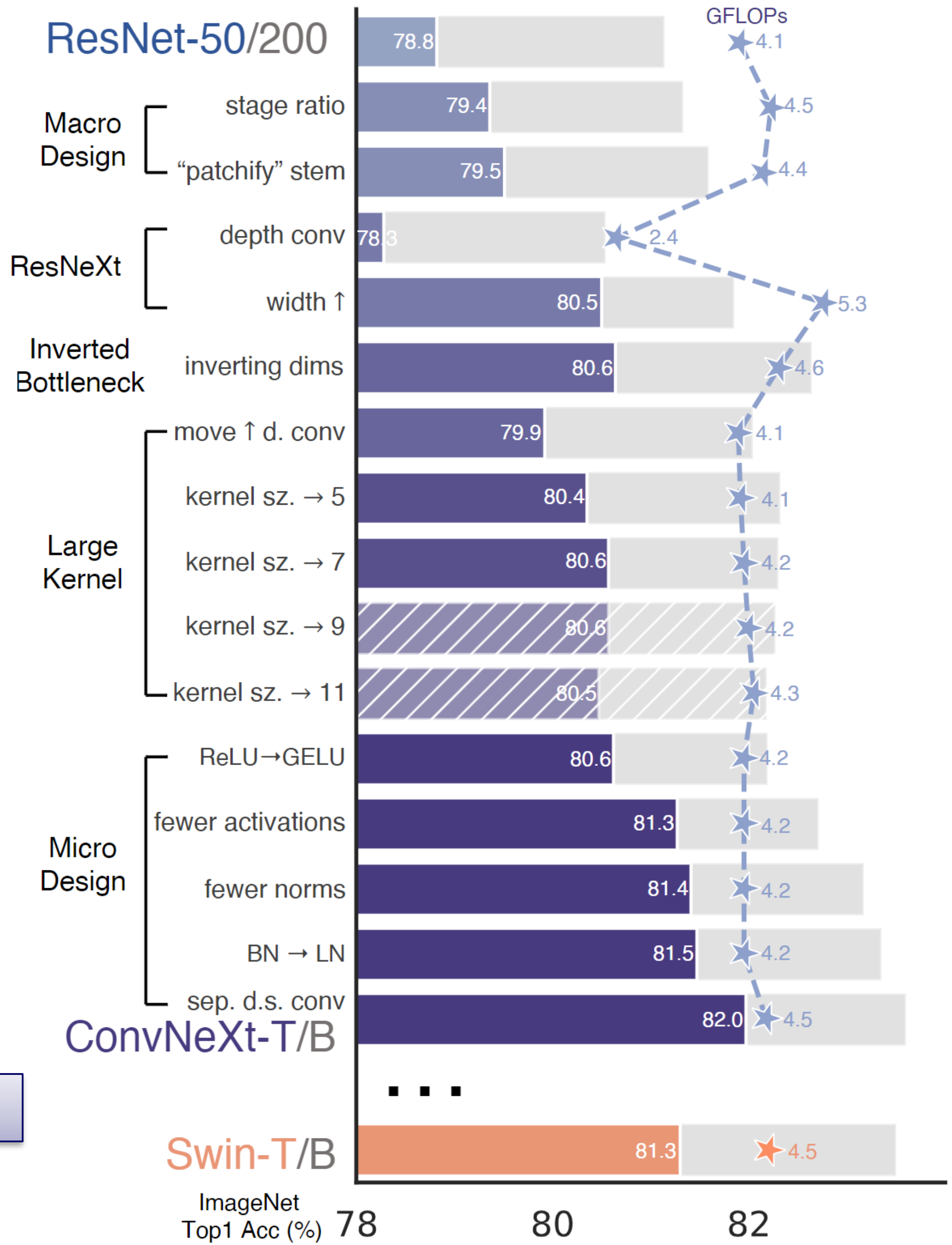
- A ConvNet for the 2020s
- Transformer-inspired modifications of ResNet







Liu et al. 2022



Architectures overview

- paperswithcode.com *[paperswithcode.com, 2022]*
  - Top 20 methods in Convolutional Neural Networks

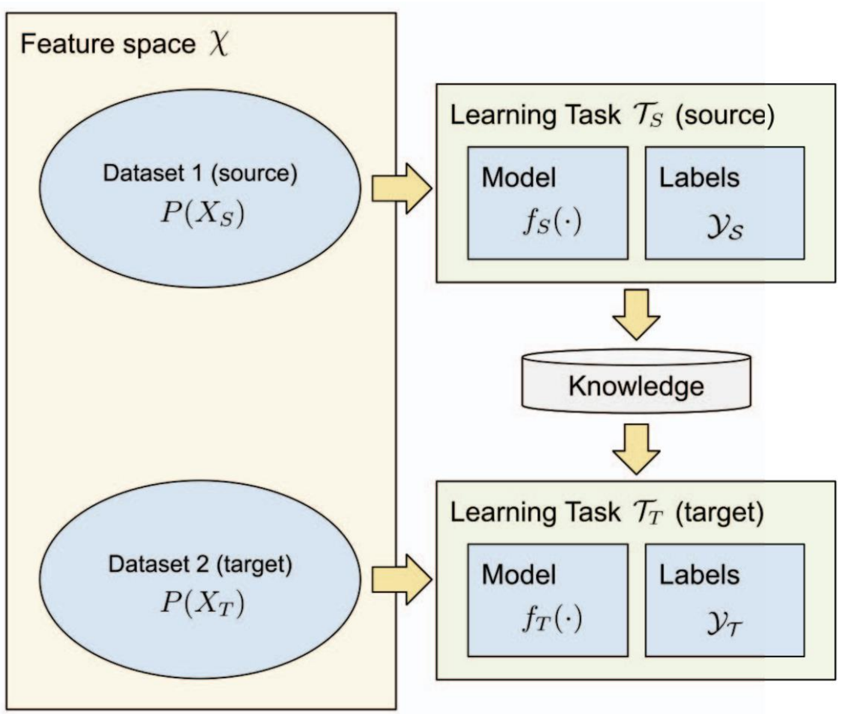
Method	Year	Papers
ResNet Deep Residual Learning for Image Recognition	2015	1461
VGG Very Deep Convolutional Networks for Large-Scale Image Recognition	2014	369
DenseNet Densely Connected Convolutional Networks	2016	300
AlexNet ImageNet Classification with Deep Convolutional Neural Networks	2012	280
VGG-16 Very Deep Convolutional Networks for Large-Scale Image Recognition	2014	258
MobileNetV2 MobileNetV2: Inverted Residuals and Linear Bottlenecks	2018	201
EfficientNet EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks	2019	154
Darknet-53 YOLOv3: An Incremental Improvement	2018	142
ResNeXt Aggregated Residual Transformations for Deep Neural Networks	2016	120
GoogLeNet Going Deeper with Convolutions	2014	119
Xception Xception: Deep Learning With Depthwise Separable Convolutions	2017	94
SqueezeNet SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size	2016	71
Inception-v3 Rethinking the Inception Architecture for Computer Vision	2015	67
CSPDarknet53 YOLOv4: Optimal Speed and Accuracy of Object Detection	2020	46
MobileNetV1 MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications	2017	44
LeNet	1998	44
Darknet-19 YOLO9000: Better, Faster, Stronger	2016	44
WideResNet Wide Residual Networks	2016	42
ShuffleNet ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices	2017	36
MobileNetV3 Searching for MobileNetV3	2019	34

Pretrained models

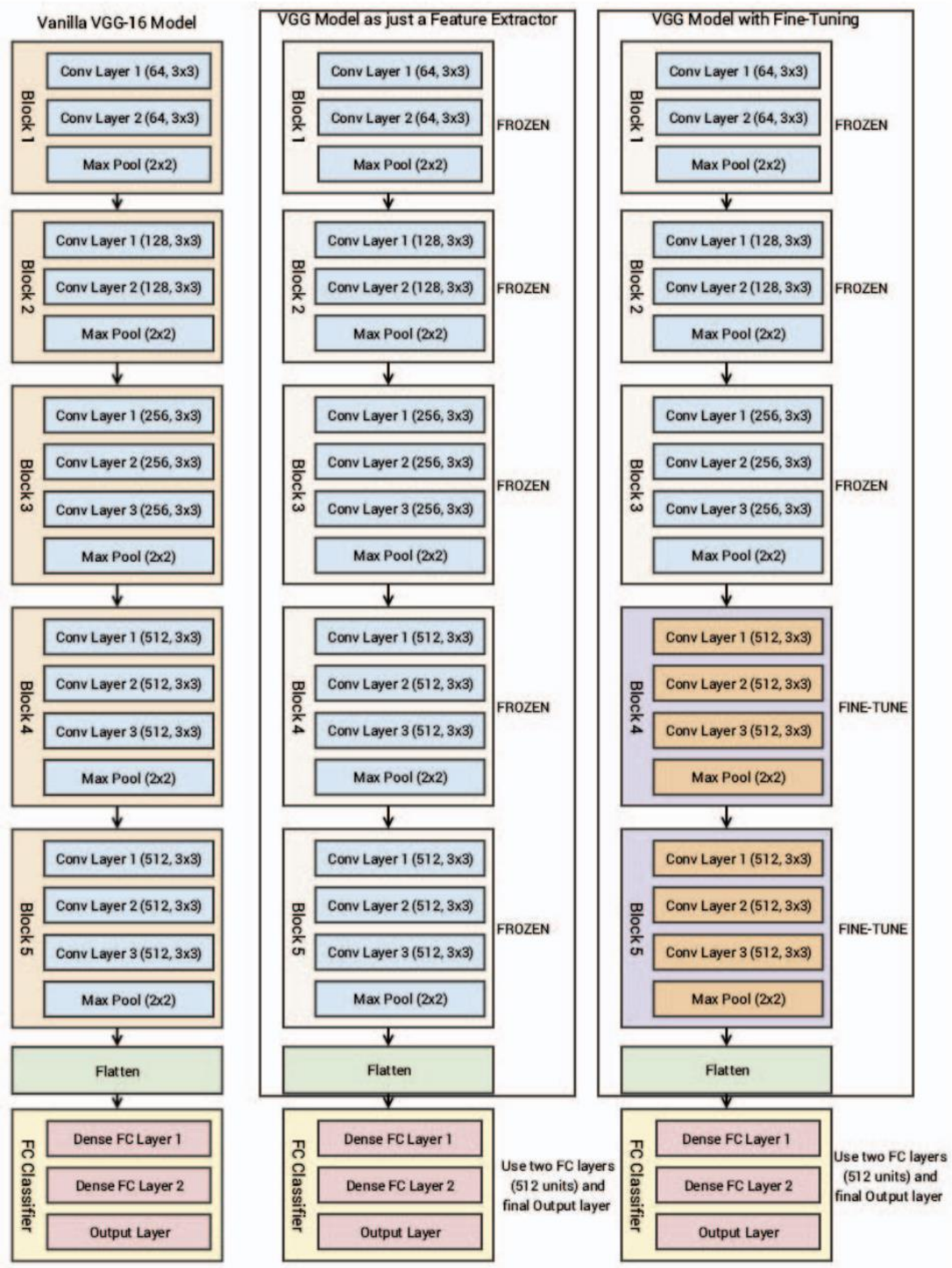
```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
squeezenet = models.squeezenet1_0(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
densenet = models.densenet161(pretrained=True)
inception = models.inception_v3(pretrained=True)
googlenet = models.googlenet(pretrained=True)
shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
mobilenet_v2 = models.mobilenet_v2(pretrained=True)
mobilenet_v3_large = models.mobilenet_v3_large(pretrained=True)
mobilenet_v3_small = models.mobilenet_v3_small(pretrained=True)
resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
wide_resnet50_2 = models.wide_resnet50_2(pretrained=True)
mnasnet = models.mnasnet1_0(pretrained=True)
```

Transfer learning

- Train on a large related dataset
- Fine-tune on the target dataset
- Heavily used



[Ribani & Marengoni 2019]

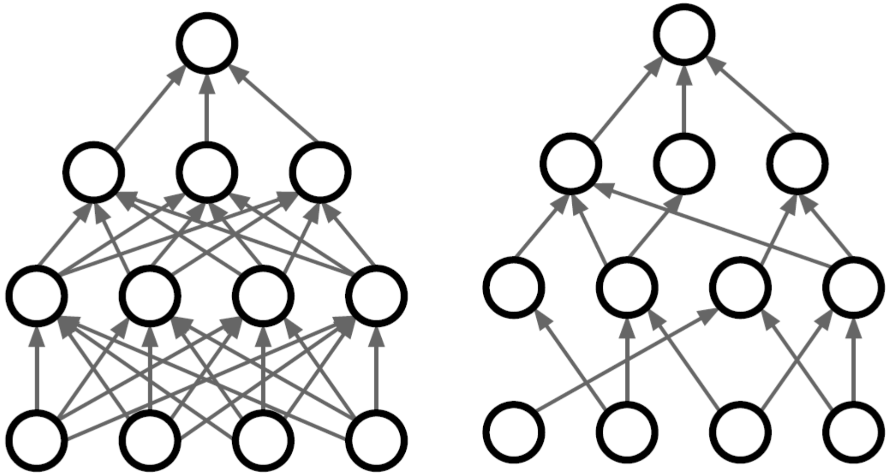


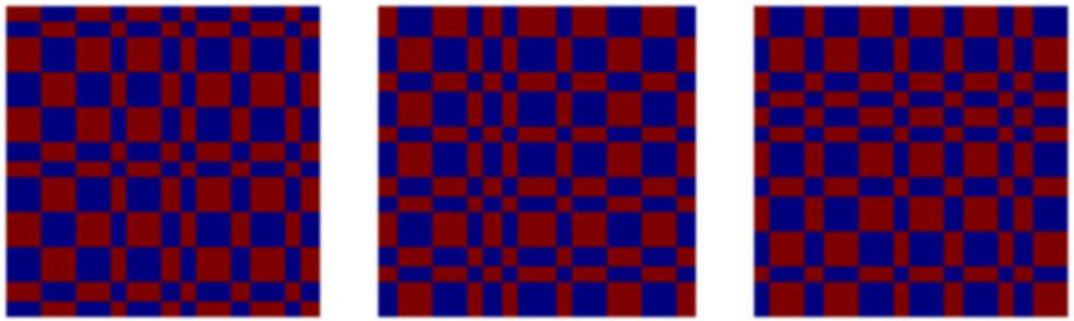
Regularisation

- How to avoid overfitting:
  - Increase the number of training images ☹
  - Decrease the number of parameters ☹
  - Regularization ☺

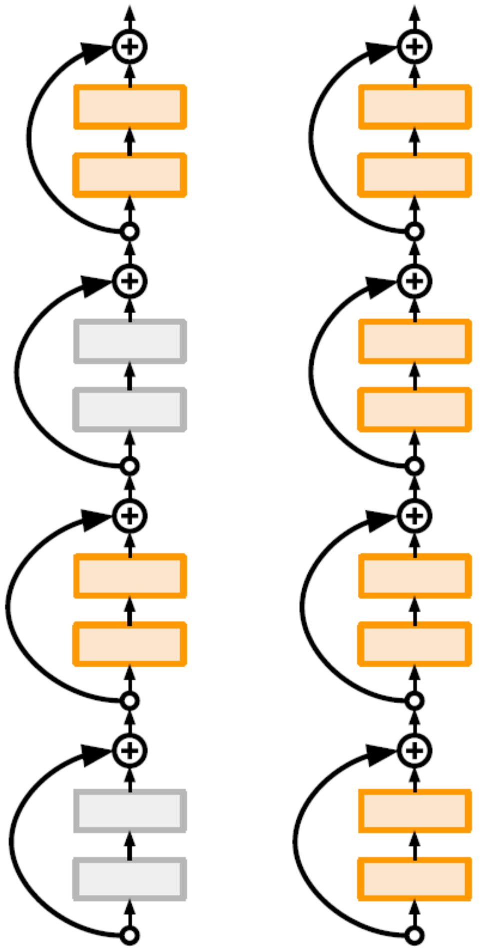
- Data Augmentation
- L1 regularisation
- L2 regularisation
- Dropout
- Batch Normalization
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout / Random Crop
- Mixup

[Wan et al. 2013]



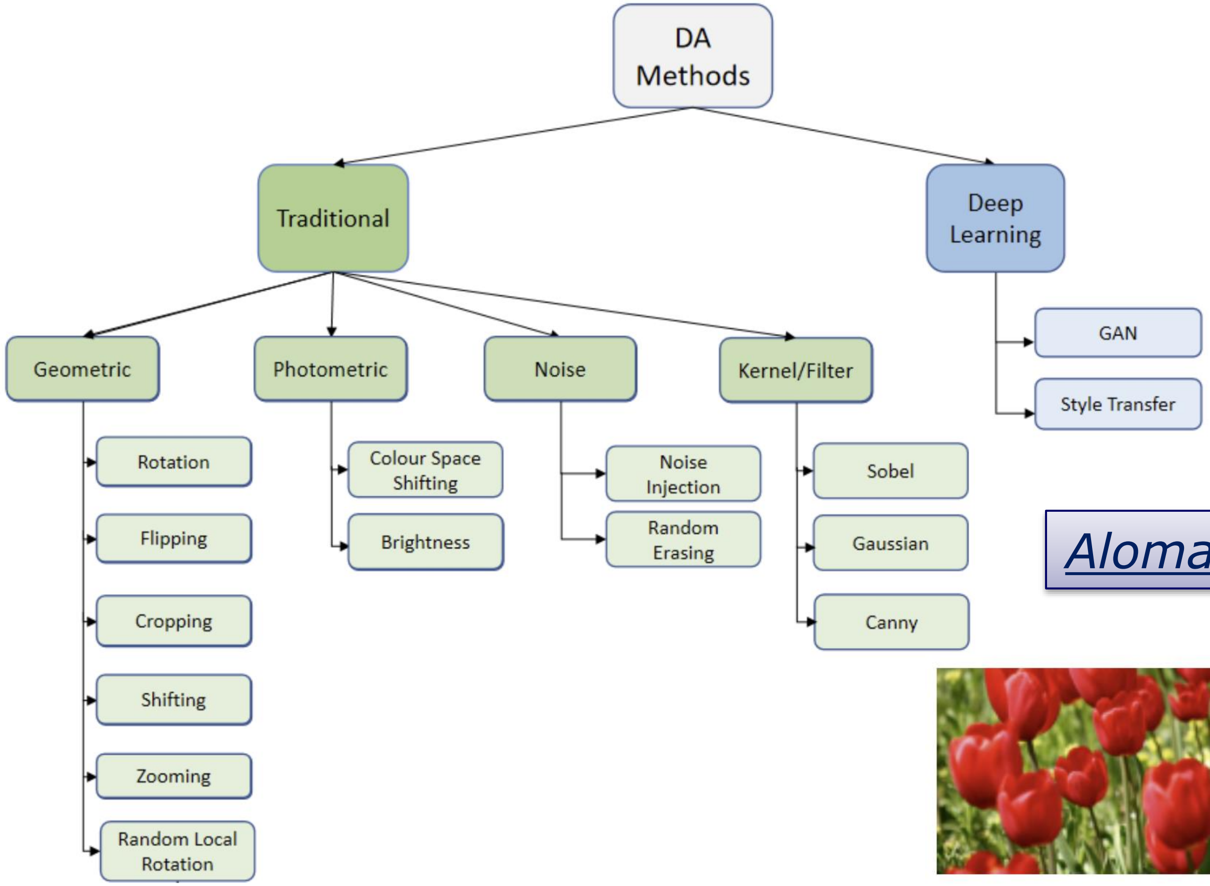


[Graham, 2014]

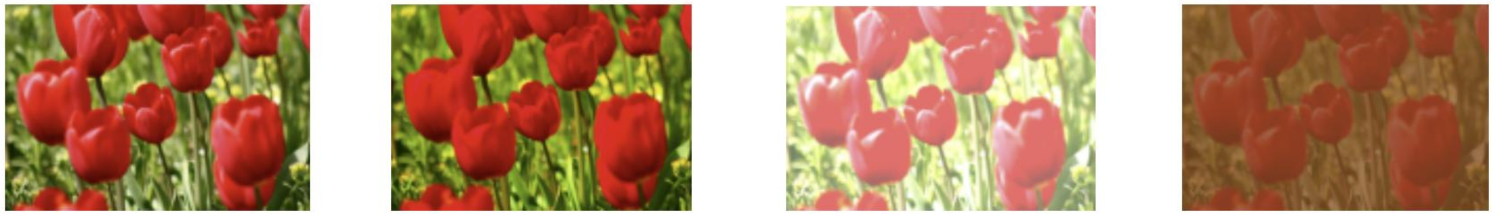


[Huang et al. 2016]

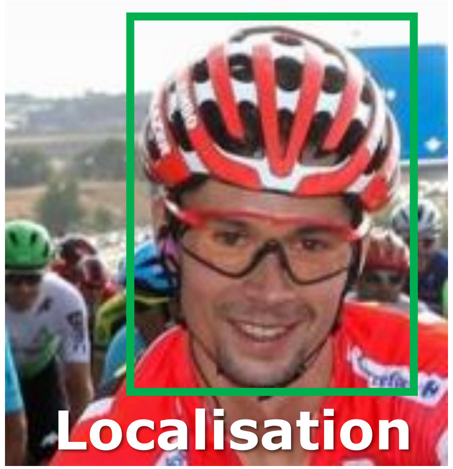
Data augmentation

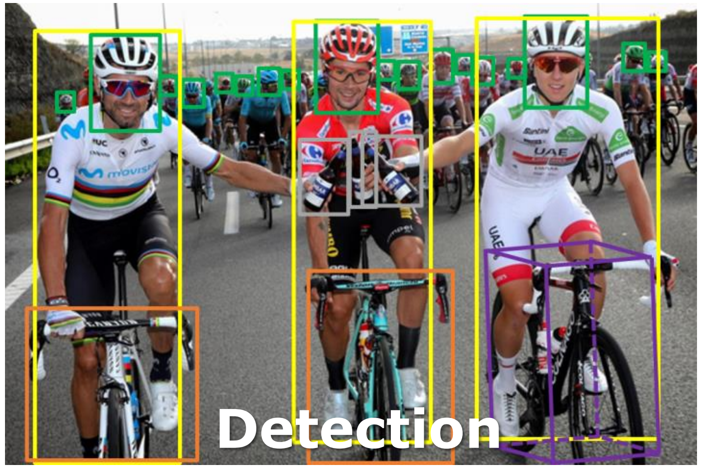


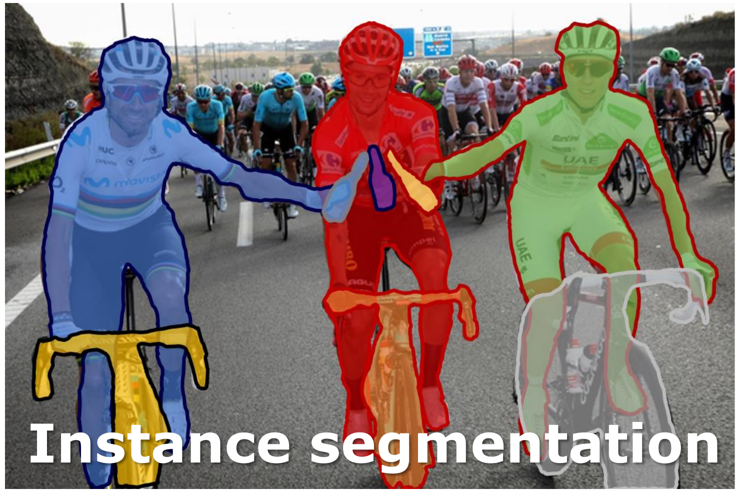
Alomar et al., 2023

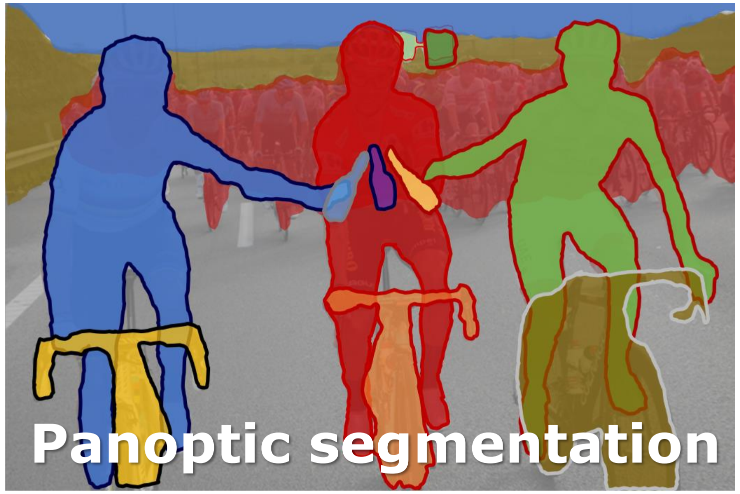


Main computer vision tasks









Classification

- What is depicted in the image?

Categorisation

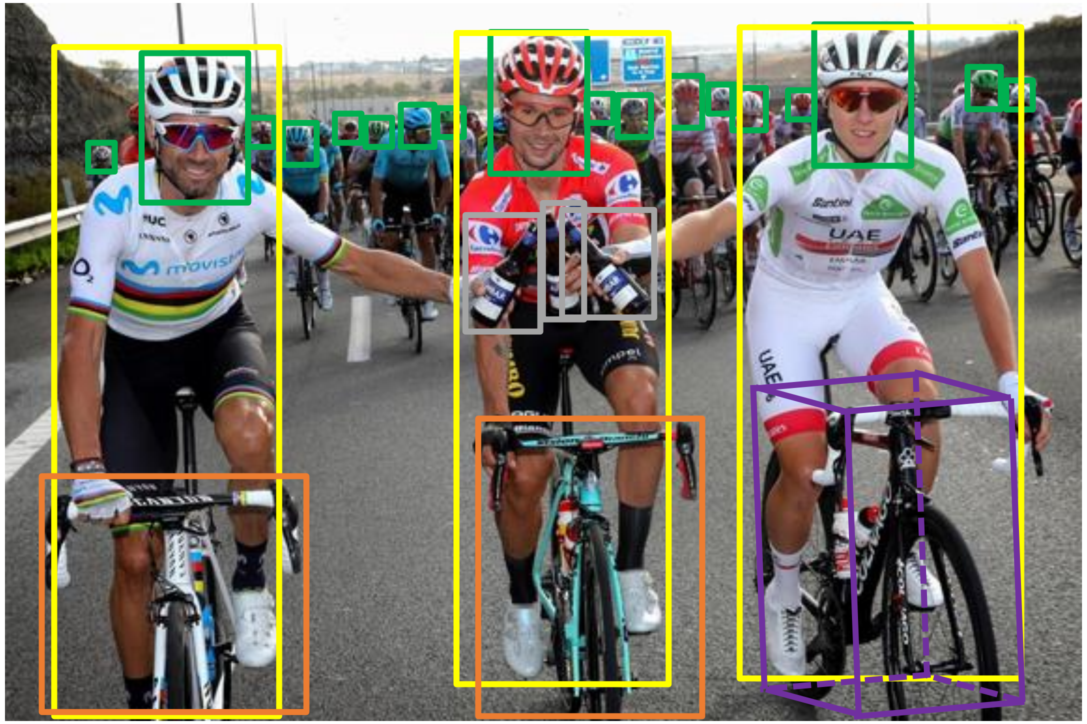
Localisation

Recognition/identification of instances

Detection

- Where in the image?

Detection



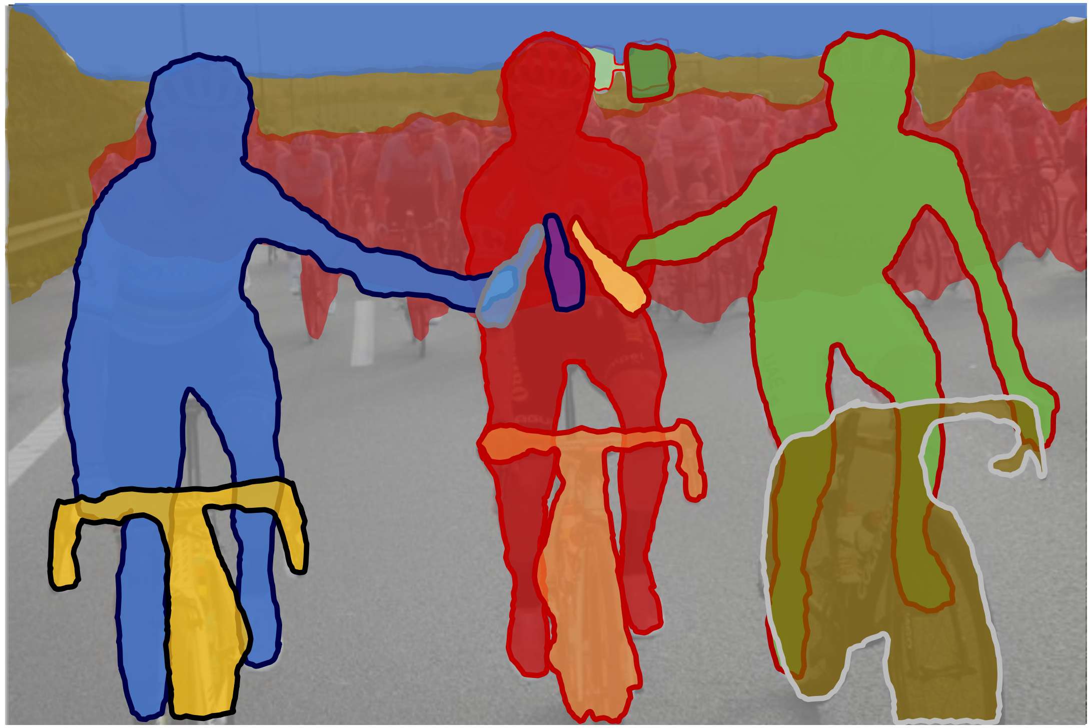
Instance segmentation

Segmentation

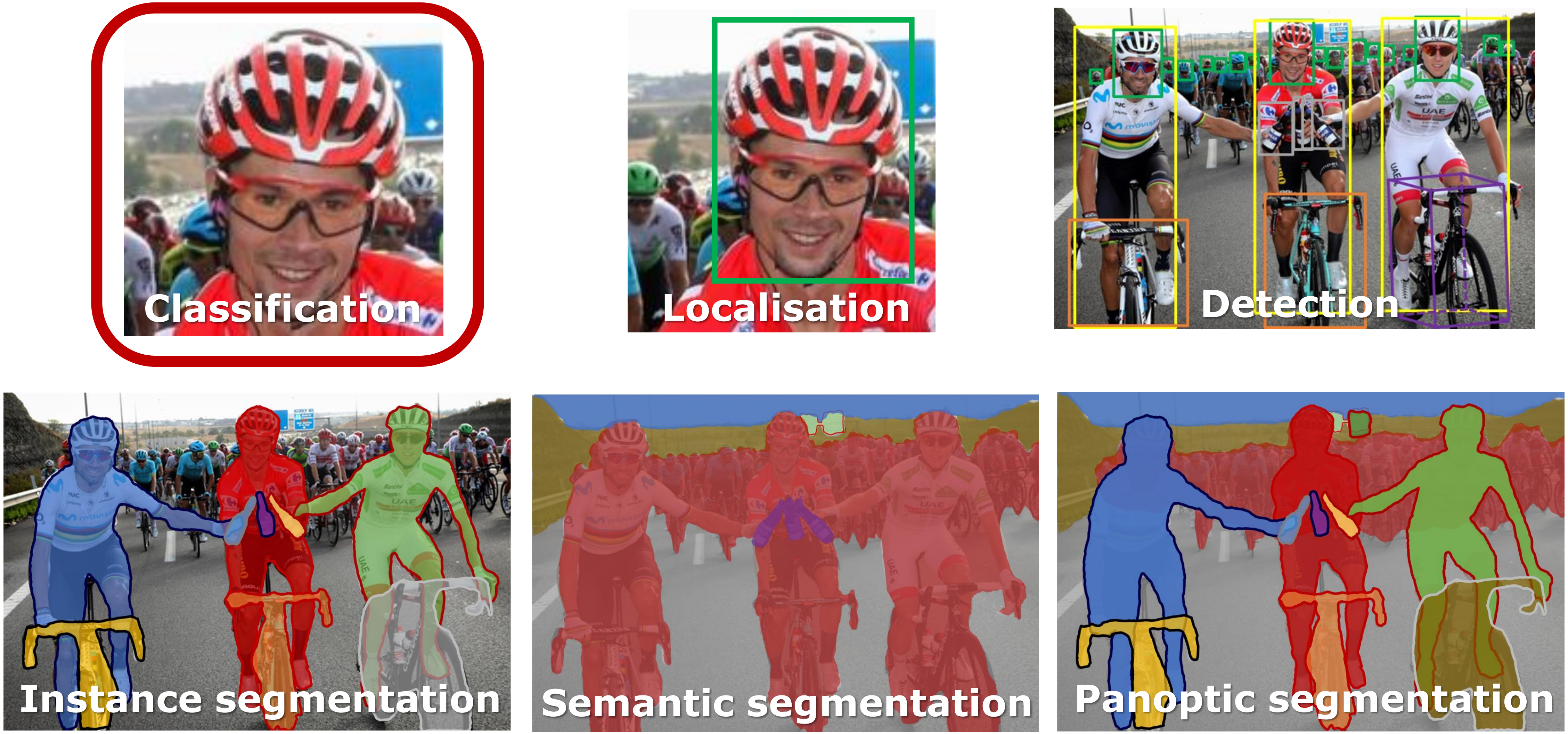
- What does every pixel represent?

Semantic segmentation

Panoptic segmentation

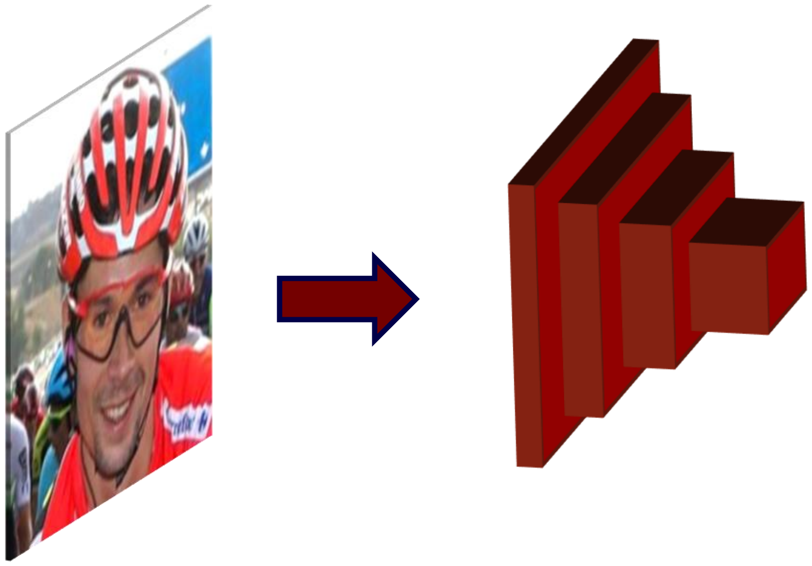


Classification

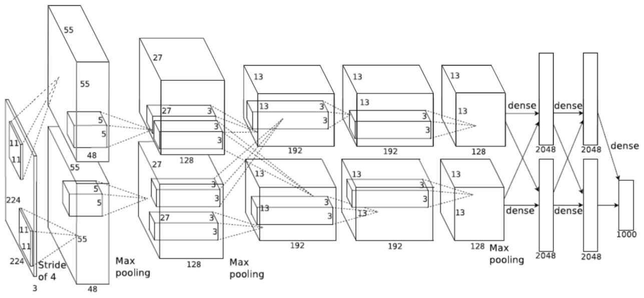


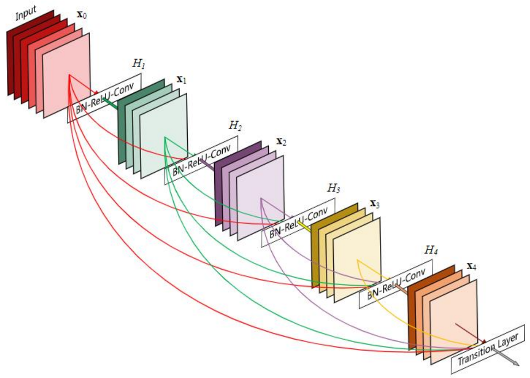
Classification

- Image classification: What is in the image?

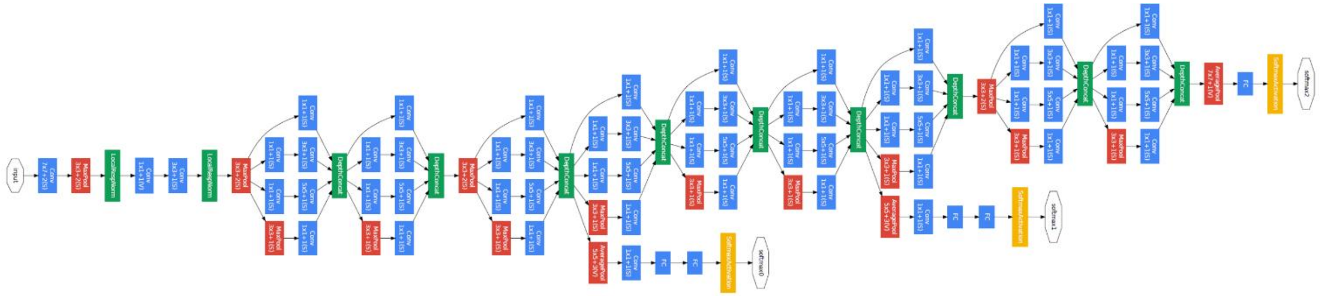


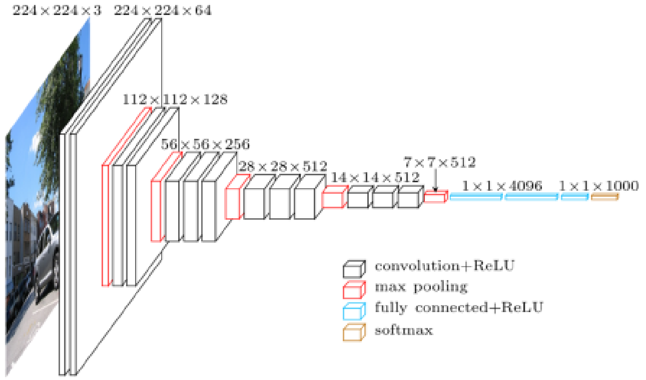
- T. Pogačar
- W. van Aert
- P. Roglič
- L. Dončić
- J. Oblak
- E. Klinec

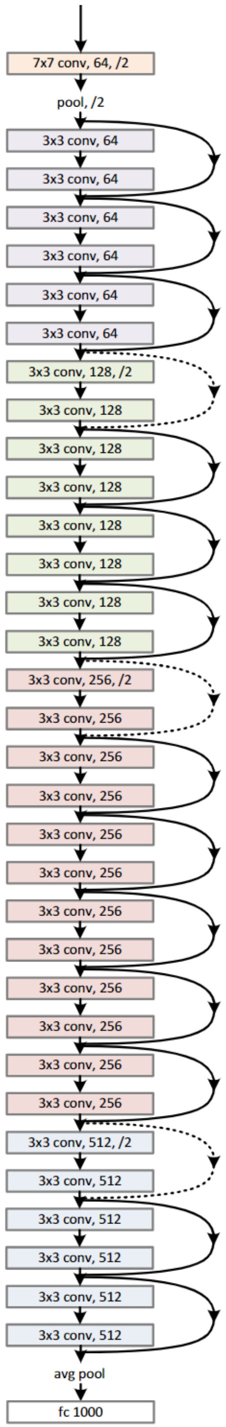




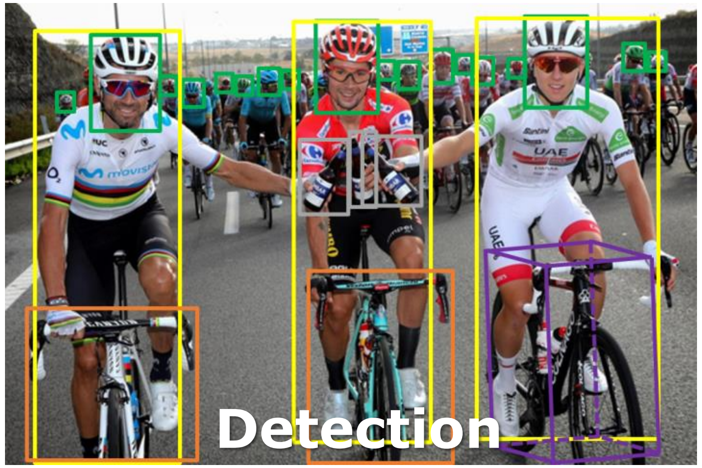
- Typically Cross entropy loss is used
- Any CNN backbone architecture can be used

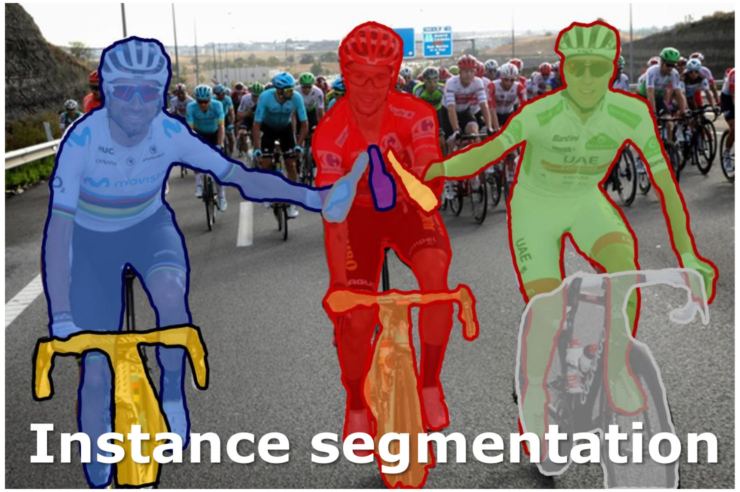


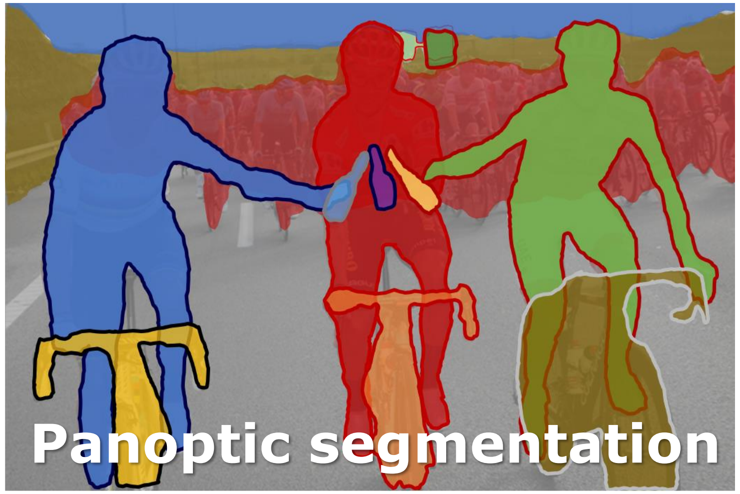




Localisation

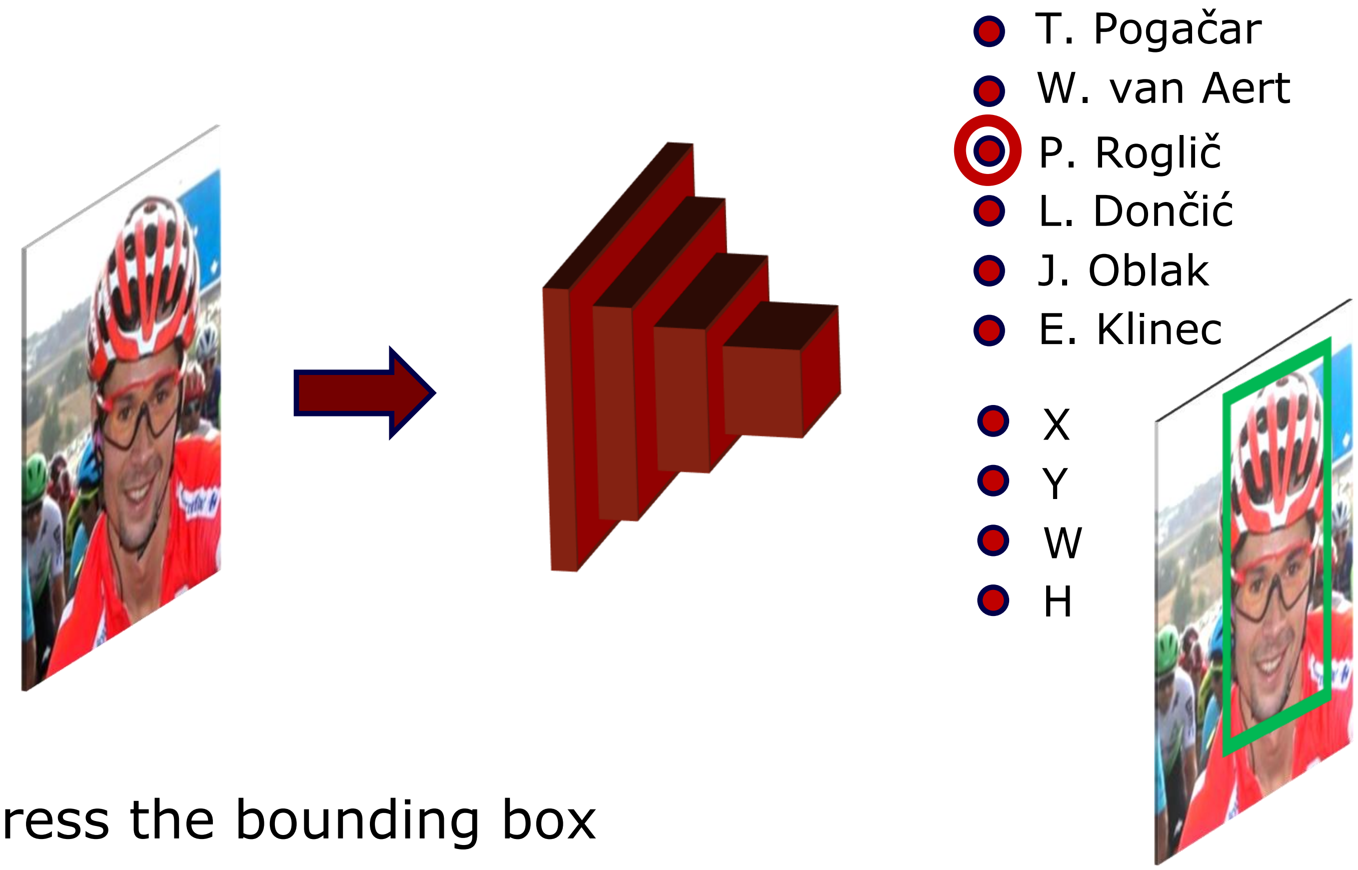






Localisation

- Object localisation – Where (besides what) in the image (is the only object)?



Classification loss
(Cross entropy)

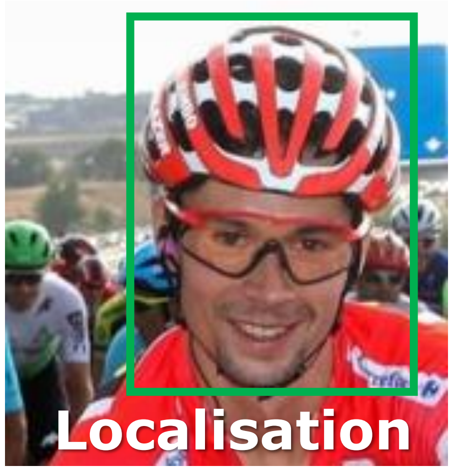
\+ = Multitask loss

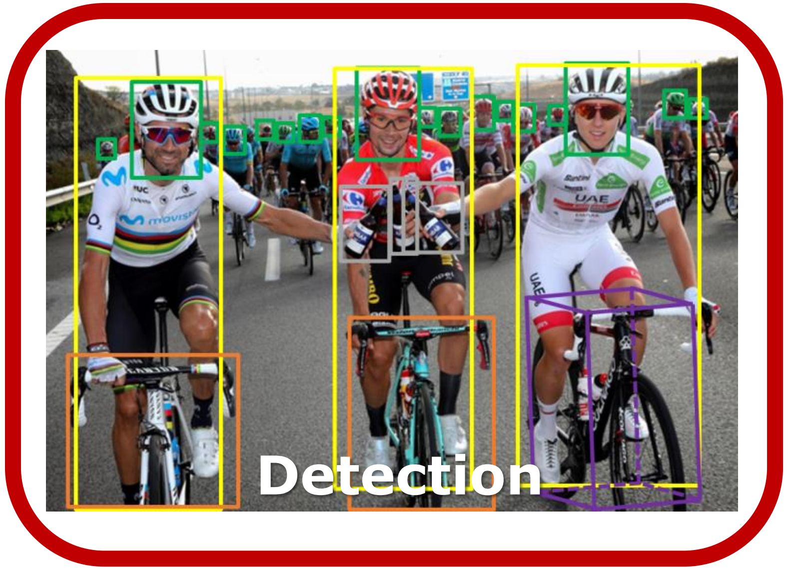
Regression loss
(L2)

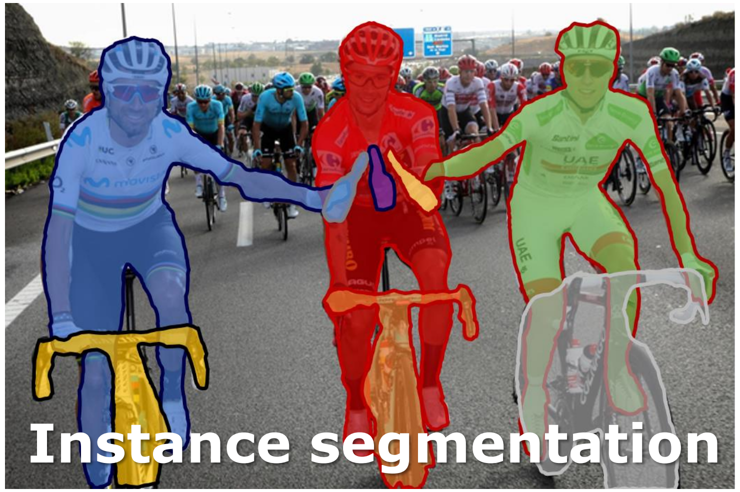
- Regress the bounding box

Detection

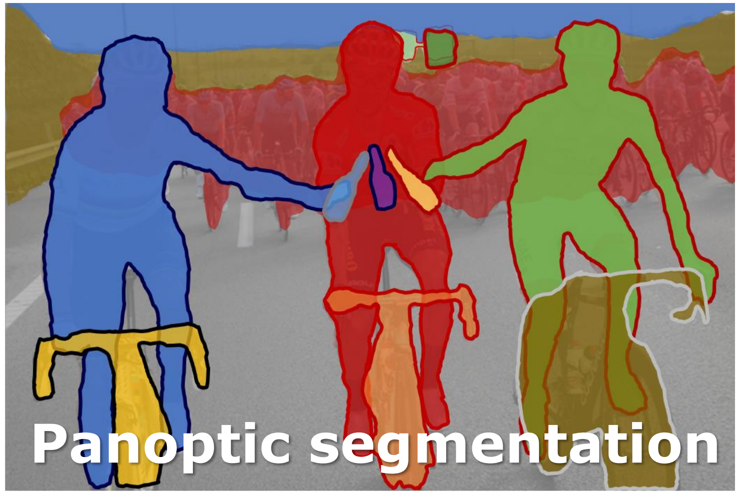

Classification


Localisation


Detection

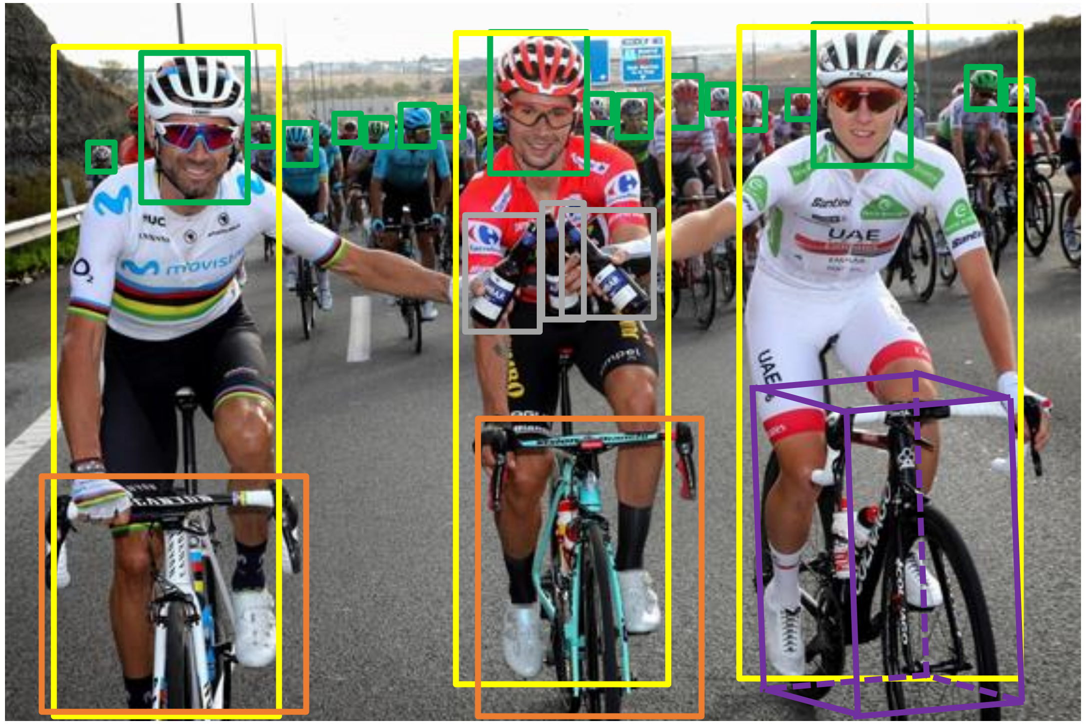

Instance segmentation


Semantic segmentation


Panoptic segmentation

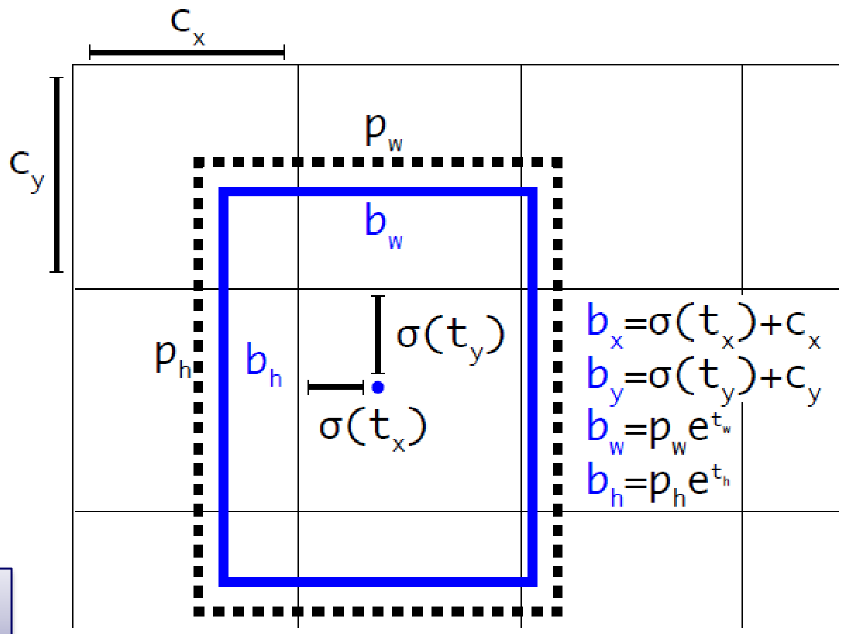
Detection

- Object detection – detect (localise and categorise) all the objects in the image
  - Unknown (arbitrary) number of objects
- Naive approach: Sliding window + classification
  - Too many locations, scales, aspect ratios!
  - Very expensive!



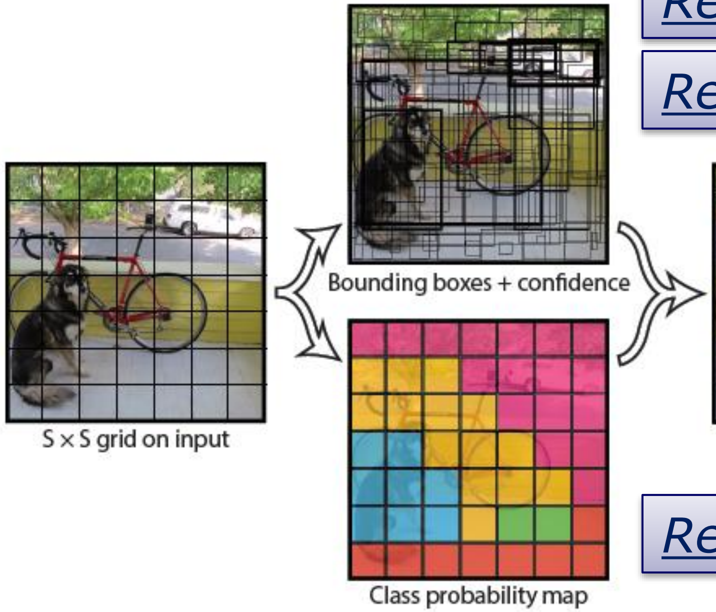
YOLOv3

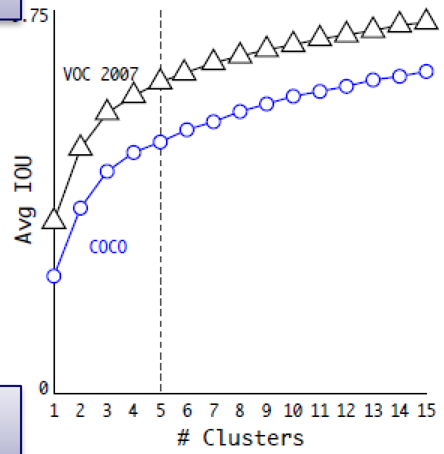
- You Only Look Once
- Prediction of bounding boxes on 3 scales
- 3 anchors as prior box shapes
- Prediction of objectness score for each BB
- Multilabel classification of each box
- Non-maxima suppression
- Real-time performance

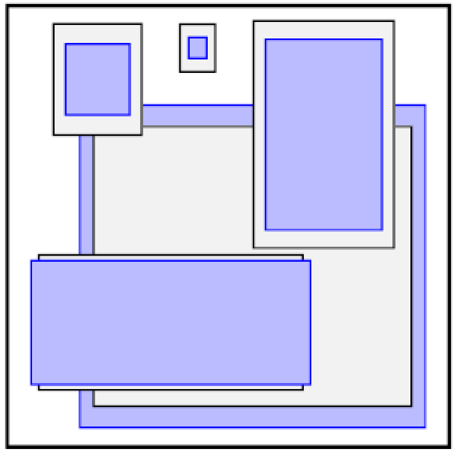


[Redmond et al., 2016](#)

[Redmond et al., 2017](#)

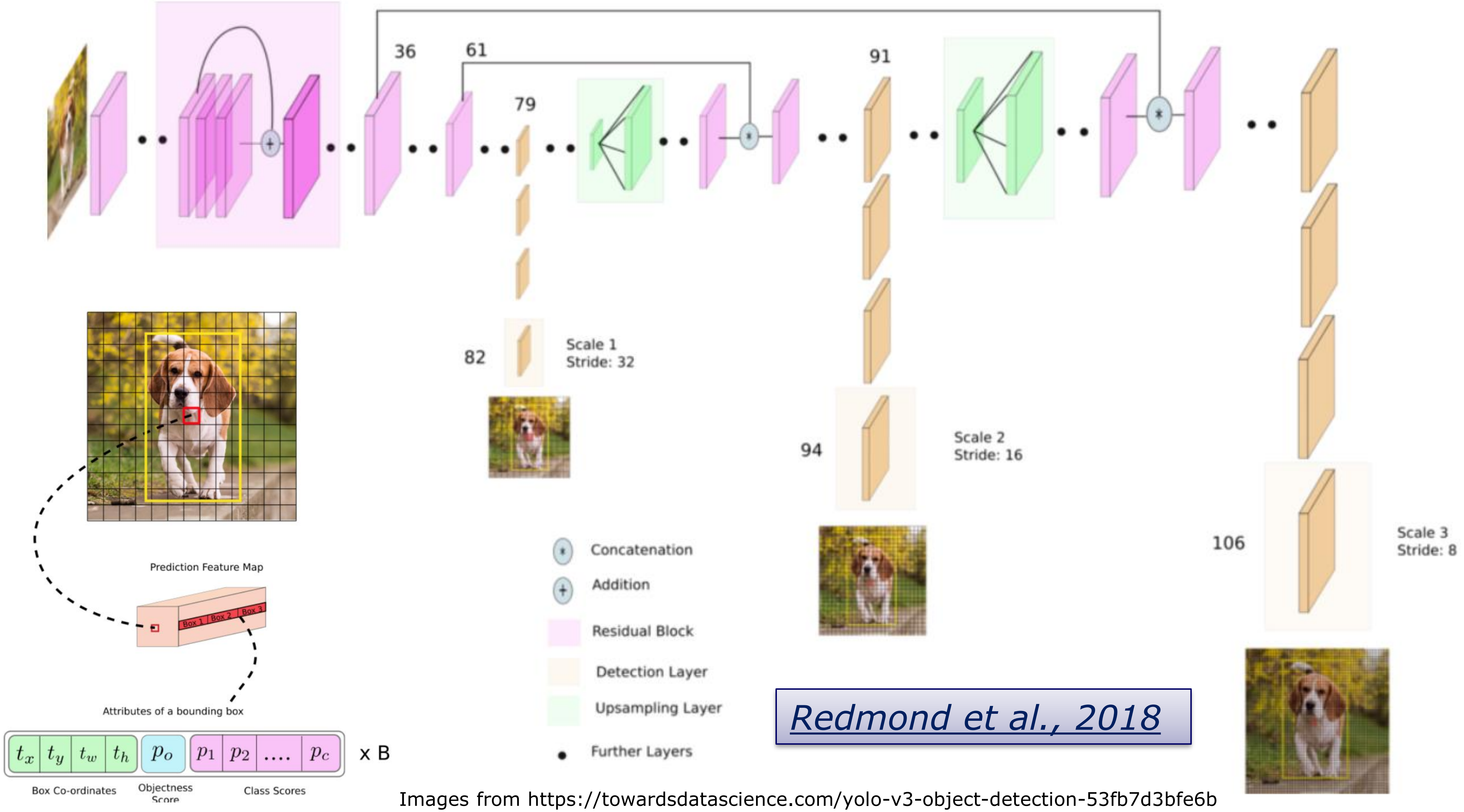






[Redmond et al., 2018](#)

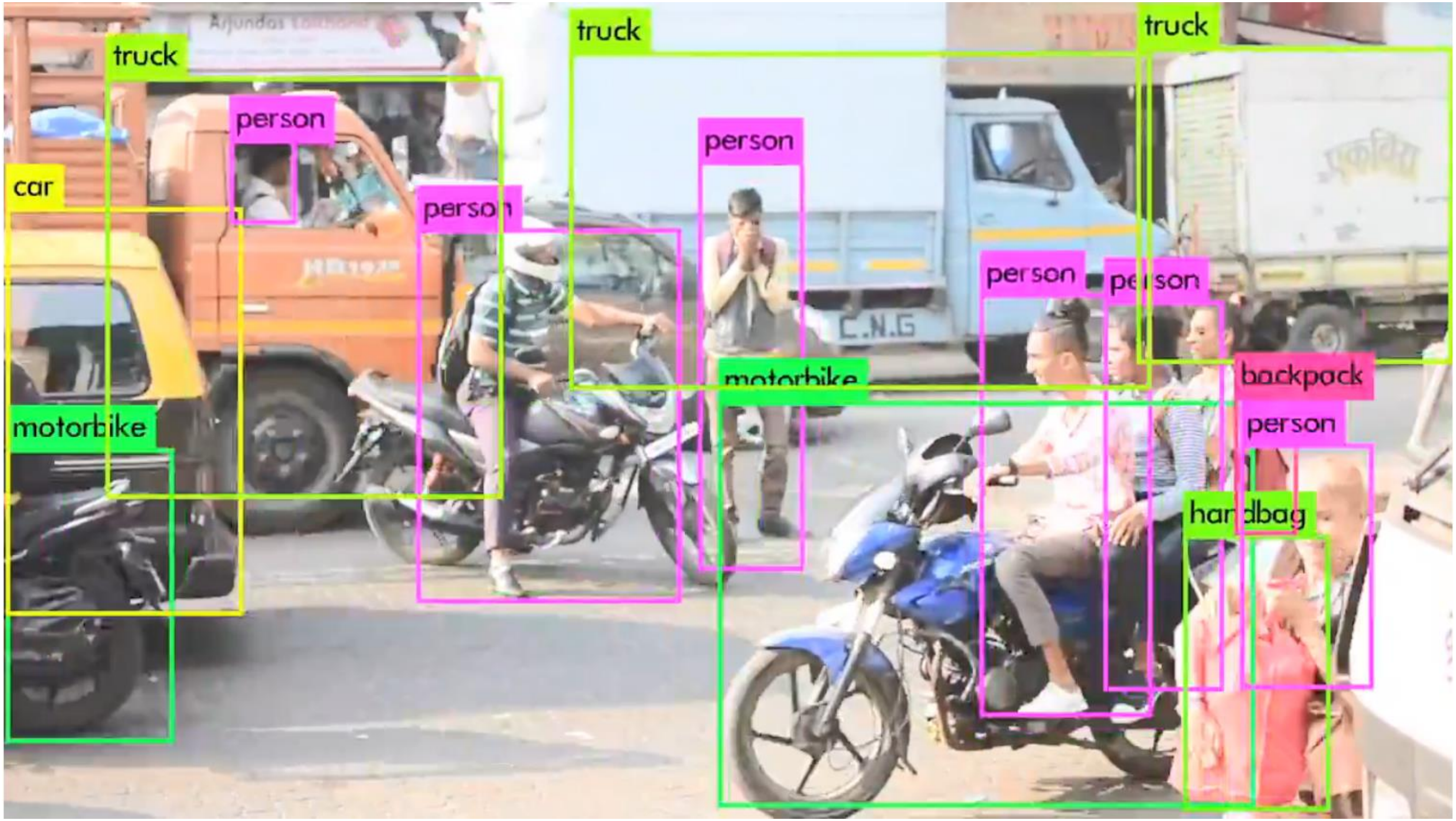
YOLOv3



[Redmond et al., 2018]

Images from https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

YOLOv3 results

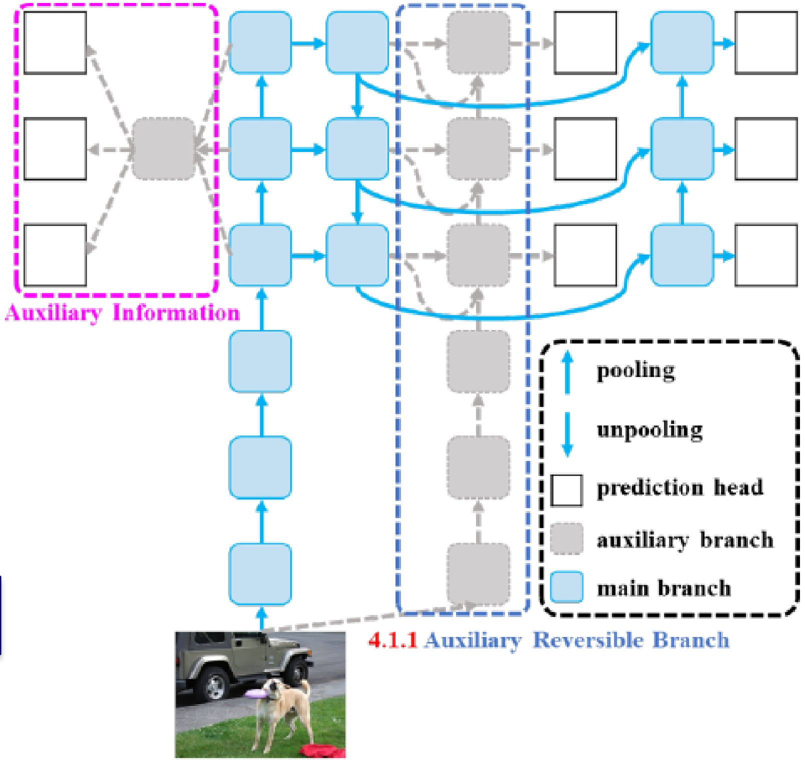


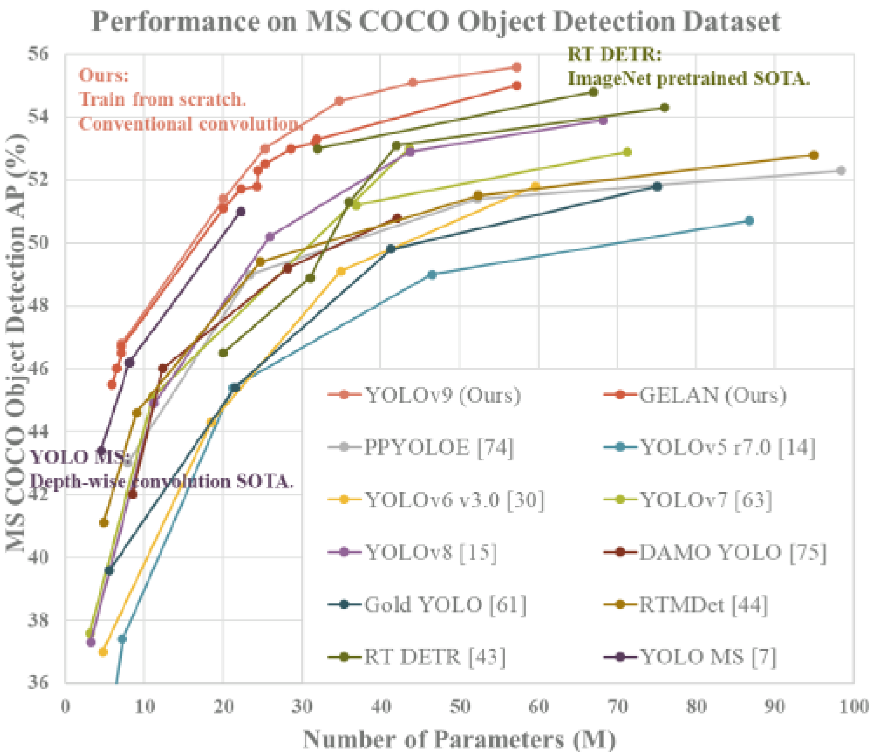
YOLO versions

- YOLOv1 (2016): Introduced the concept of predicting bounding boxes and class probabilities directly from full images in one evaluation.
- YOLOv2 (YOLO9000) (2017): Improved speed and accuracy, introduced anchor boxes to predict more precise bounding boxes.
- YOLOv3 (2018): Featured detection at three different scales and a better backbone for feature extraction, increasing accuracy especially for small objects.
- YOLOv4 (2020): Enhanced speed and accuracy with new features like Weighted-Residual-Connections, Cross-Stage-Partial connections, Mosaic data augmentation.
- YOLOv5 (2020): Focus on simplicity and speed, with PyTorch implementation and scalable to various devices.
- YOLOv6 (2021): Aims to balance the trade-off between accuracy, speed, and model size, enhancing cross-platform flexibility.
- YOLOv7 (2022): Improved upon predecessors with better architecture and training strategies, achieving SOTA performance, additional tasks, such as pose estimation.
- **YOLOv8** (2023): Focuses on optimizing model efficiency and deployment, introducing new techniques for faster inference and better accuracy, full range of vision AI tasks, including detection, segmentation, pose estimation, tracking, and classification.
- YOLOv9 (2024): Latest iteration aiming at maximizing real-time performance while maintaining high accuracy, leveraging the latest advancements in deep learning.

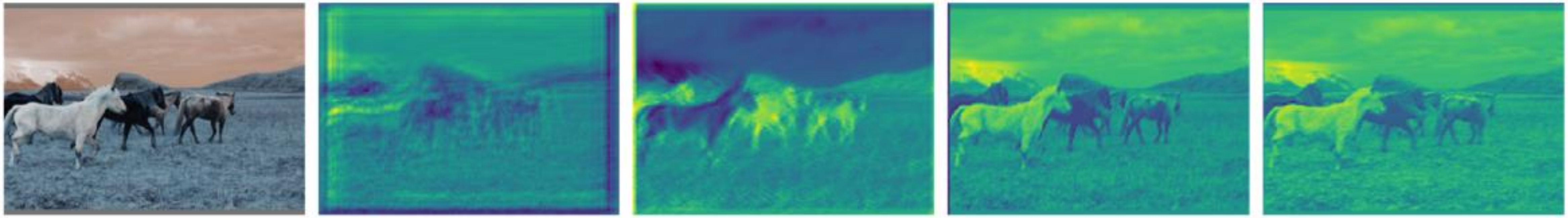
YOLOv9

- Programmable Gradient Information (PGI)
- Generalized Efficient Layer Aggregation Network (GELAN)





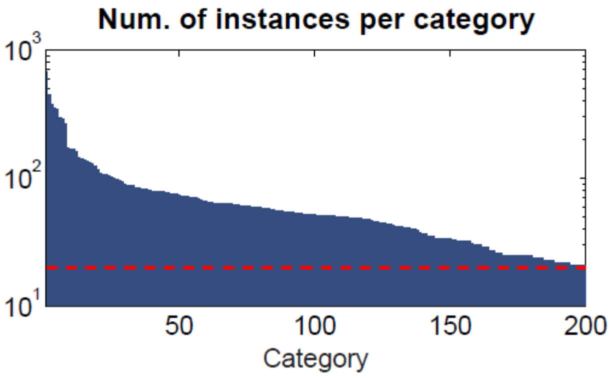
Wang et al., 2024



(a) Input Image (b) PlainNet (c) ResNet (d) CSPNet (e) GELAN

Detection of traffic signs

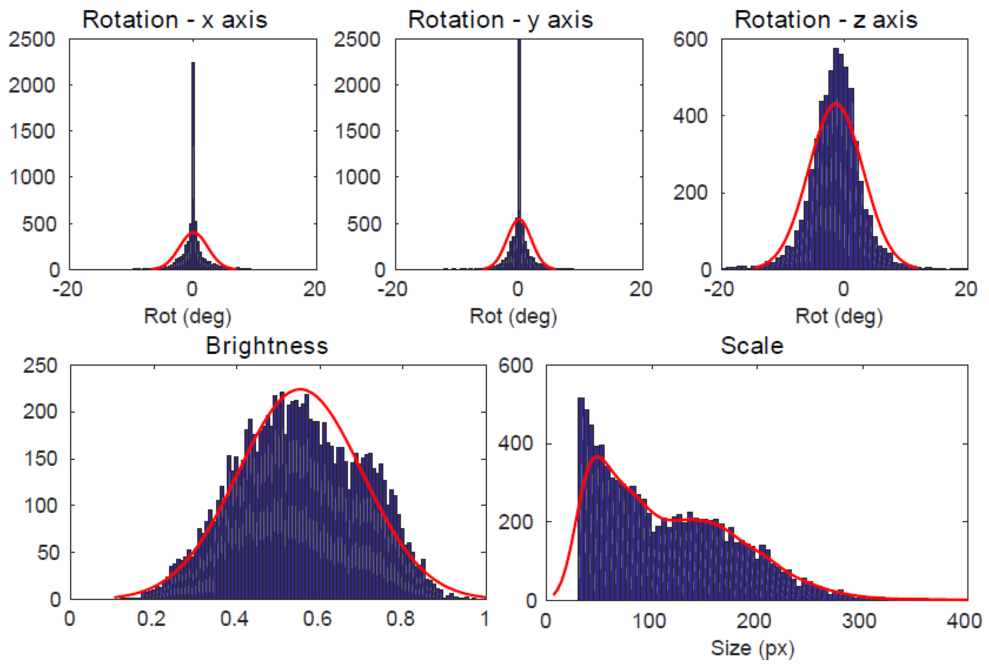
- DFG dataset
- 200 categories
- 6.957 images
- 13.239 signs

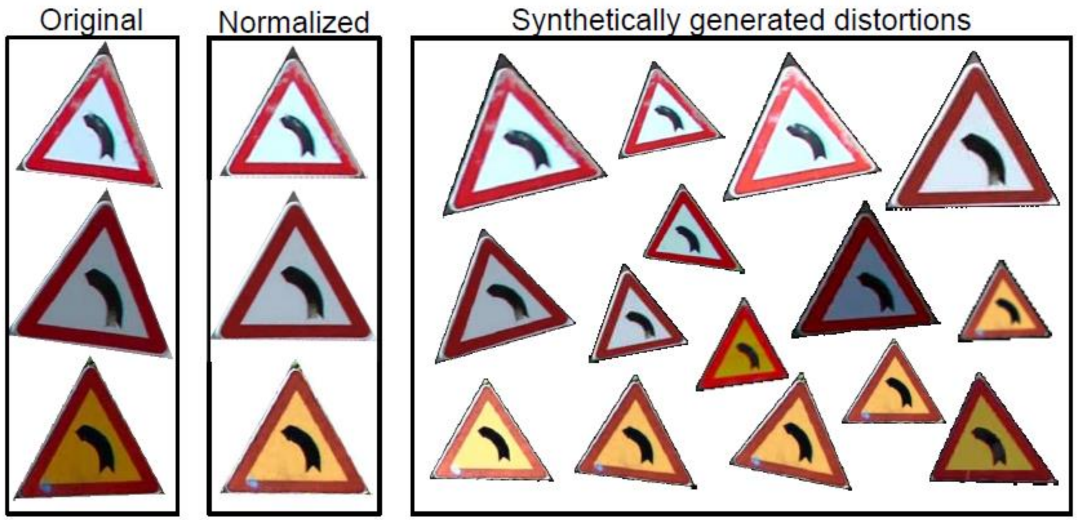


Tabernik & Skočaj, 2020

Detection of traffic signs

- Data augmentation

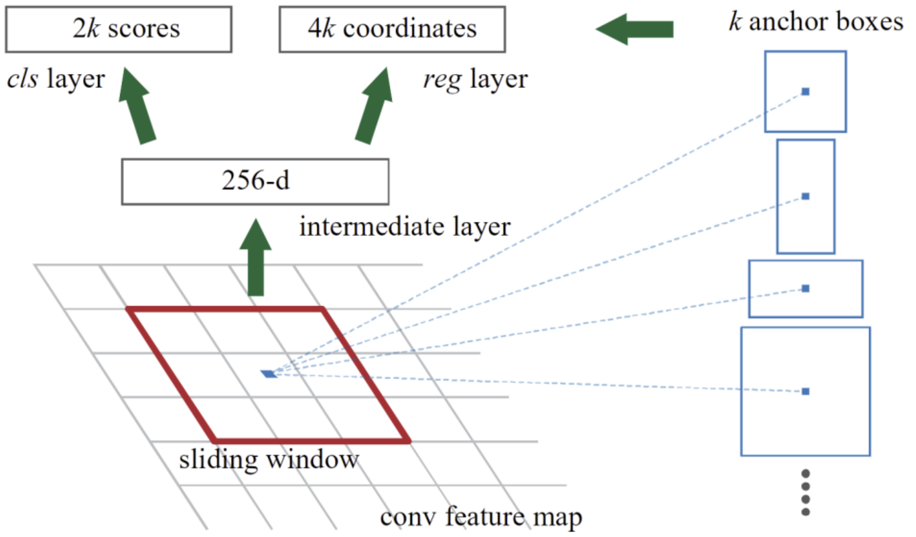


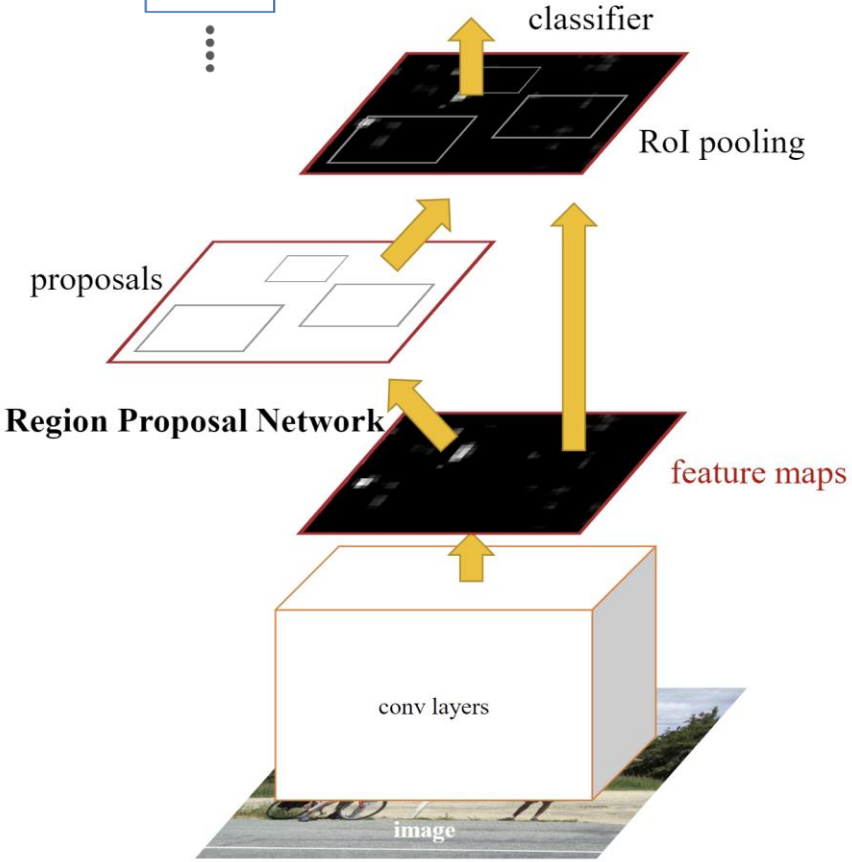


- Mask R-CNN +
  - Online hard-example mining
  - Distribution of selected training samples
  - Sample weighting
  - Adjusting region pass-through during detection

Faster R-CNN

- Region Proposal Network
  - Included in the method
  - Anchor boxes
  - Sliding window on feature map
- Two stage method (four losses)
  - Detect region proposals
    - Objectness score - RP cls loss (is object?)
    - Object bounds - RP BB loss (bb corrections)
  - Classify individual proposals
    - Cls loss (what it is?)
    - BB loss (refine RP BB)
- Alternating / end-to-end learning
- Significantly faster than Fast R-CNN
- SOTA in 2015

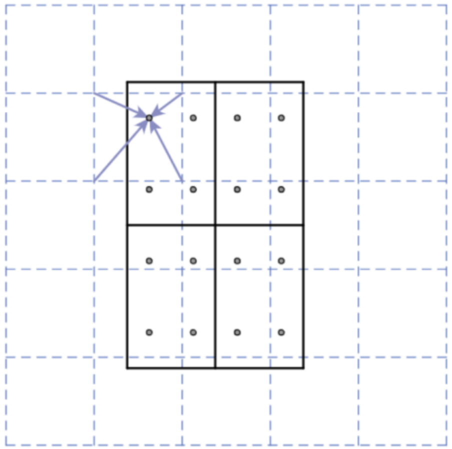


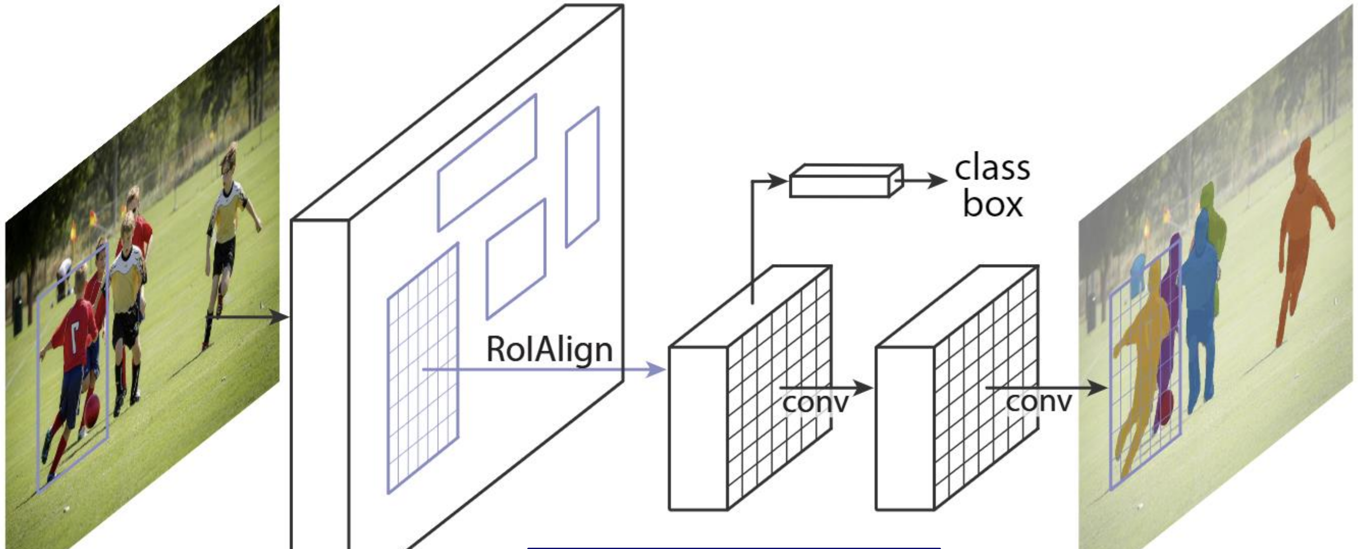


[*Ren et al., 2015*]

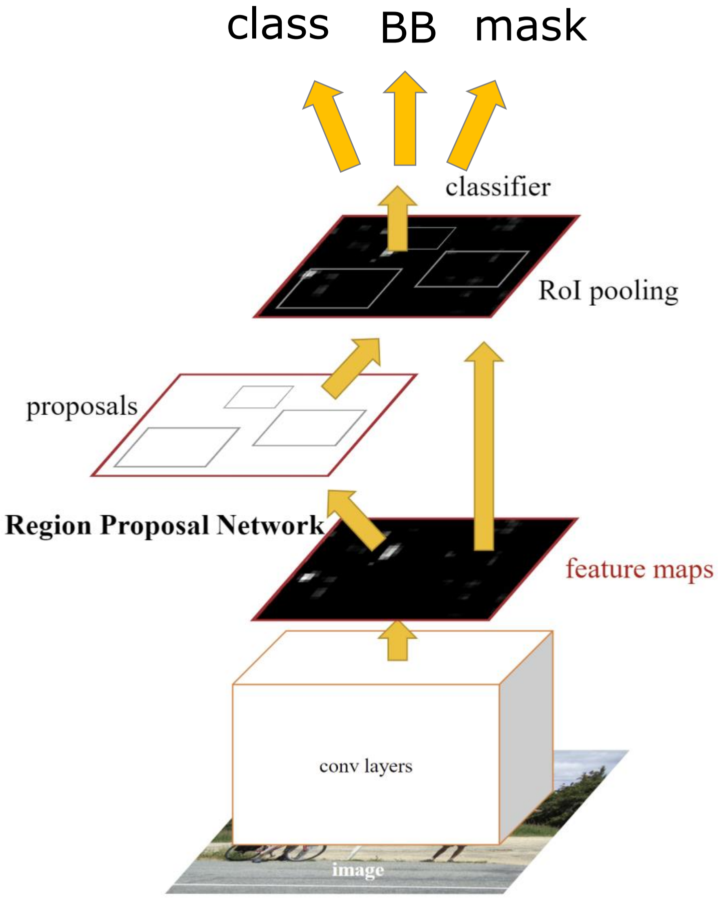
Mask R-CNN

- Add segmentation head
  - Additional segmentation loss
  - Produces segmentation mask for every RoI
- RoI align
- Other extensions possible



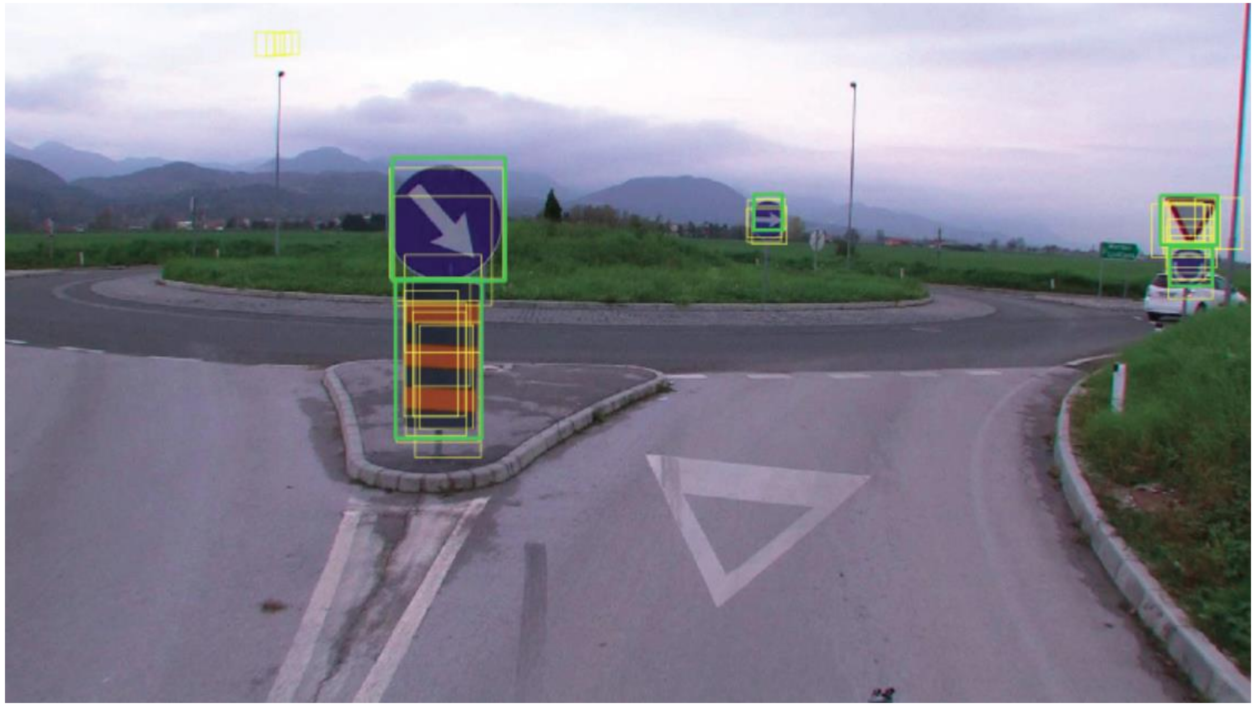


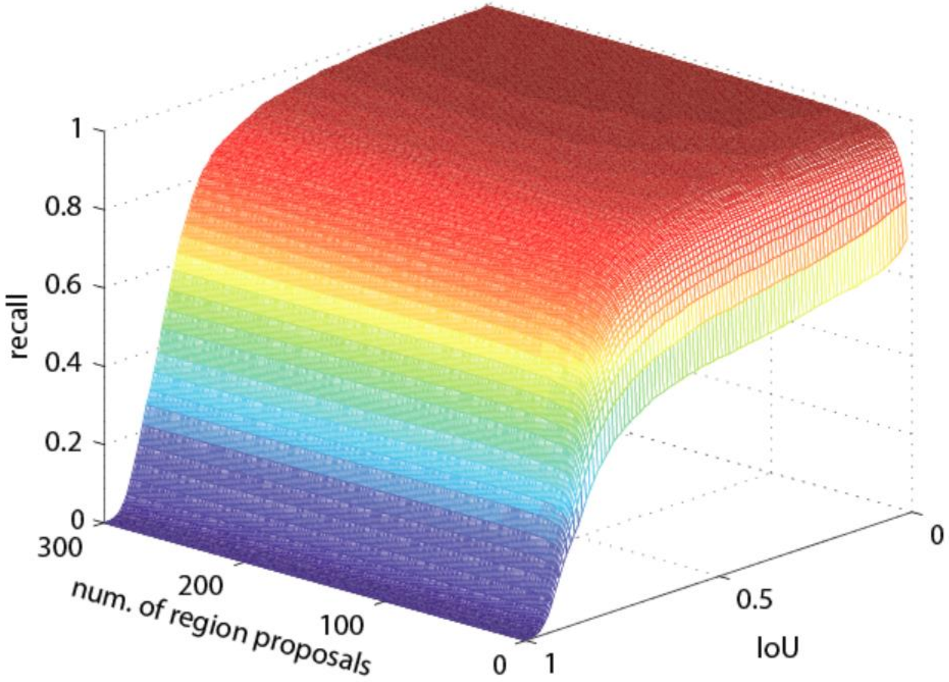
He et al., 2017



Detection of region proposals

- Top proposals are very good

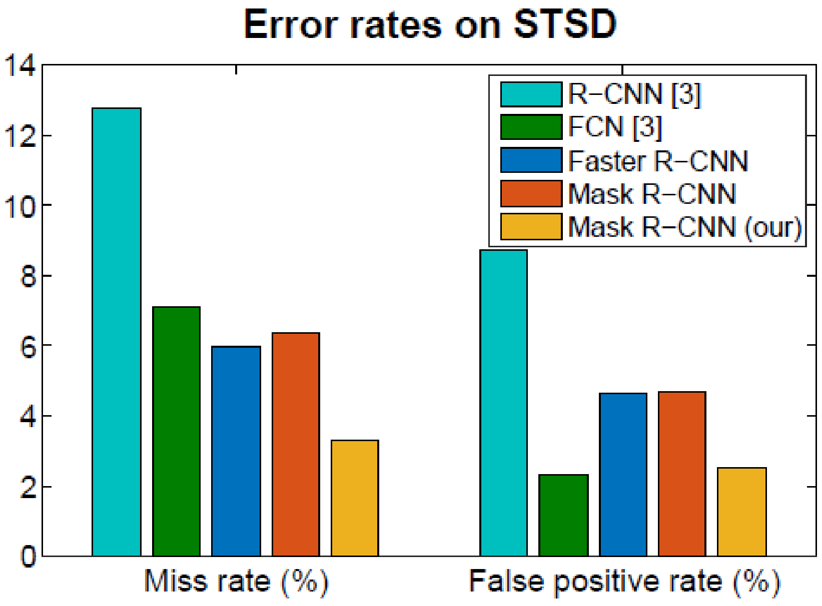




Experimental results

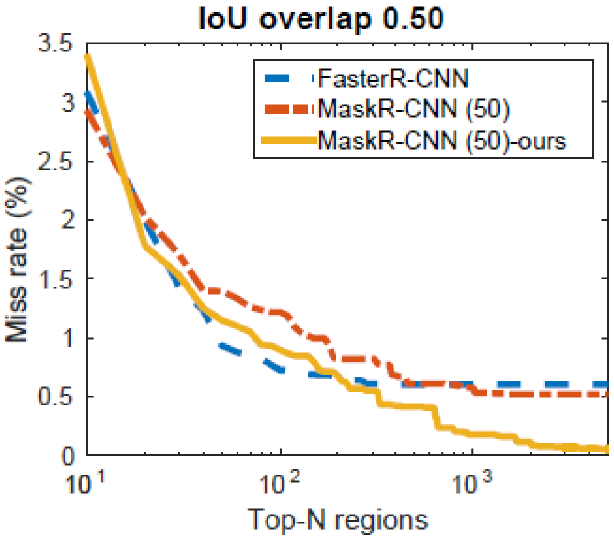
- Swedish traffic sign dataset

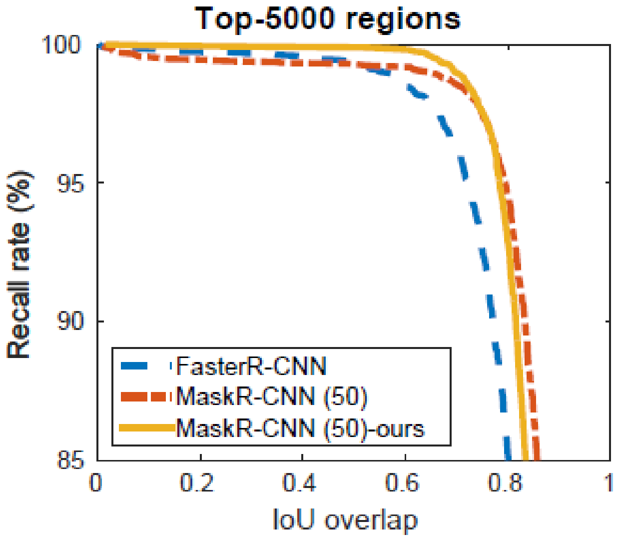
Average	R-CNN [6]	FCN [6]	Faster R-CNN	Mask R-CNN (ResNet-50) No adapt.	Mask R-CNN (ResNet-50) Adapt. (ours)
Precision	91.2	**97.7**	95.4	95.3	97.5
Recall	87.2	92.9	94.0	93.6	**96.7**
F-measure	88.8	95.0	94.6	93.8	**97.0**
mAP^{50}	/	/	94.3	94.9	**95.2**



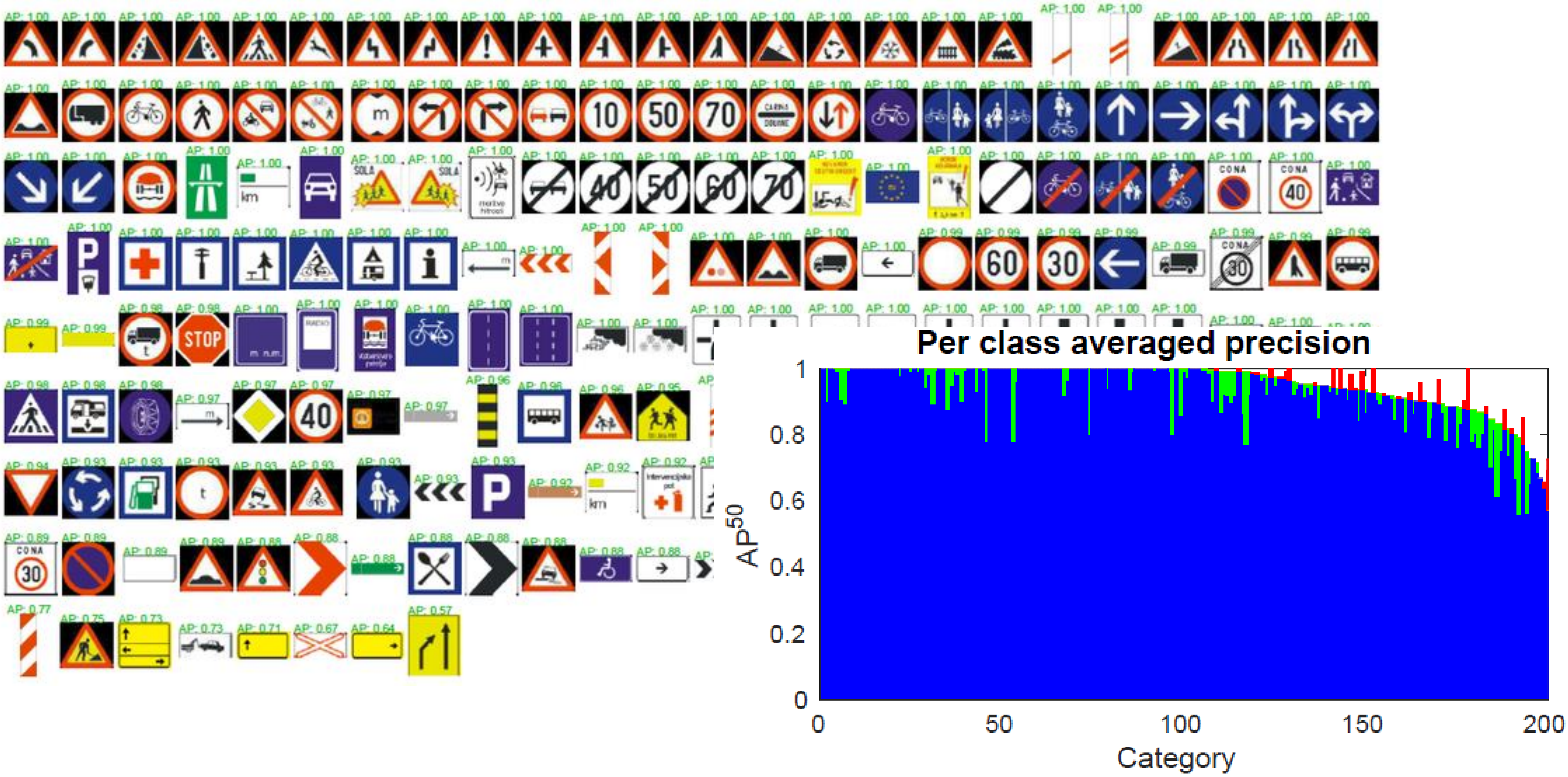
DFG traffic sign dataset

	Faster R-CNN	Mask R-CNN (ResNet-50) No adapt.	Mask R-CNN (ResNet-50) With adapt.	Mask R-CNN (ResNet-50) With adapt. and data augment.
mAP^{50}	92.4	93.0	95.2	**95.5**
$mAP^{50:95}$	80.4	82.3	82.0	**84.4**
Max recall	93.8	94.6	**96.5**	**96.5**

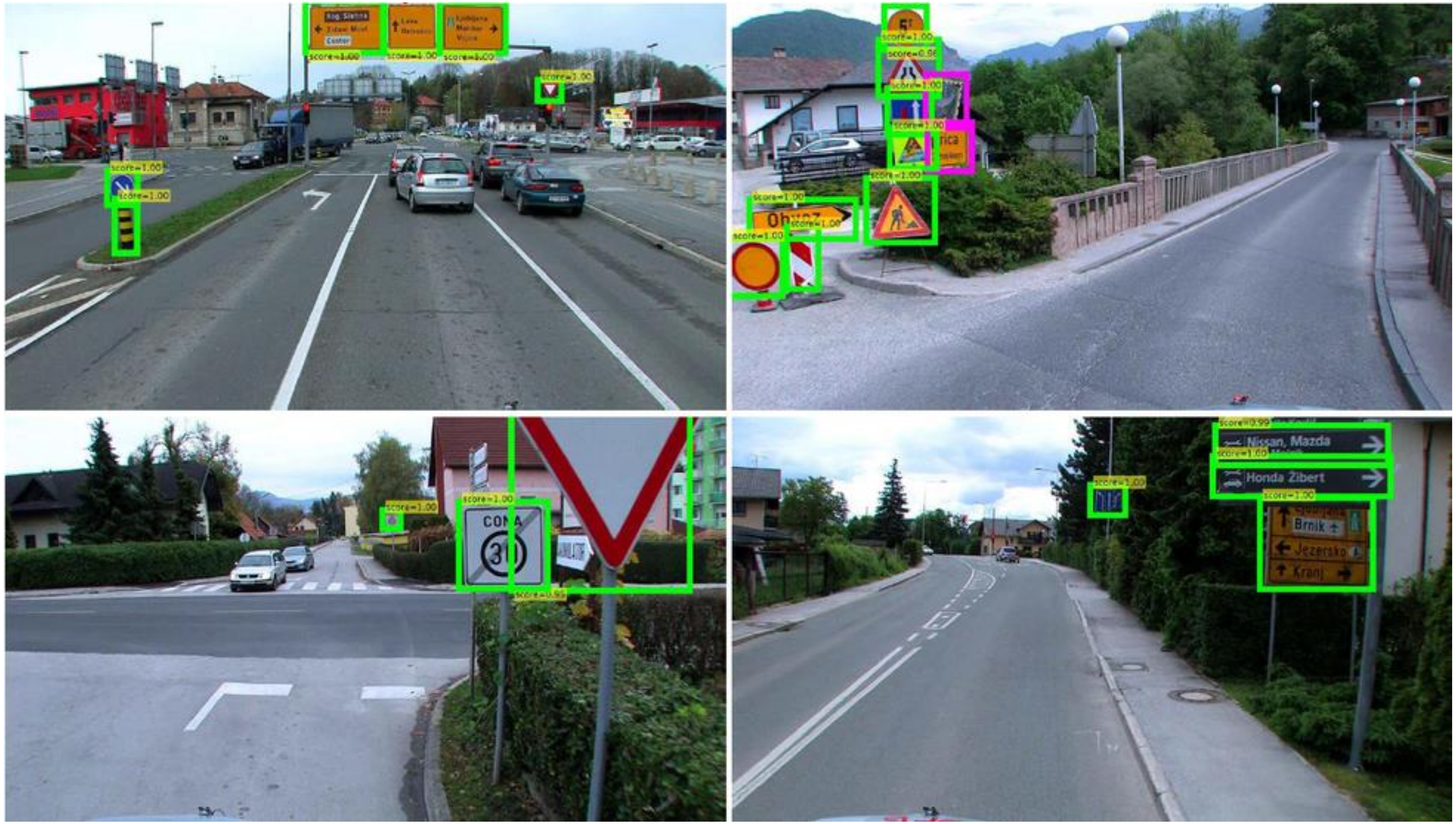




Experimental results



Experimental results



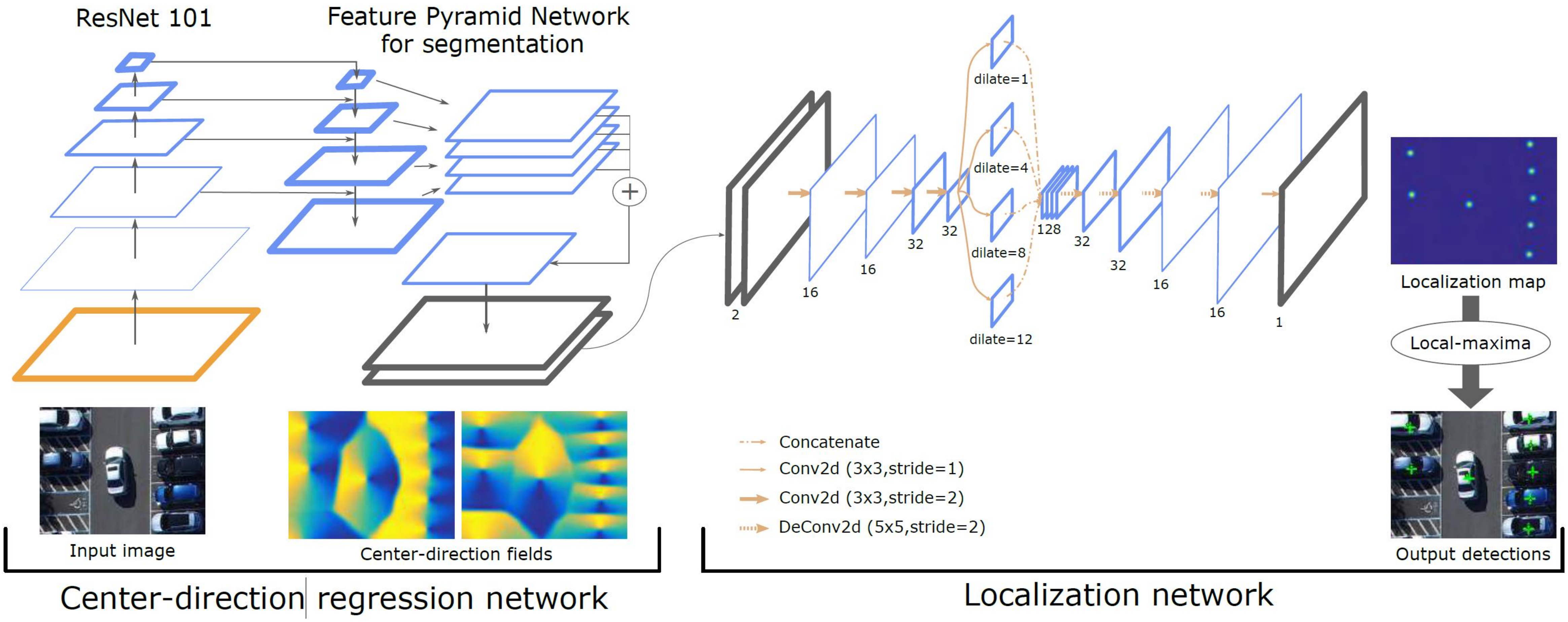
Experimental results

Traffic sign detection

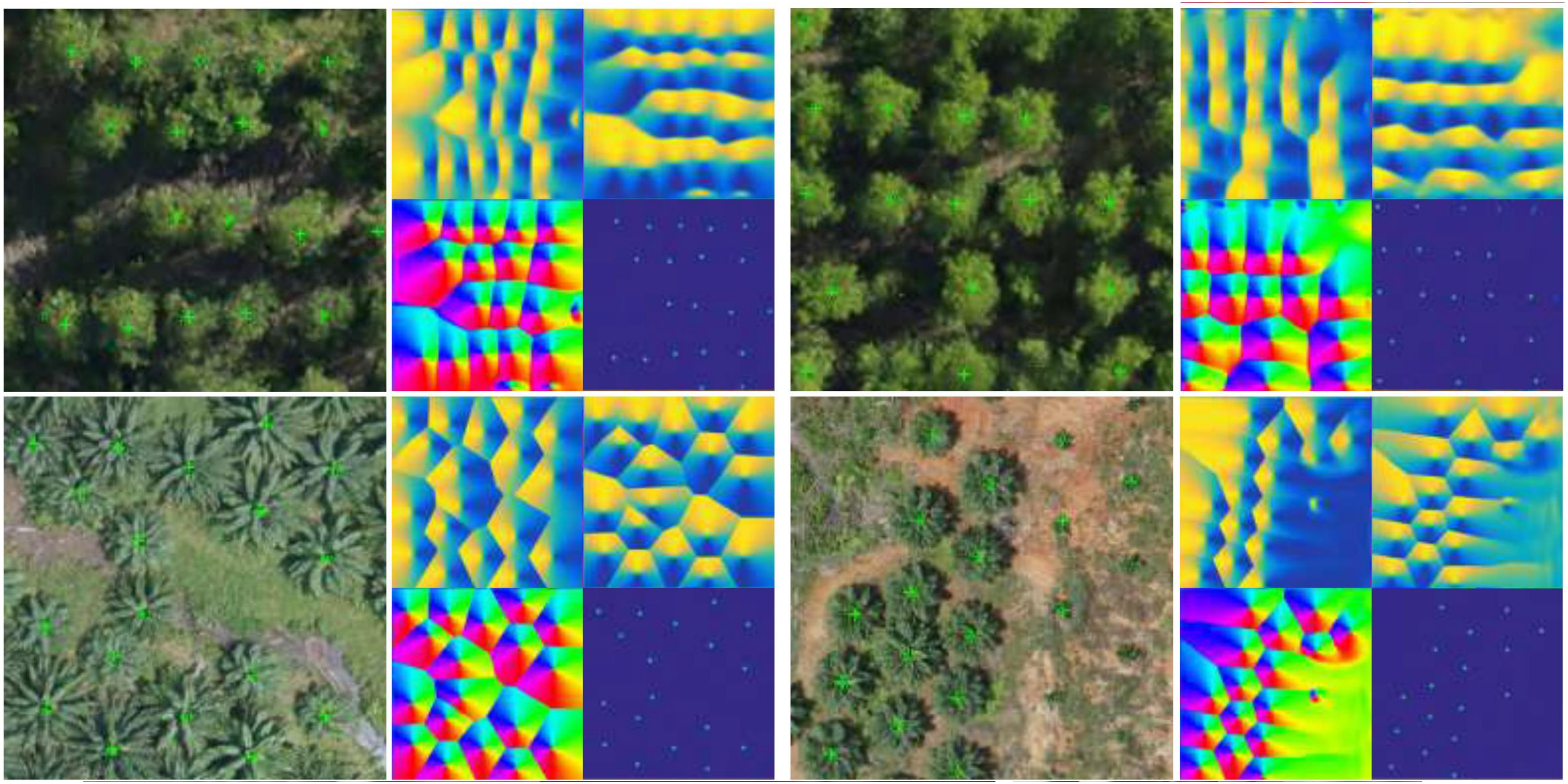
CeDirNet for object counting and localisation

- Dense Center-Direction Regression for Object Counting and Localization with Point Supervision

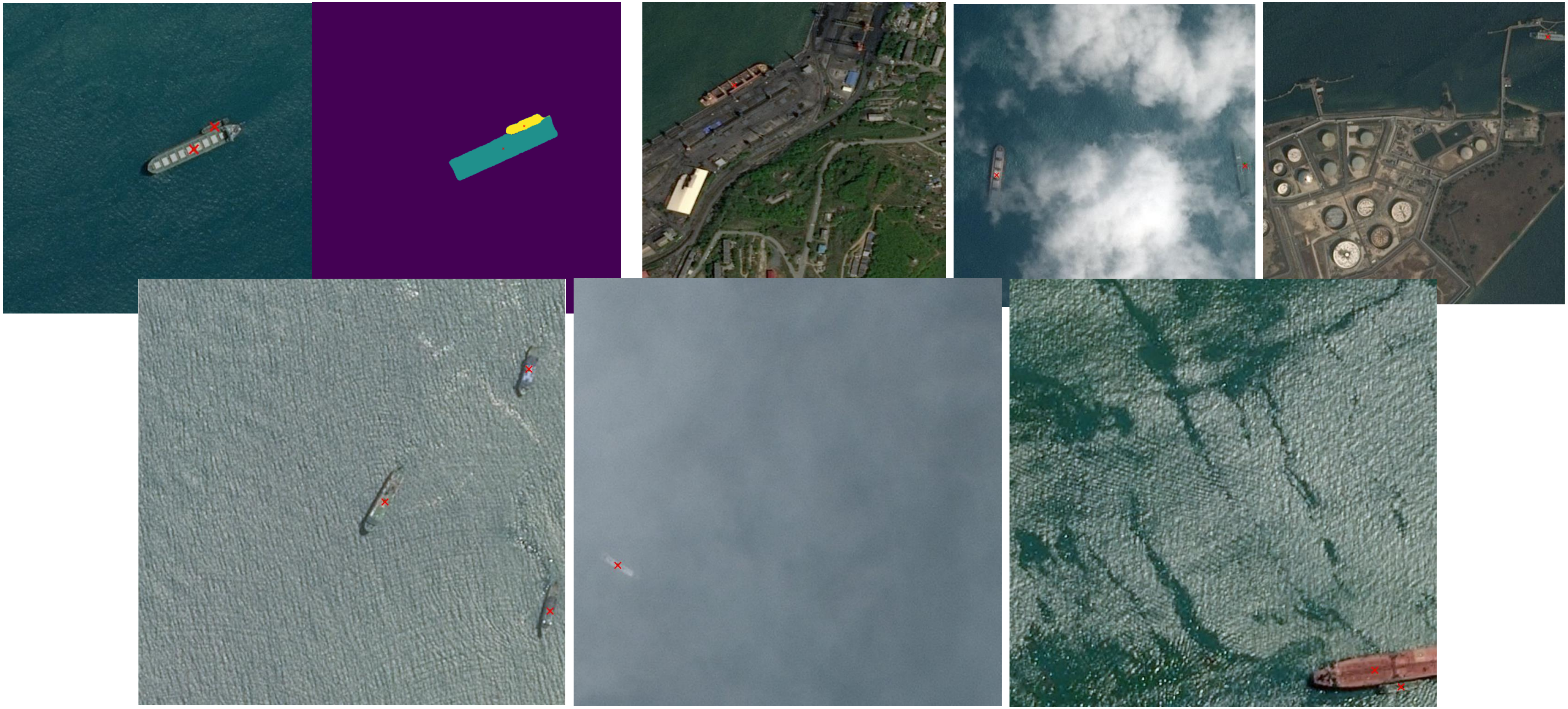
[Tabernik et. al, 2024]



CeDirNet

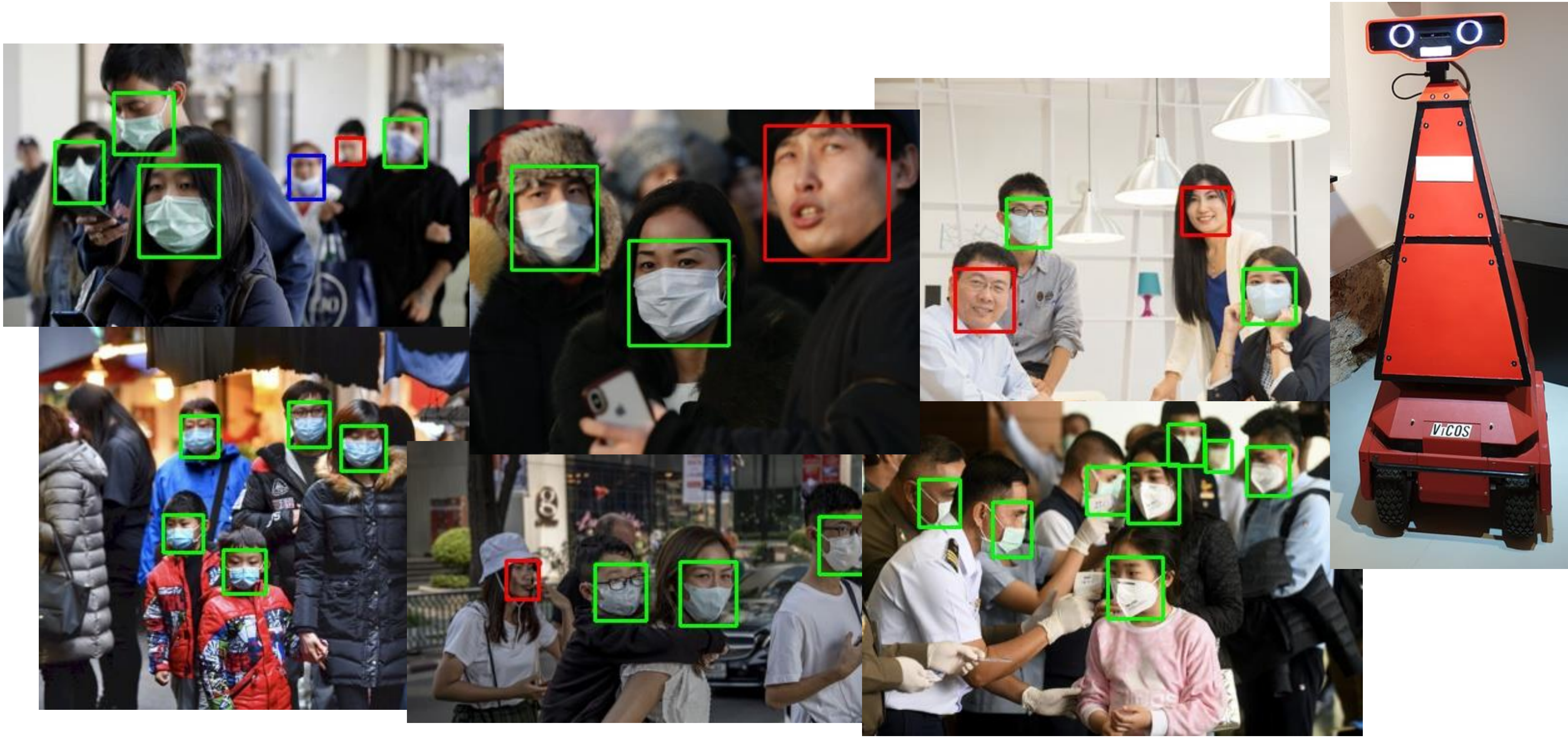


Ship detection



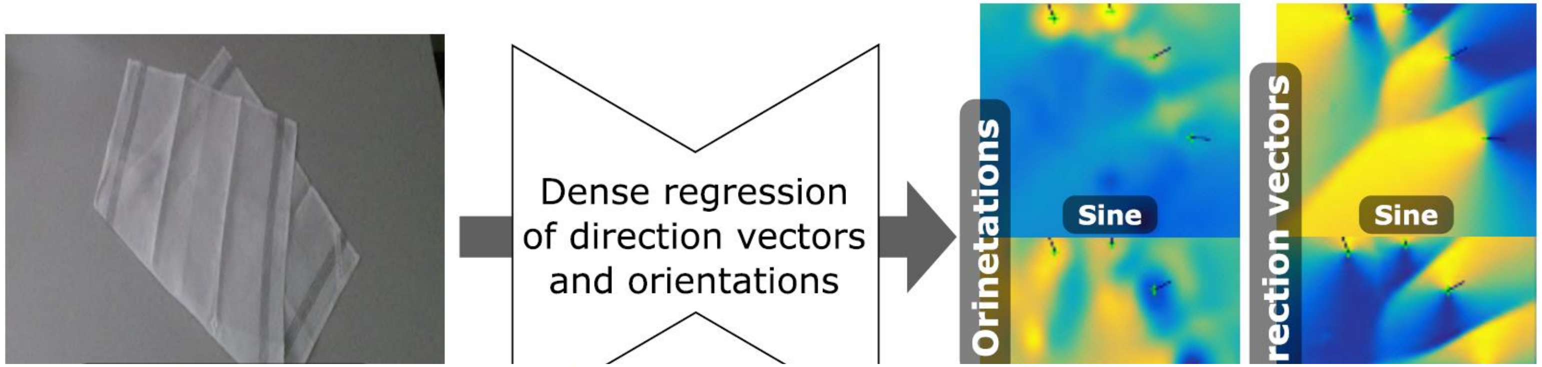
Face detection

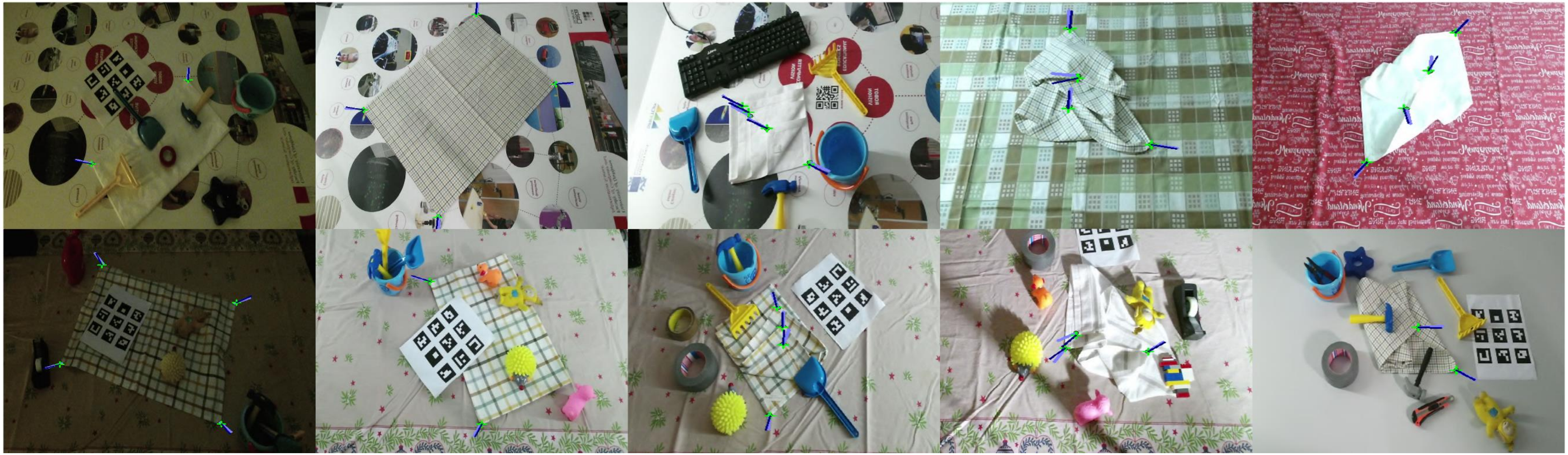
Mask-wearing detection



Grasping Point Localization on Cloths

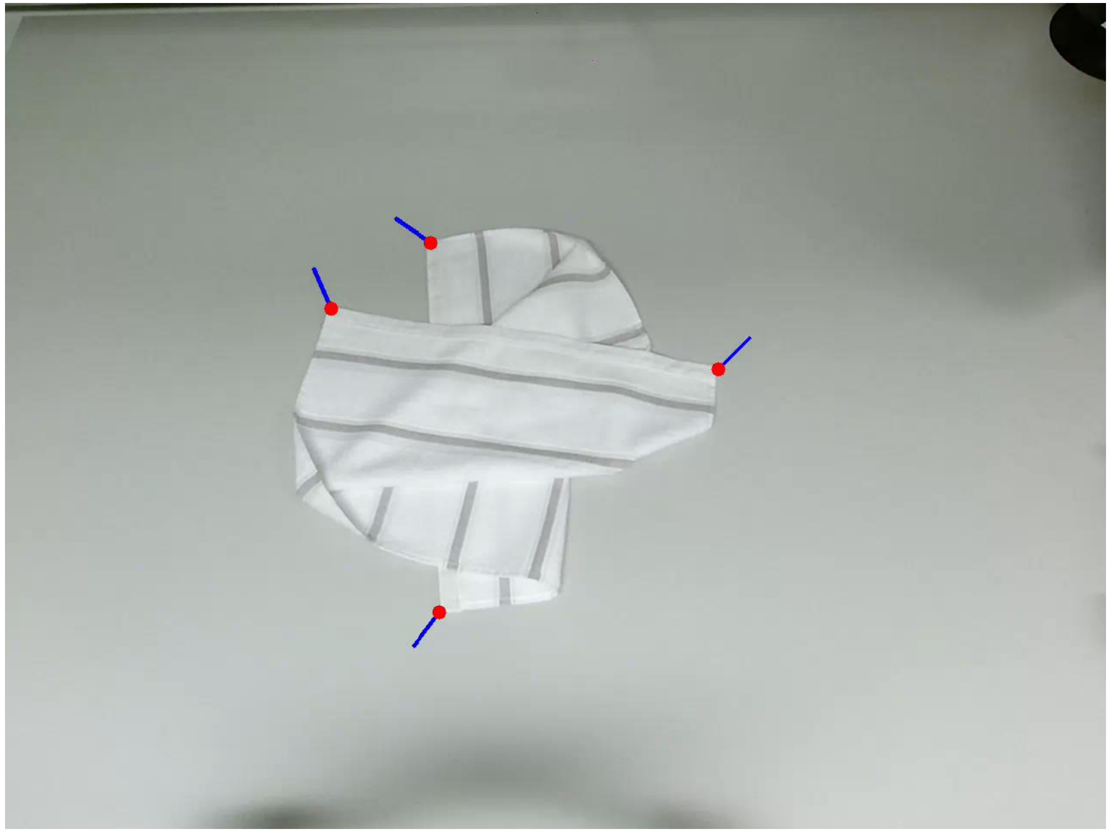
Tabernik et. al, 2024





3DOF object localisation

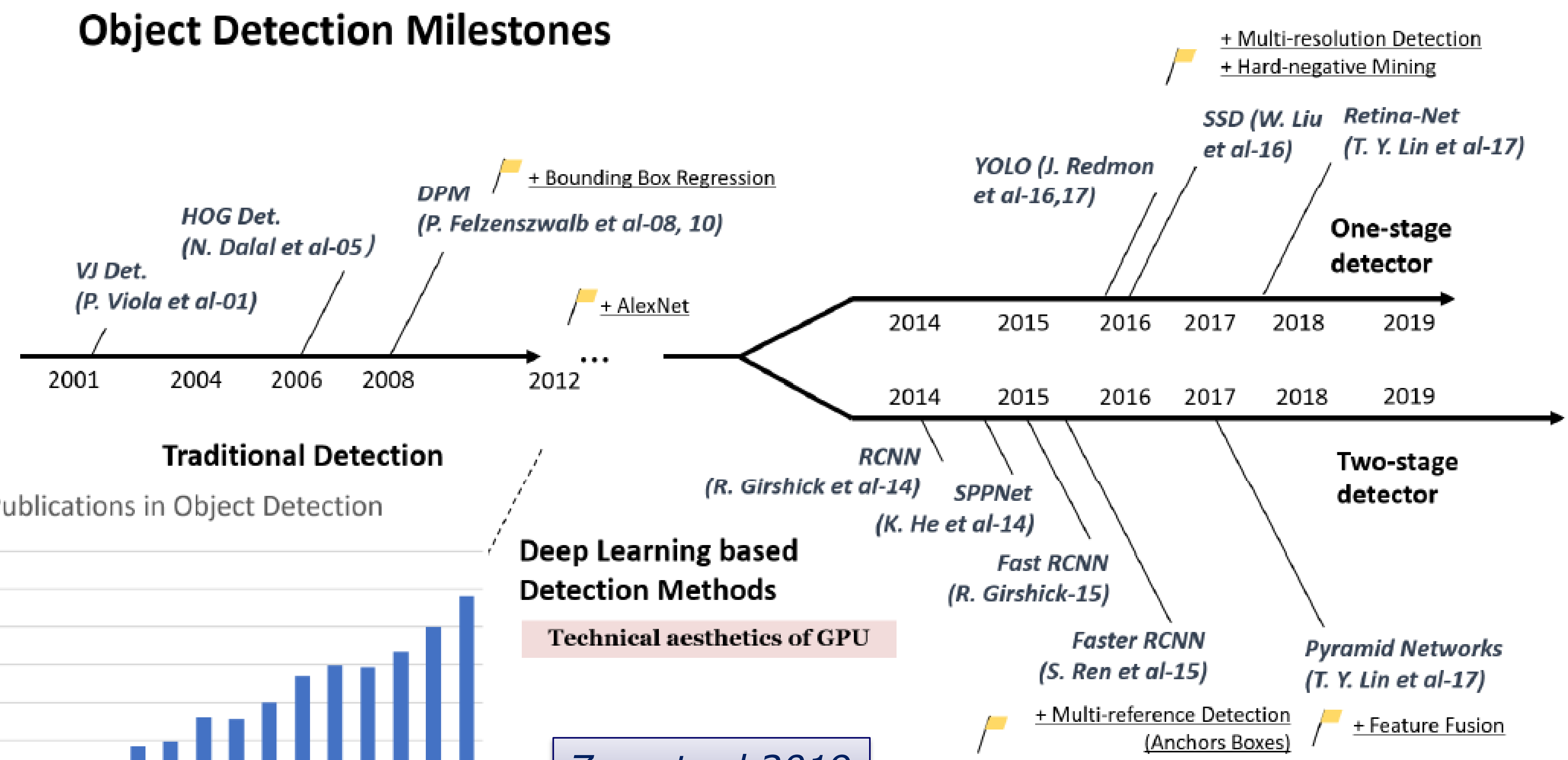
- Detection of grasping points



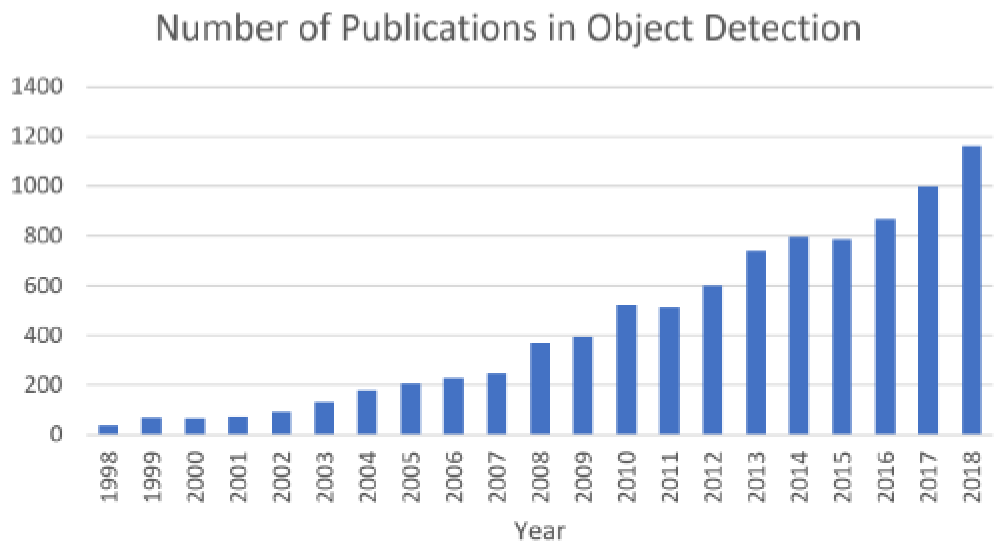
Tabernik et. al, 2024

Tabernik et. al, 2023

Object detection overview



Technical aesthetics of GPU



Zou et. al 2019

[Zou et al, "Object Detection in 20 Years: A Survey", 2019]