

# Machine Learning for Computer Vision

Last update: 11 November 2024

Academic Year 2024 – 2025  
Alma Mater Studiorum · University of Bologna

# Contents

|          |                                                         |           |
|----------|---------------------------------------------------------|-----------|
| <b>1</b> | <b>Optimizers</b>                                       | <b>1</b>  |
| 1.1      | Stochastic gradient descent with mini-batches . . . . . | 1         |
| 1.2      | Second-order methods . . . . .                          | 2         |
| 1.3      | Momentum . . . . .                                      | 3         |
| 1.4      | Adaptive learning rates methods . . . . .               | 4         |
| 1.4.1    | AdaGrad . . . . .                                       | 5         |
| 1.4.2    | RMSProp . . . . .                                       | 5         |
| 1.4.3    | Adam . . . . .                                          | 6         |
| 1.4.4    | AdamW . . . . .                                         | 7         |
| <b>2</b> | <b>Architectures</b>                                    | <b>9</b>  |
| 2.1      | Inception-v1 (GoogLeNet) <sup>1</sup> . . . . .         | 9         |
| 2.2      | Residual networks <sup>2</sup> . . . . .                | 10        |
| 2.2.1    | ResNet . . . . .                                        | 10        |
| 2.2.2    | Inception-ResNet-v4 . . . . .                           | 11        |
| 2.3      | ResNeXt . . . . .                                       | 11        |
| 2.4      | Squeeze-and-excitation network (SENet) . . . . .        | 14        |
| 2.5      | MobileNetV2 . . . . .                                   | 15        |
| 2.6      | Model scaling . . . . .                                 | 16        |
| 2.6.1    | Wide ResNet . . . . .                                   | 18        |
| 2.7      | EfficientNet . . . . .                                  | 18        |
| 2.8      | RegNet . . . . .                                        | 19        |
| <b>3</b> | <b>Transformers in computer vision</b>                  | <b>21</b> |
| 3.1      | Transformer . . . . .                                   | 21        |
| 3.1.1    | Attention mechanism . . . . .                           | 21        |
| 3.1.2    | Embeddings . . . . .                                    | 24        |
| 3.1.3    | Encoder . . . . .                                       | 24        |
| 3.1.4    | Decoder . . . . .                                       | 26        |
| 3.1.5    | Positional encoding . . . . .                           | 28        |
| 3.2      | Vision transformer . . . . .                            | 29        |
| <b>4</b> | <b>Object detection</b>                                 | <b>32</b> |
| 4.1      | Metrics . . . . .                                       | 32        |
| 4.2      | Viola-Jones . . . . .                                   | 34        |
| 4.2.1    | Boosting . . . . .                                      | 34        |
| 4.2.2    | Integral images . . . . .                               | 37        |
| 4.2.3    | Cascade . . . . .                                       | 38        |
| 4.2.4    | Non-maximum suppression . . . . .                       | 38        |
| 4.3      | CNN for object detection . . . . .                      | 38        |
| 4.3.1    | Object localization . . . . .                           | 38        |

---

<sup>1</sup>Excerpt from IPCV2

<sup>2</sup>Excerpt from IPCV2

|          |                                                           |           |
|----------|-----------------------------------------------------------|-----------|
| 4.3.2    | Region proposal detectors . . . . .                       | 39        |
| 4.3.3    | Multi-scale detectors . . . . .                           | 44        |
| 4.3.4    | One-stage detectors . . . . .                             | 46        |
| 4.3.5    | Keypoint-based detectors . . . . .                        | 50        |
| 4.3.6    | Transformer-based detectors . . . . .                     | 52        |
| 4.A      | Appendix: EfficientDet . . . . .                          | 54        |
| <b>5</b> | <b>Segmentation</b>                                       | <b>56</b> |
| 5.1      | Semantic segmentation . . . . .                           | 56        |
| 5.1.1    | Kinect human pose estimation . . . . .                    | 56        |
| 5.1.2    | R-CNN . . . . .                                           | 58        |
| 5.1.3    | Fully convolutional network . . . . .                     | 59        |
| 5.1.4    | U-Net . . . . .                                           | 60        |
| 5.1.5    | Dilated CNN . . . . .                                     | 62        |
| 5.2      | Instance segmentation . . . . .                           | 64        |
| 5.2.1    | Mask R-CNN . . . . .                                      | 64        |
| 5.3      | Panoptic segmentation . . . . .                           | 66        |
| 5.3.1    | Panoptic feature pyramid network . . . . .                | 67        |
| 5.3.2    | MaskFormer . . . . .                                      | 67        |
| 5.3.3    | Masked-attention mask transformer (Mask2Former) . . . . . | 68        |
| 5.A      | Appendix: Spatial pyramid pooling layer . . . . .         | 69        |
| 5.A.1    | DeepLab v2 ASPP . . . . .                                 | 69        |
| 5.A.2    | DeepLab v3 ASPP . . . . .                                 | 70        |
| <b>6</b> | <b>Depth estimation</b>                                   | <b>71</b> |
| 6.1      | Monocular depth estimation . . . . .                      | 71        |
| 6.1.1    | Monodepth . . . . .                                       | 71        |

# 1 Optimizers

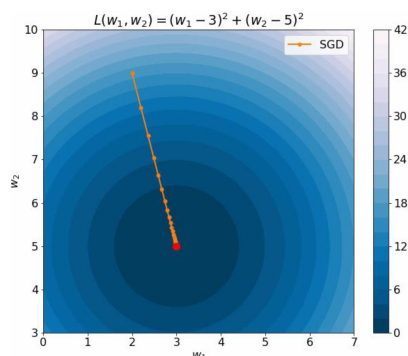
## 1.1 Stochastic gradient descent with mini-batches

**Stochastic gradient descent (SGD)** Gradient descent based on a noisy approximation of the gradient computed on mini-batches of  $B$  data samples. SGD

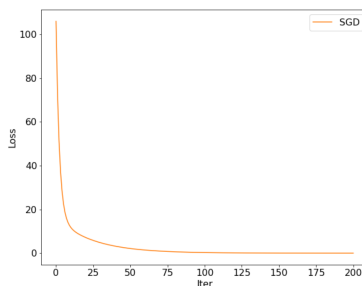
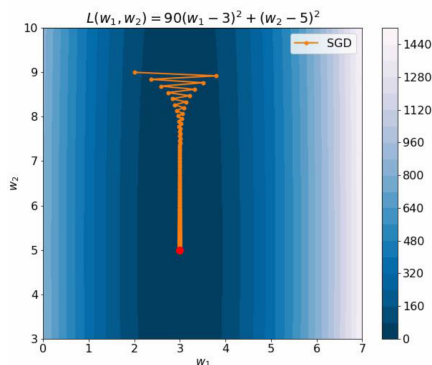
An epoch  $e$  of SGD with mini-batches of size  $B$  does the following:

1. Shuffle the training data  $\mathbf{D}^{\text{train}}$ .
2. For  $u = 0, \dots, U$ , with  $U = \lceil \frac{N}{B} \rceil$ :
  - a) Classify the examples  $\mathbf{X}^{(u)} = \{\mathbf{x}^{(Bu)}, \dots, \mathbf{x}^{(B(u+1)-1)}\}$  to obtain the predictions  $\hat{Y}^{(u)} = f(\mathbf{X}^{(u)}; \boldsymbol{\theta}^{(e*U+u)})$  and the loss  $\mathcal{L}(\boldsymbol{\theta}^{(e*U+u)}, (\mathbf{X}^{(u)}, \hat{Y}^{(u)}))$ .
  - b) Compute the gradient  $\nabla \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}^{(e*U+u)}, (\mathbf{X}^{(u)}, \hat{Y}^{(u)}))$ .
  - c) Update the parameters  $\boldsymbol{\theta}^{(e*U+u+1)} = \boldsymbol{\theta}^{(e*U+u)} - \text{lr} \cdot \nabla \mathcal{L}$ .

**Remark (Spheres).** GD/SGD works better on convex functions (e.g., paraboloids) as there are no preferred directions to reach a minimum. Moreover, faster convergence can be obtained by using a higher learning rate. SGD spheres

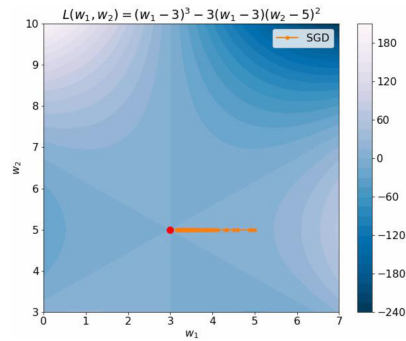


**Remark (Canyons).** A function has a canyon shape if it grows faster in some directions. The trajectory of SGD oscillates in a canyon (the steep area) and a smaller learning rate is required to reach convergence. Note that, even though there are oscillations, the loss alone decreases and is unable to show the oscillating behavior. SGD canyons





**Remark** (Local minima). GD/SGD converges to a critical point. Therefore, it might end up in a saddle point or local minima. SGD local minima



## 1.2 Second-order methods

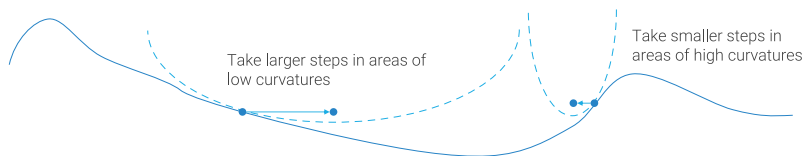
Methods that also consider the second-order derivatives when determining the step.

**Newton's method** Second-order method for the 1D case based on the Taylor expansion: Newton's method

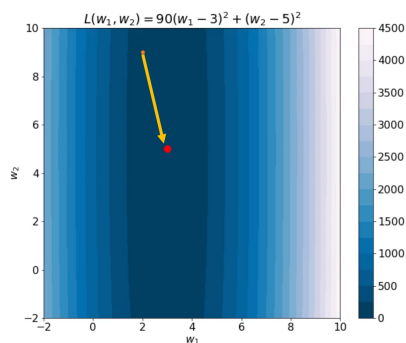
$$f(x_t + \Delta x) \approx f(x_t) + f'(x_t)\Delta x + \frac{1}{2}f''(x_t)\Delta x^2$$

which can be seen as a paraboloid over the variable  $\Delta x$ .

Given a function  $f$  and a point  $x_t$ , the method fits a paraboloid at  $x_t$  with the same slope and curvature at  $f(x_t)$ . The update is determined as the step required to reach the minimum of the paraboloid from  $x_t$ . It can be shown that this step is  $-\frac{f'(x_t)}{f''(x_t)}$ .



**Remark.** For quadratic functions, second-order methods converge in one step.



**General second-order method** For a generic multivariate non-quadratic function, the update is: General second-order method

$$-\mathbf{r} \cdot \mathbf{H}_f^{-1}(\mathbf{x}_t) \nabla f(\mathbf{x}_t)$$

where  $\mathbf{H}_f$  is the Hessian matrix.

**Remark.** Given  $k$  variables,  $\mathbf{H}$  requires  $O(k^2)$  memory. Moreover, inverting a matrix has time complexity  $O(k^3)$ . Therefore, in practice second-order methods are not applicable for large models.

## 1.3 Momentum

**Standard momentum** Add a velocity term  $v^{(t)}$  to account for past gradient updates:

Standard momentum

$$\begin{aligned} v^{(t+1)} &= \mu v^{(t)} - \text{lr} \nabla \mathcal{L}(\theta^{(t)}) \\ \theta^{(t+1)} &= \theta^{(t)} + v^{(t+1)} \end{aligned}$$

where  $\mu \in [0, 1[$  is the momentum coefficient.

In other words,  $v^{(t+1)}$  represents a weighted average of the updates steps done up until time  $t$ .

**Remark.** Momentum helps to counteract a poor conditioning of the Hessian matrix when working with canyons.

**Remark.** Momentum helps to reduce the effect of variance of the approximated gradients (i.e., acts as a low-pass filter).

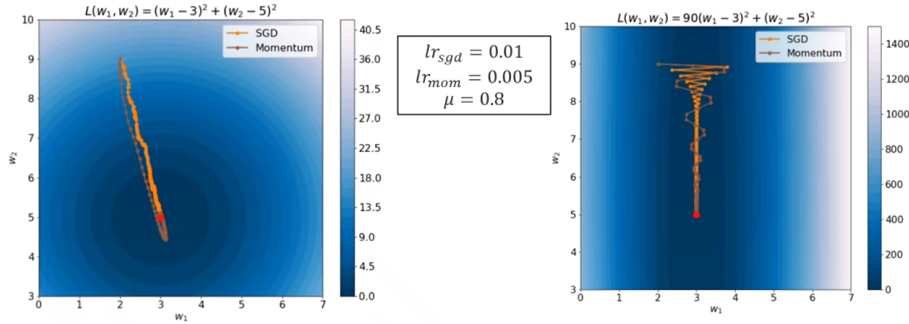


Figure 1.1: Plain SGD vs momentum SGD in a sphere and a canyon. In both cases, momentum converges before SGD.

**Nesterov momentum** Variation of the standard momentum that computes the gradient step considering the velocity term:

Nesterov momentum

$$\begin{aligned} v^{(t+1)} &= \mu v^{(t)} - \text{lr} \nabla \mathcal{L}(\theta^{(t)} + \mu v^{(t)}) \\ \theta^{(t+1)} &= \theta^{(t)} + v^{(t+1)} \end{aligned}$$

**Remark.** The key idea is that, once  $\mu v^{(t)}$  is summed to  $\theta^{(t)}$ , the gradient computed at  $\theta^{(t)}$  is obsolete as  $\theta^{(t)}$  has been partially updated.

**Remark.** In practice, there are methods to formulate Nesterov momentum without the need of computing the gradient at  $\theta^{(t)} + \mu v^{(t)}$ .

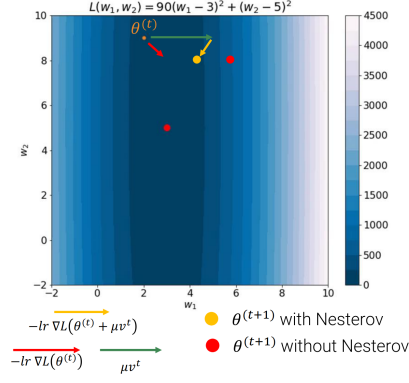


Figure 1.2: Visualization of the step in Nesterov momentum

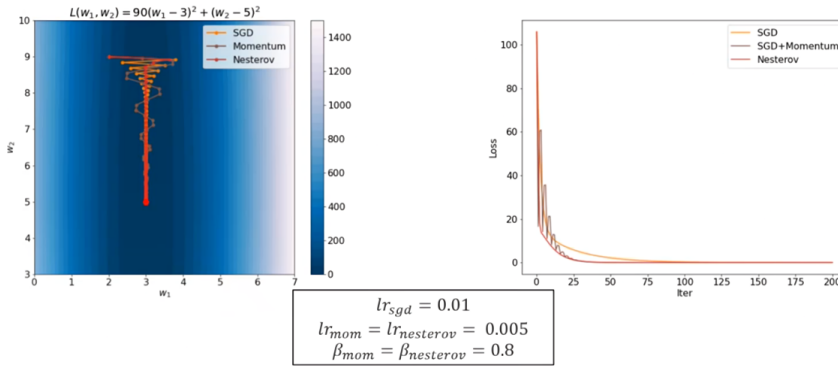


Figure 1.3: Plain SGD vs standard momentum vs Nesterov momentum

## 1.4 Adaptive learning rates methods

**Adaptive learning rates** Methods to define per-parameter adaptive learning rates.

Adaptive learning rates

Ideally, assuming that the changes in the curvature of the loss are axis-aligned (i.e., the parameters are independent), it is reasonable to obtain a faster convergence by:

- Reducing the learning rate along the dimension where the gradient is large.
- Increasing the learning rate along the dimension where the gradient is small.

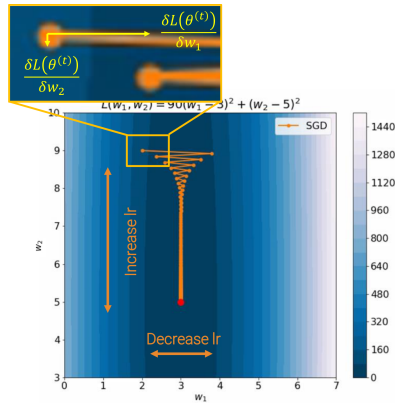


Figure 1.4: Loss where the  $w_1$  parameter has a larger gradient, while  $w_2$  has a smaller gradient

**Remark.** As the landscape of a high-dimensional loss cannot be seen, automatic methods to adjust the learning rates must be used.

### 1.4.1 AdaGrad

**Adaptive gradient (AdaGrad)** Each entry of the gradient is rescaled by the inverse of the history of its squared values:

Adaptive gradient  
(AdaGrad)

$$\begin{aligned}\mathbb{R}^{N \times 1} \ni \mathbf{s}^{(t+1)} &= \mathbf{s}^{(t)} + \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \odot \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \\ \mathbb{R}^{N \times 1} \ni \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} - \frac{\text{lr}}{\sqrt{\mathbf{s}^{(t+1)}} + \varepsilon} \odot \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)})\end{aligned}$$

where:

- $\odot$  is the element-wise product.
- Division and square root are element-wise.
- $\varepsilon$  is a small constant.

**Remark.** By how it is defined,  $\mathbf{s}^{(t)}$  is monotonically increasing which might reduce the learning rate too early when the minimum is still far away.

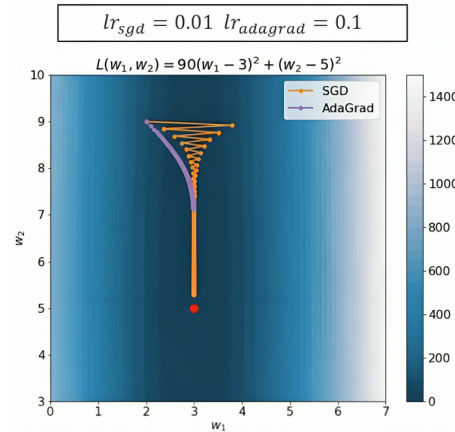


Figure 1.5: SGD vs AdaGrad. AdaGrad stops before getting close to the minimum.

### 1.4.2 RMSProp

**RMSProp** Modified version of AdaGrad that down-weights the gradient history  $\mathbf{s}^{(t)}$ :

RMSProp

$$\begin{aligned}\mathbf{s}^{(t+1)} &= \beta \mathbf{s}^{(t)} + (1 - \beta) \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \odot \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \\ \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} - \frac{\text{lr}}{\sqrt{\mathbf{s}^{(t+1)}} + \varepsilon} \odot \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)})\end{aligned}$$

where  $\beta \in [0, 1]$  (typically 0.9 or higher) makes  $\mathbf{s}^{(t)}$  an exponential moving average.

**Remark.** RMSProp is faster than SGD at the beginning and slows down reaching similar performances as SGD.

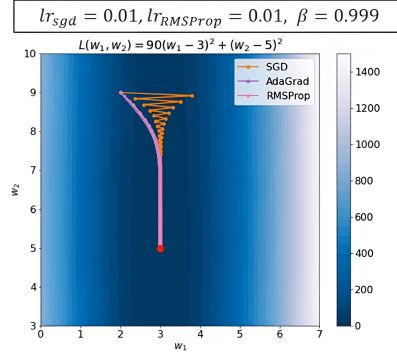


Figure 1.6: SGD vs AdaGrad vs RMSProp

### 1.4.3 Adam

**Adaptive moments (Adam)** Extends RMSProp by also considering a running average for the gradients:

Adaptive moments (Adam)

$$\begin{aligned}\mathbf{g}^{(t+1)} &= \beta_1 \mathbf{g}^{(t)} + (1 - \beta_1) \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \\ \mathbf{s}^{(t+1)} &= \beta_2 \mathbf{s}^{(t)} + (1 - \beta_2) \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \odot \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)})\end{aligned}$$

where  $\beta_1, \beta_2 \in [0, 1]$  (typically  $\beta_1 = 0.9$  and  $\beta_2 = 0.999$ ).

Moreover, as  $\mathbf{g}^{(0)} = 0$ ,  $\mathbf{s}^{(0)} = 0$ , and  $\beta_1, \beta_2$  are typically large (i.e., past history weighs more), Adam starts by taking small steps (e.g.,  $\mathbf{g}^{(1)} = (1 - \beta_1) \nabla \mathcal{L}(\boldsymbol{\theta}^{(0)})$  is simply rescaling the gradient for no reason). To cope with this, a debiased formulation of  $\mathbf{g}$  and  $\mathbf{s}$  is used:

$$\mathbf{g}_{\text{debiased}}^{(t)} = \frac{g^{(t+1)}}{1 - \beta_1^{t+1}} \quad \mathbf{s}_{\text{debiased}}^{(t)} = \frac{s^{(t+1)}}{1 - \beta_2^{t+1}}$$

where the denominator  $(1 - \beta_i^{t+1}) \rightarrow 1$  for increasing values of  $t$ .

Finally, the update is defined as:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\text{lr}}{\sqrt{s_{\text{debiased}}^{(t)} + \varepsilon}} \odot g_{\text{debiased}}^{(t)}$$

**Remark.** It can be shown that  $\frac{g_{\text{debiased}}^{(t)}}{\sqrt{s_{\text{debiased}}^{(t)}}}$  has a bounded domain, making it more controlled than RMSProp.

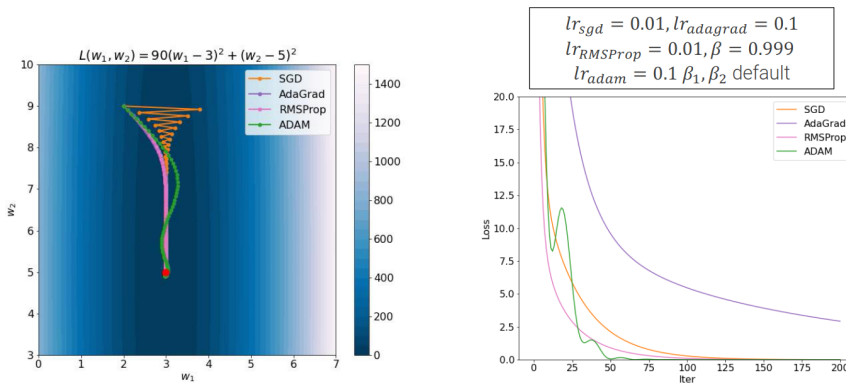


Figure 1.7: SGD vs AdaGrad vs RMSProp vs Adam

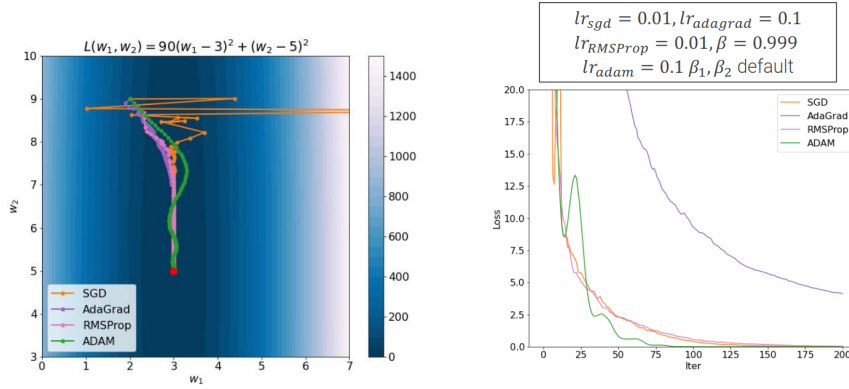
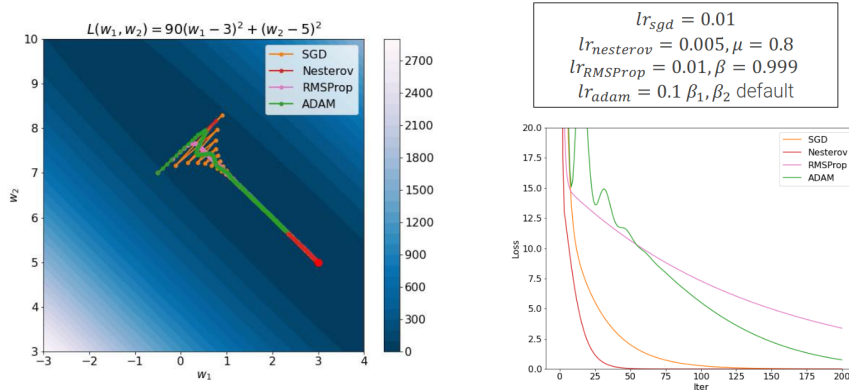


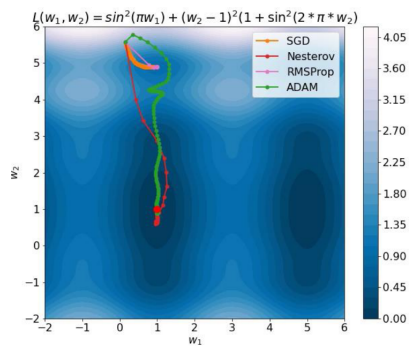
Figure 1.8: SGD vs AdaGrad vs RMSProp vs Adam with a smaller batch size

**Remark.** Adam is based on the assumption of unrelated parameters (i.e., axis-aligned). If this does not actually hold, it might be slower to converge.



**Remark.** Empirically, in computer vision Nesterov momentum (properly tuned) works better than Adam.

**Remark.** Momentum based approaches tend to prefer large basins. Intuitively, by accumulating momentum, it is possible to “escape” small basins.



#### 1.4.4 AdamW

**Adam with weight decay (AdamW)** Modification on the gradient update of Adam to include weight decay:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\text{lr}}{\sqrt{s_{\text{debiased}}^{(t)} + \varepsilon}} \odot g_{\text{debiased}}^{(t)} - \lambda \theta^{(t)}$$

Adam with weight decay (AdamW)

**Remark.** Differently from SGD, L2 regularization on Adam is not equivalent to applying weight decay. In fact, by definition, the regularization term is applied to the gradient and not on the update step:

$$\nabla_{\text{actual}}\mathcal{L}(\boldsymbol{\theta}^{(t)}) = \nabla\mathcal{L}(\boldsymbol{\theta}^{(t)}) + \lambda\boldsymbol{\theta}^{(t)}$$

where  $\nabla_{\text{actual}}\mathcal{L}(\boldsymbol{\theta}^{(t)})$  is the actual gradient used to compute the running averages  $\mathbf{g}$  and  $\mathbf{s}$ .

## 2 Architectures

### 2.1 Inception-v1 (GoogLeNet)<sup>1</sup>

Network that aims to optimize computing resources (i.e., small amount of parameters and FLOPs).

Inception-v1  
(GoogLeNet)

**Stem layers** Down-sample the image from a shape of 224 to 28. As in ZFNet, multiple layers are used (5) and the largest convolution is of shape  $7 \times 7$  with stride 2.

**Inception module** Main component of Inception-v1 that computes multiple convolutions on the input.

Inception module

Given the input activation, the output is the concatenation of:

- A  $1 \times 1$  (stride 1) and a  $5 \times 5$  (stride 1, padding 2) convolution.
- A  $1 \times 1$  (stride 1) and a  $3 \times 3$  (stride 1 and padding 1) convolution.
- A  $1 \times 1$  (stride 1 and padding 0) convolution.
- A  $1 \times 1$  (stride 1) convolution and a  $3 \times 3$  (stride 1 and padding 1) max-pooling.

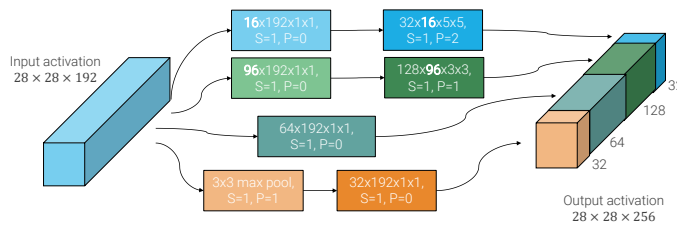


Figure 2.1: Inception module on the output of the stem layers

**Remark.** The multiple convolutions of an inception module can be seen as decision components.

**Auxiliary softmax** Intermediate **softmax**s are used to ensure that hidden features are good enough. They also act as regularizers. During inference, they are discarded.

**Global average pooling classifier** Instead of flattening between the convolutional and fully connected layers, global average pooling is used to reduce the number of parameters.

Global average  
pooling classifier

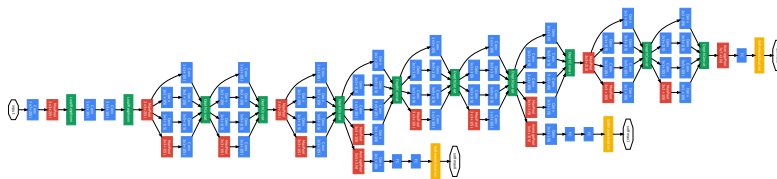


Figure 2.2: Architecture of Inception-v1

<sup>1</sup>Excerpt from IPCV2



## 2.2 Residual networks<sup>2</sup>

**Standard residual block** Block that allows to easily learn the identity function through a skip connection. The output of a residual block with input  $x$  and a series of convolutional layers  $F$  is:

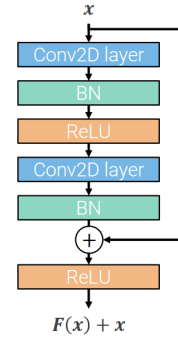
$$F(x; \theta) + x$$

**Skip connection** Connection that skips a certain number of layers (e.g. 2 convolutional blocks).

**Remark.** Training starts with small weights so that the network starts as the identity function. Updates can be seen as perturbations of the identity function.

**Remark.** Batch normalization is heavily used.

**Remark.** Skip connections are applied before the activation function (ReLU) as otherwise it would be summed to all positive values making the perturbation of the identity function less effective.



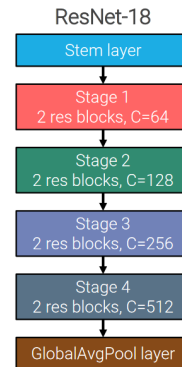
Standard residual block

Skip connection

### 2.2.1 ResNet

VGG-inspired network with residual blocks. It has the following properties:

- A stage is composed of residual blocks.
- A residual block is composed of two  $3 \times 3$  convolutions followed by batch normalization.
- The first residual block of each stage halves the spatial dimension and doubles the number of channels (there is no pooling).



ResNet-18

Figure 2.3: Architecture of ResNet-18

**Bottleneck residual network** Variant of residual blocks that uses more layers with approximately the same number of parameters and FLOPs of the standard residual block. Instead of using two  $3 \times 3$  convolutions, bottleneck residual network has the following structure:

- $1 \times 1$  convolution to compress the channels of the input by an order of 4 (and the spatial dimension by 2 if it is the first block of a stage, as in normal ResNet).
- $3 \times 3$  convolution.
- $1 \times 1$  convolution to match the shape of the skip connection.

Bottleneck residual network

<sup>2</sup>Excerpt from IPCV2

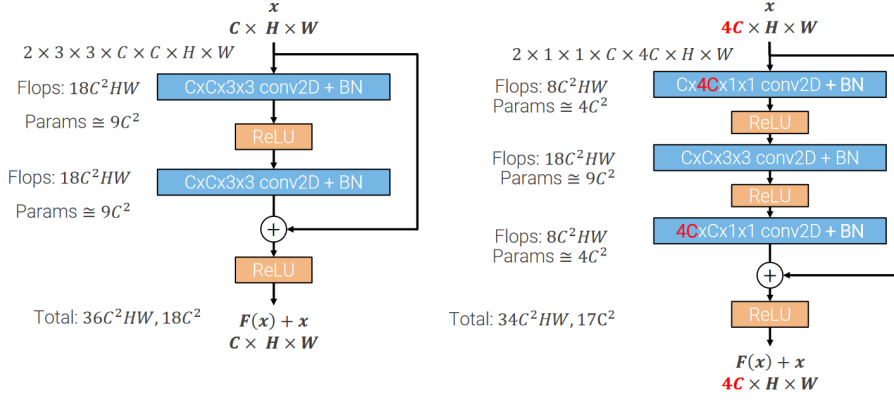


Figure 2.4: Standard residual block (left) and bottleneck block (right)

## 2.2.2 Inception-ResNet-v4

Network with bottleneck-block-inspired inception modules.

**Inception-ResNet-A** Three  $1 \times 1$  convolutions are used to compress the input channels. Inception-ResNet-A Each of them leads to a different path:

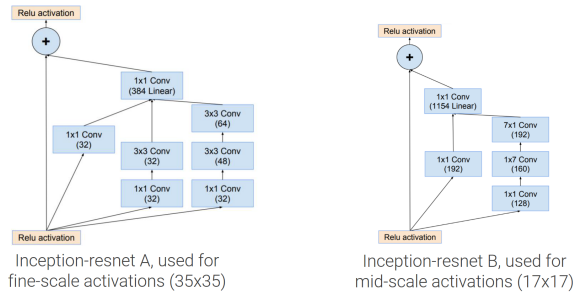
- Directly to the final concatenation.
- To a  $3 \times 3$  convolution.
- To two  $3 \times 3$  convolutions (i.e. a factorized  $5 \times 5$  convolution).

The final concatenation is passed through a  $1 \times 1$  convolution to match the skip connection shape.

**Inception-ResNet-B** Three  $1 \times 1$  convolutions are used to compress the input channels. Inception-ResNet-B Each of them leads to:

- Directly to the final concatenation.
- A  $1 \times 7$  and  $7 \times 1$  convolutions (i.e. a factorized  $7 \times 7$  convolution).

The final concatenation is passed through a  $1 \times 1$  convolution to match the skip connection shape.



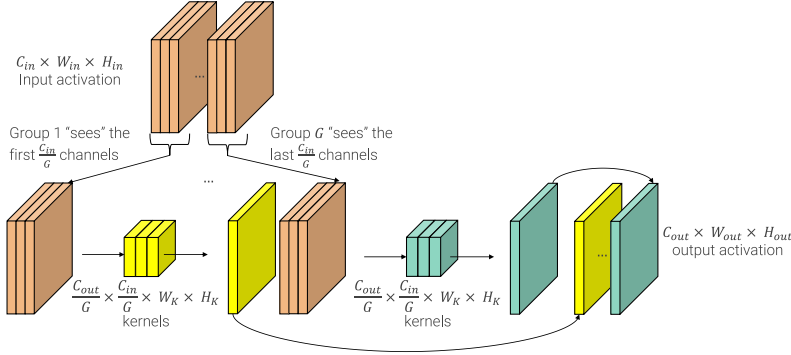
## 2.3 ResNeXt

**Remark.** Inception and Inception-ResNet modules are multi-branch architectures and can be interpreted as a split-transform-merge paradigm. Moreover, their architectures have been specifically “hand” designed.

**Grouped convolution** Given:

- The input activation of shape  $C_{in} \times W_{in} \times H_{in}$ ,
- The desired number of output channels  $C_{out}$ ,
- The number of groups  $G$ ,

a grouped convolution splits the input into  $G$  chunks of  $\frac{C_{in}}{G}$  channels and processes each with a dedicated set of kernels of shape  $\frac{C_{out}}{G} \times \frac{C_{in}}{G} \times W_K \times H_K$ . The output activation is obtained by stacking the outputs of each group.



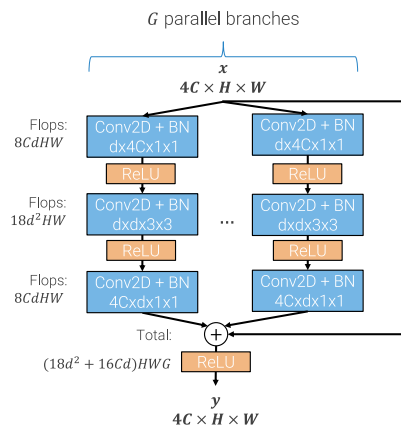
By processing the input in smaller chunks, there are the following gains:

- The number of parameters is  $G$  times less.
- The number of FLOPs is  $G$  times less.

**Remark.** Grouped convolutions are trivially less expressive than convolving on the full input activation. However, as convolutions are expected to build a hierarchy of features, it is reasonable to process the input in chunks as, probably, not all of it is needed.

**ResNetXt block** Given the number of branches  $G$  and the number of intermediate channels  $d$ , a ResNeXt block decomposes a bottleneck residual block into  $G$  parallel branches that are summed out at the end.

ResNetXt block



**Remark.** The branching in a ResNeXt block should not be confused with grouped convolutions.

**Remark.** Parametrizing  $G$  and  $d$  allows obtaining configurations that are FLOP-wise comparable with the original ResNet by fixing  $G$  and solving a second-order equation over  $d$ .

**Equivalent formulation** Given an input activation  $\mathbf{x}$  of shape  $4C \times H \times W$ , each layer of the ResNeXt block can be reformulated as follows:

**Second  $1 \times 1$  convolution** Without loss of generality, consider a ResNeXt block with  $G = 2$  branches.

The output  $\mathbf{y}_k$  at each channel  $k = 1, \dots, 4C$  is obtained as:

$$\mathbf{y}_k = \mathbf{y}_k^{(1)} + \mathbf{y}_k^{(2)} + \mathbf{x}_k$$

where the output  $\mathbf{y}_k^{(b)}$  of a branch  $b$  is computed as:

$$\begin{aligned} \mathbf{y}_k^{(b)}(j, i) &= \left[ \mathbf{w}_k^{(b)} * \mathbf{a}^{(b)} \right]_k(j, i) \\ &= \mathbf{w}_k^{(b)} \cdot \mathbf{a}^{(b)}(j, i) \\ &= \mathbf{w}_k^{(b)}(1) \mathbf{a}^{(b)}(j, i, 1) + \dots + \mathbf{w}_k^{(b)}(d) \mathbf{a}^{(b)}(j, i, d) \end{aligned}$$

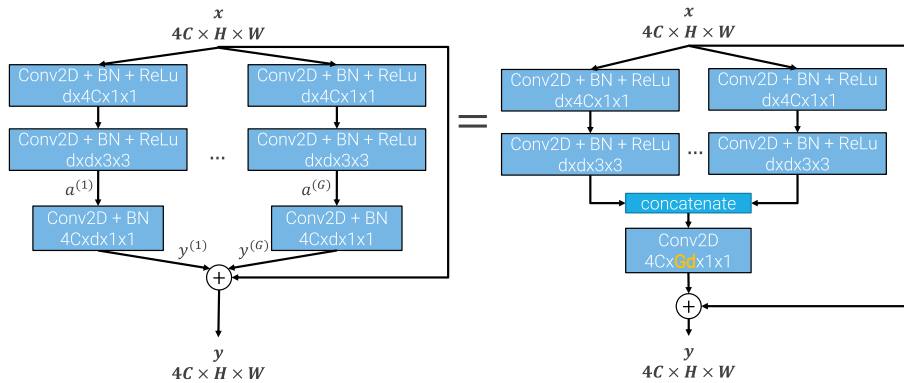
where:

- $*$  represents a convolution,
- $\mathbf{a}^{(b)}$  is the input activation with  $d$  channels from the previous layer.
- $\mathbf{w}^{(b)}$  is the convolutional kernel.  $\mathbf{w}_k^{(b)} \in \mathbb{R}^d$  is the kernel used to obtain the  $k$ -th output channel.

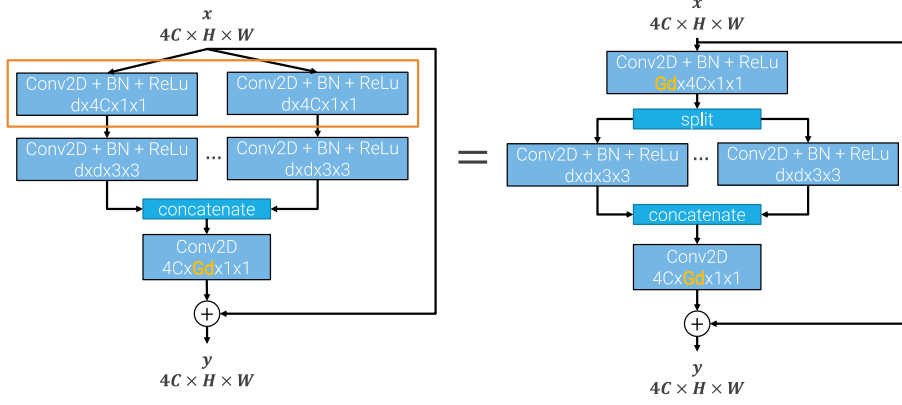
By putting everything together:

$$\begin{aligned} \mathbf{y}_k(j, i) &= \mathbf{w}_k^{(1)} \cdot \mathbf{a}^{(1)}(j, i) + \mathbf{w}_k^{(2)} \cdot \mathbf{a}^{(2)}(j, i) + \mathbf{x}_k \\ &= \underbrace{\left[ \mathbf{w}_k^{(1)} \mathbf{w}_k^{(2)} \right]}_{\text{by stacking, this is a } 1 \times 1 \text{ convolution with } 2d \text{ channels}} \cdot \underbrace{\left[ \mathbf{a}^{(1)}(j, i) \mathbf{a}^{(2)}(j, i) \right]}_{\text{by stacking depth-wise, this is an activation with } 2d \text{ channels}} + \mathbf{x}_k \end{aligned}$$

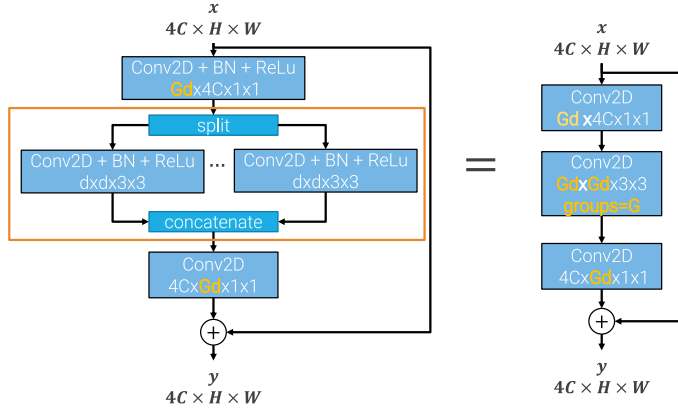
Therefore, the last ResNeXt layer with  $G$  branches is equivalent to a single convolution with  $Gd$  input channels that processes the concatenation of the activations of the previous layer.



**First  $1 \times 1$  convolution** The  $G$   $1 \times 1$  convolutions at the first layer of ResNeXt all process the same input  $\mathbf{x}$ . Trivially, this can also be represented using a single  $1 \times 1$  convolution with  $G$  times more output channels that can be split afterwards.



**3 × 3 convolution** By putting together the previous two equivalences, the middle layer has the same definition of a grouped convolution with  $G$  groups. Therefore, it can be seen as a single grouped convolution with  $G$  groups and  $Gd$  input and output channels.



**Remark.** Therefore, a ResNeXt block is similar to a bottleneck block.

**Remark.** It has been empirically seen that, with the same FLOPs, it is better to have more groups (i.e., wider activations).

## 2.4 Squeeze-and-excitation network (SENet)

**Squeeze-and-excitation module** Block that weighs the channels of the input activation. Given the  $c$ -th channel of the input activation  $\mathbf{x}_c$ , the output  $\tilde{\mathbf{x}}_c$  is computed as:

Squeeze-and-excitation module

$$\tilde{\mathbf{x}}_c = s_c \mathbf{x}_c$$

where  $s_c \in [0, 1]$  is the scaling factor.

The two operations of a squeeze-and-excitation block are:

**Squeeze** Global average pooling to obtain a channel-wise vector.

**Excitation** Feed-forward network that first compresses the input channels by a ratio  $r$  (typically 16) and then restores them.

**Squeeze-and-excitation network (SENet)** Deep ResNet/ResNeXt with squeeze-and-excitation modules.

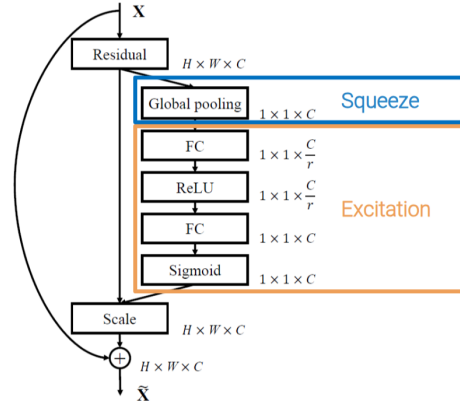


Figure 2.5: SE-ResNet module

## 2.5 MobileNetV2

**Depth-wise separable convolution** Use grouped convolutions to reduce the computational cost of standard convolutions. The operations of filtering and combining features are split:

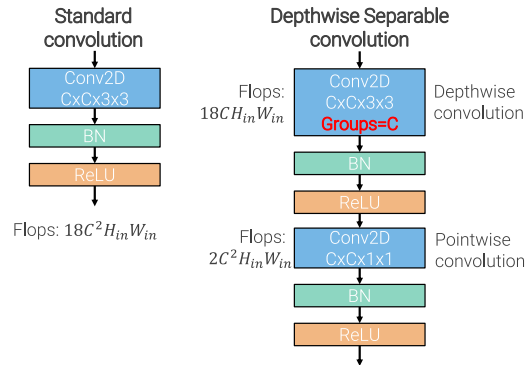
Depth-wise  
separable  
convolution

**Depth-wise convolution** Processes each channel in isolation. In other words, a grouped convolution with groups equal to the number of input channels is applied.

**Context point-wise convolution**  $1 \times 1$  convolution applied after the depth-wise convolution to reproduce the channel-wide effect of standard convolutions.

**Remark.** The gain in computation is up to 10 times the FLOPs of normal convolutions.

**Remark.** Depth-wise convolutions are less expressive than normal convolutions.



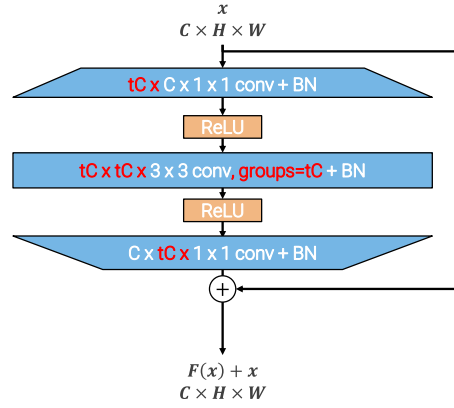
**Remark.** The  $3 \times 3$  convolution in bottleneck residual blocks process a compressed version of the input activation, which might cause loss of information when passing through the ReLUs.

**Inverted residual block** Modified bottleneck block defined as follows:

Inverted residual  
block

1. A  $1 \times 1$  convolution to expand the input channels by a factor of  $t$ .
2. A  $3 \times 3$  depth-wise convolution.
3. A  $1 \times 1$  convolution to compress the channels back to the original shape.

Moreover, non-linearity between residual blocks is removed as a result of theoretical studies.



**MobileNetV2** Stack of inverted residual blocks.

MobileNetV2

- The number of channels grows slower compared to other architectures.
- The stem layer is lightweight due to the low number of intermediate channels.
- Due to the small number of channels, the number of channels in the activation are expanded before passing to the fully-connected layers.

**Remark.** Stride 2 is applied to the middle  $3 \times 3$  convolution when downsampling is needed.

Table 2.1: Architecture of MobileNetV2 with expansion factor ( $t$ ), number of channels ( $c$ ), number of times a block is repeated ( $n$ ), and stride ( $s$ ).

| Input                    | Operator             | $t$ | $c$  | $n$ | $s$ |
|--------------------------|----------------------|-----|------|-----|-----|
| $224 \times 3$           | conv2d               | -   | 32   | 1   | 2   |
| $112 \times 32$          | bottleneck           | 1   | 16   | 1   | 1   |
| $112 \times 16$          | bottleneck           | 6   | 24   | 2   | 2   |
| $56 \times 24$           | bottleneck           | 6   | 32   | 3   | 2   |
| $28 \times 32$           | bottleneck           | 6   | 64   | 4   | 2   |
| $14 \times 64$           | bottleneck           | 6   | 96   | 3   | 1   |
| $14 \times 96$           | bottleneck           | 6   | 160  | 3   | 2   |
| $7 \times 160$           | bottleneck           | 6   | 320  | 1   | 1   |
| $7 \times 320$           | conv2d $1 \times 1$  | -   | 1280 | 1   | 1   |
| $7 \times 1280$          | avgpool $7 \times 7$ | -   | -    | 1   | -   |
| $1 \times 1 \times 1280$ | conv2d $1 \times 1$  | -   | k    | -   | 1   |

## 2.6 Model scaling

**Single dimension scaling** Scaling a baseline model by width, depth, or resolution. It generally always improve the accuracy.

**Width scaling** Increase the number of channels.

Width scaling

**Depth scaling** Increase the number of blocks.

Depth scaling

**Resolution scaling** Increase the spatial dimension of the activations.

Resolution scaling

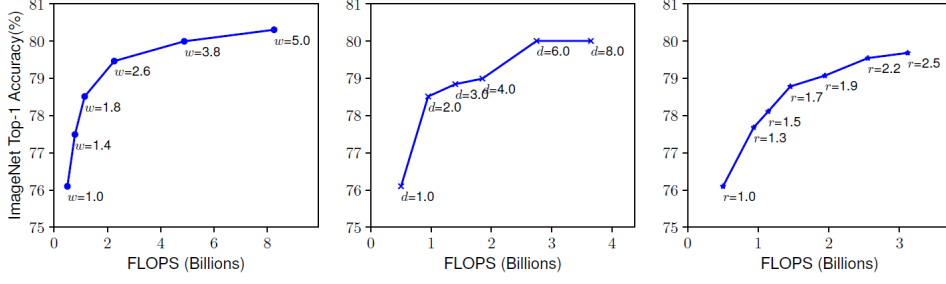


Figure 2.6: Top-1 accuracy variation with width, depth, and resolution scaling on EfficientNet

**Compound scaling** Scaling across multiple dimensions.

Compound scaling

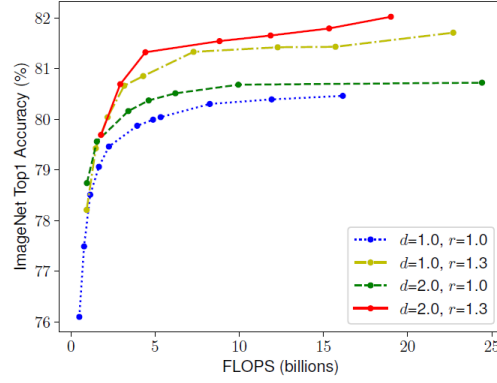


Figure 2.7: Width scaling for different fixed depths and resolutions

**Compound scaling coefficient** Use a compound coefficient  $\phi$  to scale dimensions and systematically control the FLOPs increase.

**Remark.**  $\phi = 0$  represents the baseline model.

The multiplier for depth ( $d$ ), width ( $w$ ), and resolution ( $r$ ) are determined as:

$$d = \alpha^\phi \quad w = \beta^\phi \quad r = \gamma^\phi$$

where  $\alpha$ ,  $\beta$ , and  $\gamma$  are subject to:

$$\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \quad \text{with } \alpha, \beta, \gamma \geq 1$$

By enforcing this constraint, FLOPs will approximately grow by  $2^\phi$  (i.e., double) for each increase of  $\phi$ .

In practice,  $\alpha$ ,  $\beta$ , and  $\gamma$  are determined through grid search.

**Remark.** The constraint is formulated in this way as FLOPs scales linearly with depth but quadratically with width and resolution.



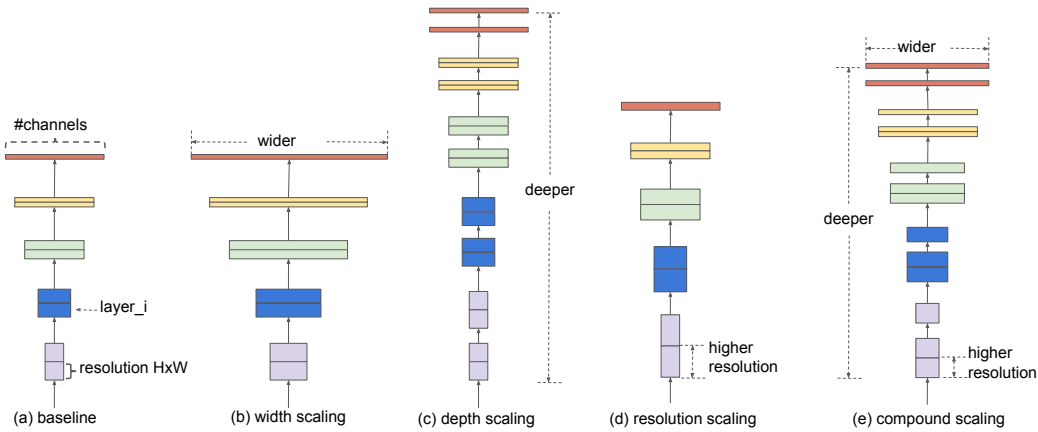
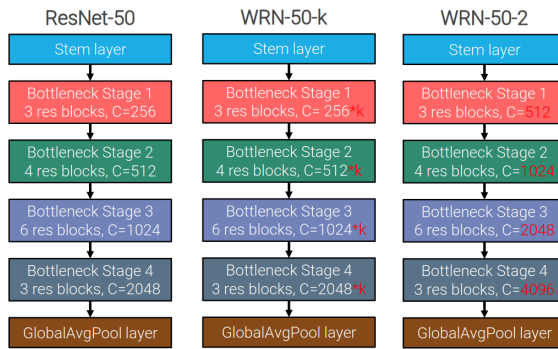


Figure 2.8: Model scaling approaches

## 2.6.1 Wide ResNet

**Wide ResNet (WRN)** ResNet scaled width-wise.

Wide ResNet (WRN)

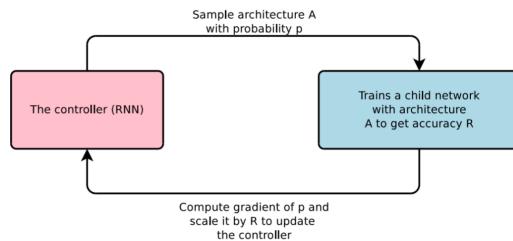


**Remark.** Wider layers are easier to parallelize on GPUs.

## 2.7 EfficientNet

**Neural architecture search (NAS)** Train a controller neural network using gradient policy to output network architectures.

Neural architecture search (NAS)

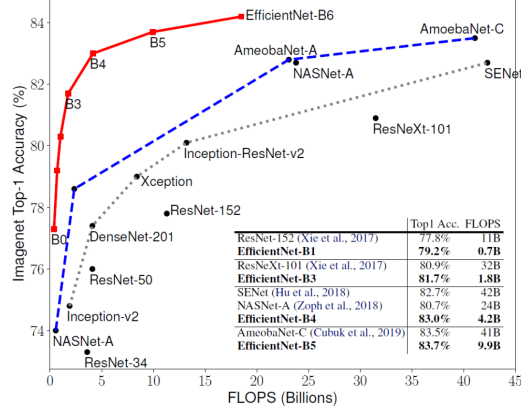


**Remark.** Although effective, we usually cannot extract guiding principles from the architecture outputted by NAS.

**EfficientNet-B0** Architecture obtained through neural architecture search starting from MobileNet.

EfficientNet-B0

Scaling the baseline model (B0) allowed obtaining high accuracies with a controlled number of FLOPs.



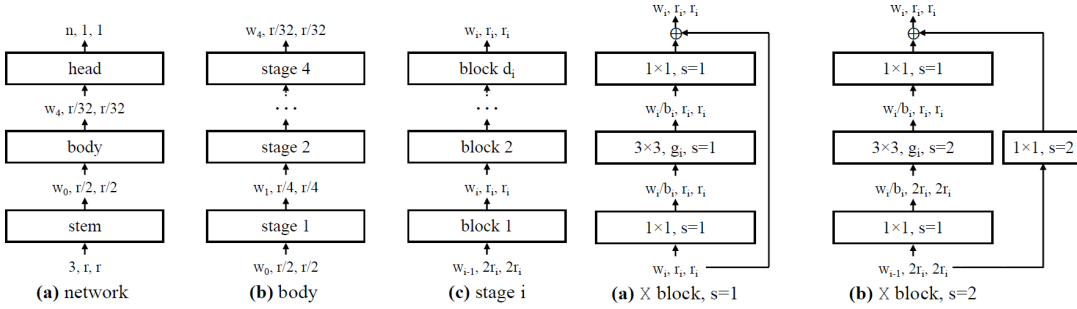
## 2.8 RegNet

**Design space** Space of a parametrized population of neural network architectures. By sampling networks from a design space, it is possible to determine a distribution and evaluate it using statistical tools.

**Remark.** Comparing distributions is more robust than searching for a single well performing architecture (as in NAS).

**RegNet** Classic stem-body-head architecture (similar to ResNeXt with fewer constraints) with four stages. Each stage  $i$  has the following parameters:

- Number of blocks (i.e., depth)  $d_i$ .
- Width of the blocks  $w_i$  (so each stage does not necessarily double the number of channels).
- Number of groups of each block  $g_i$ .
- Bottleneck ratio of each block  $b_i$ .



In other words, RegNet defines a 16-dimensional design space. To evaluate the architectures, the following is done:

1. Sample  $n = 500$  models from the design space and train them on a low-epoch training regime.
2. Determine the error empirical cumulative distribution function  $F$  computed as the fraction of models with an error less than  $e$ :

$$F(e) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}[e_i < e]$$

3. Evaluate the design space by plotting  $F$ .

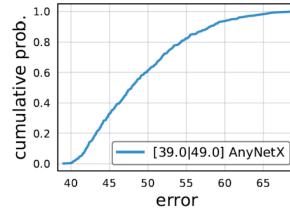


Figure 2.9: Example of cumulative distribution

**Remark.** Similarly to the ROC curve, the plot of the perfect design space is a straight line at 1.0 probability starting from 0% error rate.

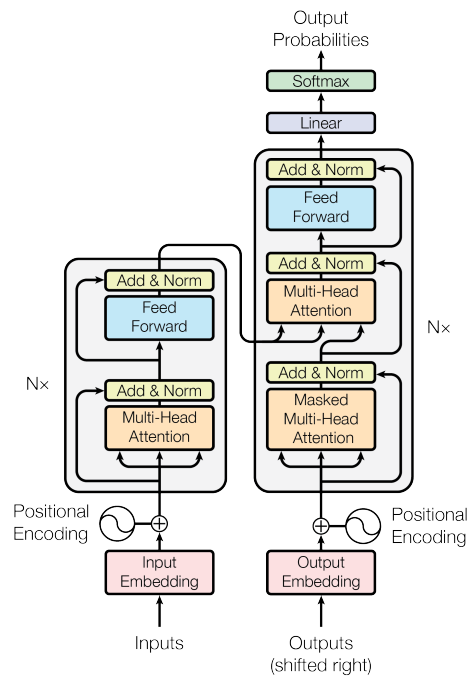
4. Repeat by fixing or finding relationships between parameters (i.e., try to reduce the search space).

**Remark.** In the original paper, RegNet outperformed EfficientNet. However, results were computed by retraining EfficientNet using the same hyperparameter configuration of RegNet, while the original paper of EfficientNet explicitly tuned its hyperparameters to maximize the results.

## 3 Transformers in computer vision

### 3.1 Transformer

**Transformer** Neural architecture designed for NLP sequence-to-sequence tasks. It heavily relies on the attention mechanism. Transformer



**Autoregressive generation** A transformer generates the output sequence progressively given the input sequence and the past outputted tokens. At the beginning, the first token provided as the past output is a special start-of-sequence token ( $\langle \text{SoS} \rangle$ ). Generation is terminated when a special end-of-sequence token ( $\langle \text{EoS} \rangle$ ) is generated.

Autoregressive generation

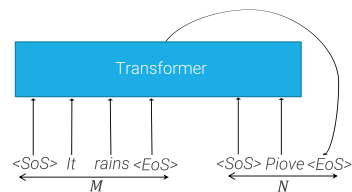


Figure 3.1: Example of autoregressive generation

#### 3.1.1 Attention mechanism

**Traditional attention** Matrix computed by a neural network to weigh each token of a sequence against the tokens of another one.

Traditional attention

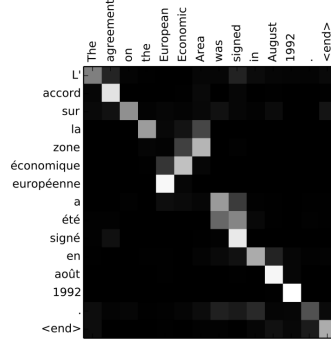


Figure 3.2: Attention weights for machine translation

**Remark.** Before attention, for tasks such as machine translation, the whole input sequence was mapped into an embedding that is used to influence the generation of the output.

**Remark.** This is not the same attention of transformers as they do not directly compute attention weights between inputs and outputs.

**Dot-product attention** Given  $M$  input tokens  $\mathbf{Y} \in \mathbb{R}^{M \times d_Y}$  and a vector  $\mathbf{x}_1 \in \mathbb{R}^{d_Y}$ , dot-product attention computes a linear combination of  $\mathbf{Y}$  where each component is weighted based on a similarity score between  $\mathbf{Y}$  and  $\mathbf{x}_1$ .

Dot-product attention

This is done as follows:

1. Determine the similarity scores of the inputs as the dot-product between  $\mathbf{x}_1$  and  $\mathbf{Y}^T$ :

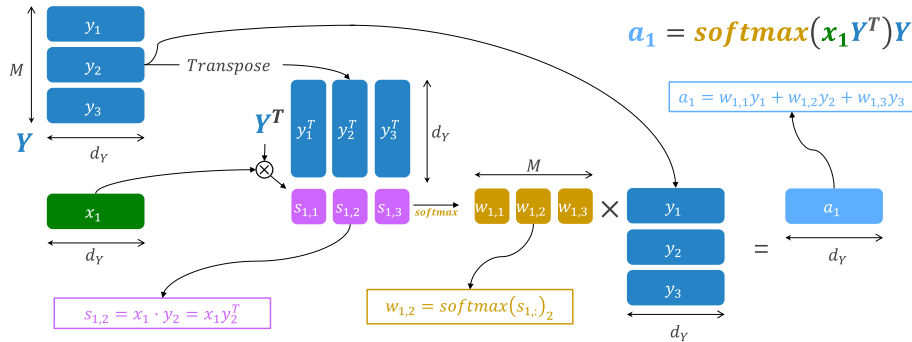
$$\mathbf{x}_1 \mathbf{Y}^T \in \mathbb{R}^M$$

2. Compute the attention weights by applying the **softmax** function on the similarity scores:

$$\text{softmax}(\mathbf{x}_1 \mathbf{Y}^T) \in \mathbb{R}^M$$

3. Determine the output activation  $\mathbf{a}_1$  as the dot-product between the attention weights and the input  $\mathbf{Y}$ :

$$\mathbb{R}^{d_Y} \ni \mathbf{a}_1 = \text{softmax}(\mathbf{x}_1 \mathbf{Y}^T) \mathbf{Y}$$



**Scaled dot-product attention** To add more flexibility, a linear transformation can be applied on the inputs  $\mathbf{Y}$  and  $\mathbf{x}_1$  to obtain:

Scaled dot-product attention

**Keys** With the projection  $\mathbf{W}_K \in \mathbb{R}^{d_Y \times d_K}$  such that  $\mathbb{R}^{M \times d_K} \ni \mathbf{K} = \mathbf{Y}\mathbf{W}_K$ , where  $d_K$  is the dimension of the keys.

**Query** With the projection  $\mathbf{W}_Q \in \mathbb{R}^{d_X \times d_K}$  such that  $\mathbb{R}^{d_K} \ni \mathbf{q}_1 = \mathbf{Y}\mathbf{W}_X$ , where  $d_X$  is the length of  $\mathbf{x}_1$  that is no longer required to be  $d_Y$  as there is a projection.

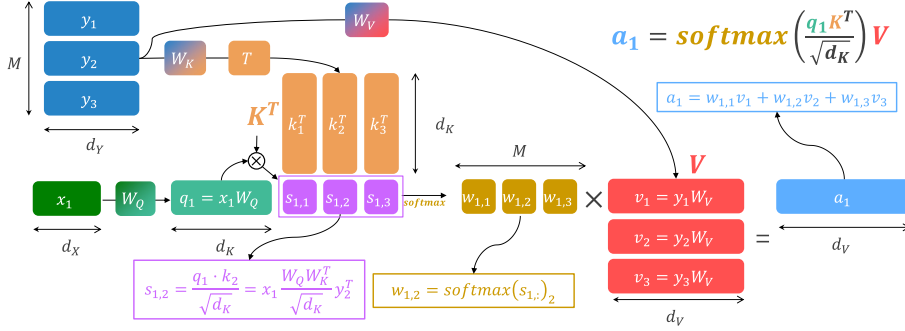
**Values** With the projection  $\mathbf{W}_V \in \mathbb{R}^{d_Y \times d_V}$  such that  $\mathbb{R}^{M \times d_V} \ni \mathbf{V} = \mathbf{Y}\mathbf{W}_V$ , where  $d_V$  is the dimension of the values.

The attention mechanism is then defined as:

$$\mathbf{a}_1 = \text{softmax}(\mathbf{q}_1 \mathbf{K}^T) \mathbf{V}$$

To obtain smoother attention weights when working with high-dimensional activations (i.e., avoid a one-hot vector from `softmax`), a temperature of  $\sqrt{d_K}$  is applied to the similarity scores:

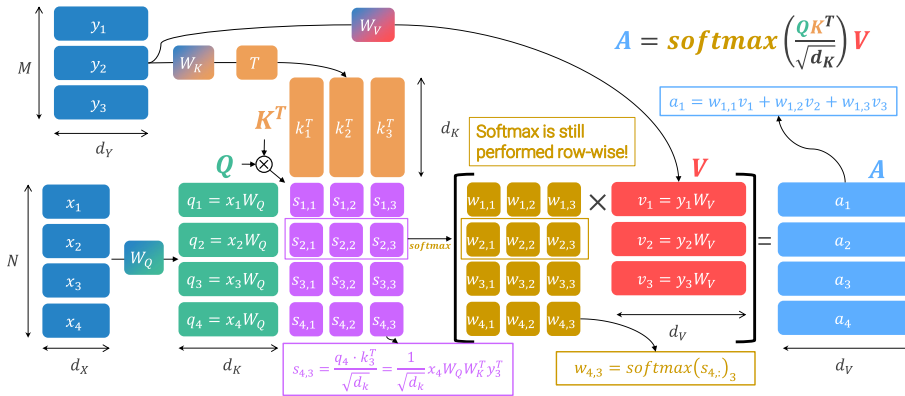
$$\mathbf{a}_1 = \text{softmax}\left(\frac{\mathbf{q}_1 \mathbf{K}^T}{\sqrt{d_K}}\right) \mathbf{V}$$



Finally, due to the linear projections, instead of a single vector there can be an arbitrary number  $N$  of inputs  $\mathbf{X} \in \mathbb{R}^{N \times d_X}$  to compute the queries  $\mathbb{R}^{N \times d_K} \ni \mathbf{Q} = \mathbf{X}\mathbf{W}_Q$ . This change affects the similarity scores  $\mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times M}$  and the output activations  $\mathbf{A} \in \mathbb{R}^{N \times d_V}$ .

The overall attention mechanism can be defined as:

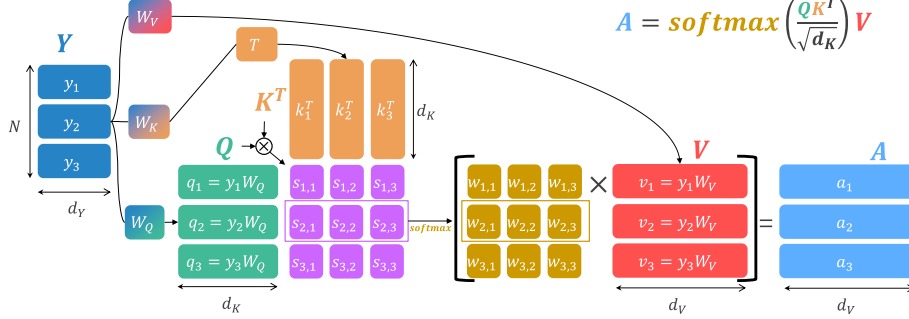
$$\mathbf{A} = \text{softmax}_{\text{row-wise}}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_K}}\right) \mathbf{V}$$



**Self-attention** Scaled dot-product attention mechanism where the inputs to compute keys, queries, and values are the same.

Self-attention

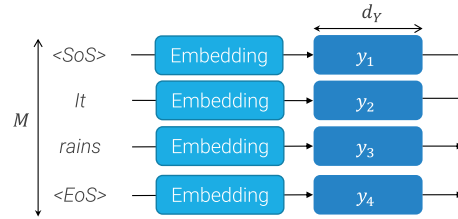
Given an input  $\mathbf{Y} \in \mathbb{R}^{N \times d_Y}$ , the shape of each component is:  $\mathbf{K} \in \mathbb{R}^{N \times d_K}$ ,  $\mathbf{Q} \in \mathbb{R}^{N \times d_K}$ ,  $\mathbf{V} \in \mathbb{R}^{N \times d_V}$ , and  $\mathbf{A} \in \mathbb{R}^{N \times d_V}$ .



### 3.1.2 Embeddings

**Embedding layer** Converts input tokens into their corresponding learned embeddings of shape  $d_Y$  (usually denoted as  $d_{\text{model}}$ ).

Embedding layer



### 3.1.3 Encoder

**Encoder components** A transformer encoder is composed of:

**Multi-head self-attention (MHSA)** Given an input  $\mathbf{Y} \in \mathbb{R}^{M \times d_Y}$ , a MHSA block parallelly passes it through  $h$  different self-attention blocks to obtain the activations  $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(h)}$ . The output  $\mathbf{A}$  of the block is obtained as a linear projection of the column-wise concatenation of the activations  $\mathbf{A}^{(i)}$ :

Multi-head self-attention

$$\mathbb{R}^{M \times d_Y} \ni \mathbf{A} = \left[ \mathbf{A}^{(1)} | \dots | \mathbf{A}^{(h)} \right] \mathbf{W}_O$$

where  $\mathbf{W}_O \in \mathbb{R}^{h d_Y \times d_Y}$  is the projection matrix.

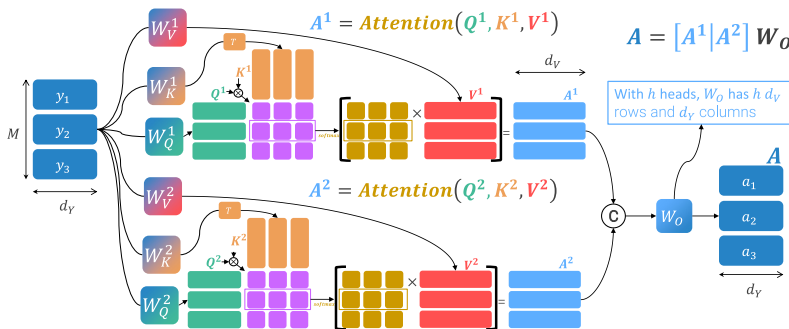


Figure 3.3: MHSA with two heads

**Remark.** The idea of multiple attention heads is to allow the model to attend to information from different representation subspaces.

**Remark.** Even though they can be freely set, the dimensions for queries, keys, and values of each attention head usually are  $d_K = d_V = d_Y/h$ .

**Layer normalization (LN)** Normalize each input activation independently to have zero mean and unit variance, regardless of the other activations in the batch.

Layer normalization

Given  $B$  activations  $\mathbf{a}^{(i)} \in \mathbb{R}^D$ , mean and variance of each activation  $i = 1, \dots, B$  are computed as:

$$\mu^{(i)} = \frac{1}{D} \sum_{j=1}^D \mathbf{a}_j^{(i)} \quad v^{(i)} = \frac{1}{D} \sum_{j=1}^D \left( \mathbf{a}_j^{(i)} - \mu^{(i)} \right)^2$$

Each component  $j$  of the normalized activation  $\hat{\mathbf{a}}^{(i)}$  is computed as:

$$\hat{\mathbf{a}}_j^{(i)} = \frac{\mathbf{a}_j^{(i)} - \mu^{(i)}}{\sqrt{v^{(i)} + \varepsilon}}$$

As in batch normalization, the actual output activation  $\mathbf{s}^{(i)}$  of each input  $\mathbf{a}^{(i)}$  is scaled and offset by learned values:

$$\mathbf{s}_j^{(i)} = \gamma_j \hat{\mathbf{a}}_j^{(i)} + \beta_j$$

**Remark.** Differently from computer vision, in NLP the input is not always of the same length and padding is needed. Therefore, batch normalization do not always work well.

**Remark.** Layer normalization is easier to distribute on multiple computation units and has the same behavior at both train and inference time.

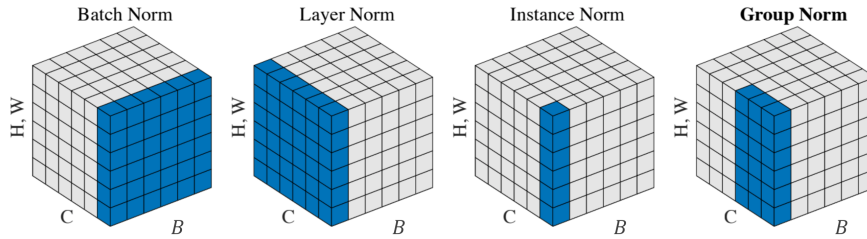


Figure 3.4: Affected axis of normalization methods

**Feed-forward network (FFN)** MLP with one hidden layer applied to each token independently. ReLU or one of its variants are used as activation function:

Feed-forward network

$$\text{FFN}(\mathbf{x}) = \text{relu}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

**Remark.** It can be implemented using two 1D convolutions with kernel size 1.

**Residual connection** Around the MHSA and FFN modules.

Residual connection

**Encoder stack** Composed of  $L$  encoder layers.

Encoder stack



**Encoder layer** Layer to compute a higher level representation of each input token while maintaining the same length of  $d_Y$ .

Encoder layer

Given the input tokens  $\mathbf{H}^{(i)} = [\mathbf{h}_1^{(i)}, \dots, \mathbf{h}_N^{(i)}]$ , depending on the position of layer normalization, an encoder layer computes the following:

**Post-norm transformer** Normalization is done after the residual connection:

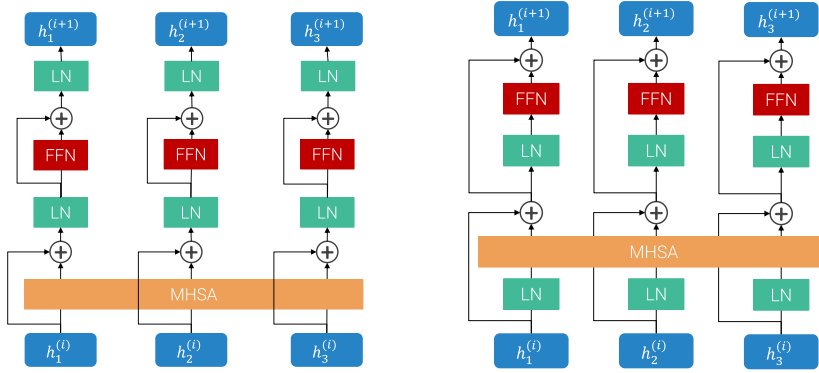
$$\begin{aligned}\bar{\mathbf{h}}_j^{(i)} &= \text{LN} \left( \mathbf{h}_j^{(i)} + \text{MHSA}_{\mathbf{H}^{(i)}}(\mathbf{h}_j^{(i)}) \right) \\ \mathbf{h}_j^{(i+1)} &= \text{LN} \left( \bar{\mathbf{h}}_j^{(i)} + \text{FNN}(\bar{\mathbf{h}}_j^{(i)}) \right)\end{aligned}$$

**Remark.** In post-norm transformers, residual connections are “disrupted” by layer normalization.

**Pre-norm transformer** Normalization is done inside the residual connection:

$$\begin{aligned}\bar{\mathbf{h}}_j^{(i)} &= \mathbf{h}_j^{(i)} + \text{MHSA}_{\mathbf{H}^{(i)}} \left( \text{LN}(\mathbf{h}_j^{(i)}) \right) \\ \mathbf{h}_j^{(i+1)} &= \bar{\mathbf{h}}_j^{(i)} + \text{FNN} \left( \text{LN}(\bar{\mathbf{h}}_j^{(i)}) \right)\end{aligned}$$

**Remark.** In practice, with pre-norm transformer training is more stable.



(a) Encoder in post-norm transformer (b) Encoder in pre-norm transformer

**Remark.** Of all the components in an encoder, attention heads are the only one that allow interaction between tokens.

### 3.1.4 Decoder

**Decoder stack** Composed of  $L$  decoder layers.

Decoder stack

**Decoder layer** Layer to autoregressively generate tokens.

Decoder layer

Its main components are:

**Multi-head self-attention** Processes the input tokens.

**Encoder-decoder multi-head attention/Cross-attention** Uses as query the output of the previous MHSA layer, and as keys and values the output of the encoder stack. In other words, it allows the tokens passed through the decoder to attend the input sequence.

Cross-attention

**Remark.** The output of cross-attention can be seen as an additive delta to improve the activations  $\mathbf{z}_j^{(i)}$  obtained from the first MHSA layer.

**Remark.** As queries are independent to each other, and keys and values are constants coming from the encoder, cross-attention works in a token-wise fashion.

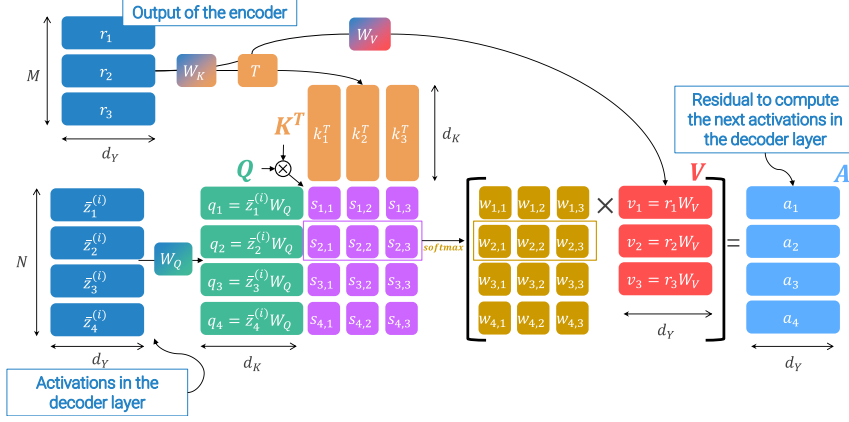


Figure 3.6: Cross-attention data flow

**Feed-forward network** MLP applied after cross-attention.

**Remark.** As for the encoder, there is a post-norm and pre-norm formulation.

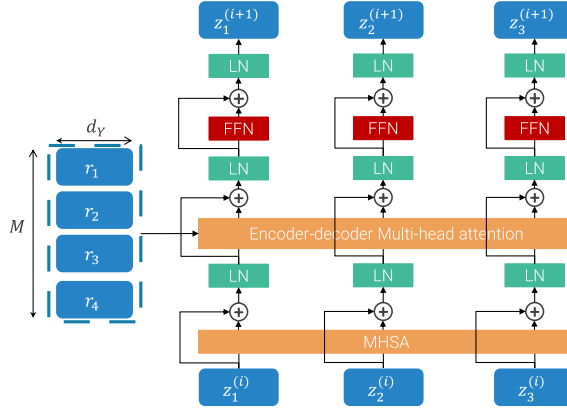


Figure 3.7: Decoder in post-norm transformer

**Parallel training** When training, as the ground truth is known, it is possible to train all decoder outputs in a single pass. Given a target sequence  $[\langle \text{SoS} \rangle, t_1, \dots, t_n, \langle \text{EoS} \rangle]$ , it is processed by the decoder in the following way:

- The input is  $[\langle \text{SoS} \rangle, t_1, \dots, t_n]$  (i.e., without end-of-sequence token).
- The expected output  $[t_1, \dots, t_n, \langle \text{EoS} \rangle]$  (i.e., without start-of-sequence token).

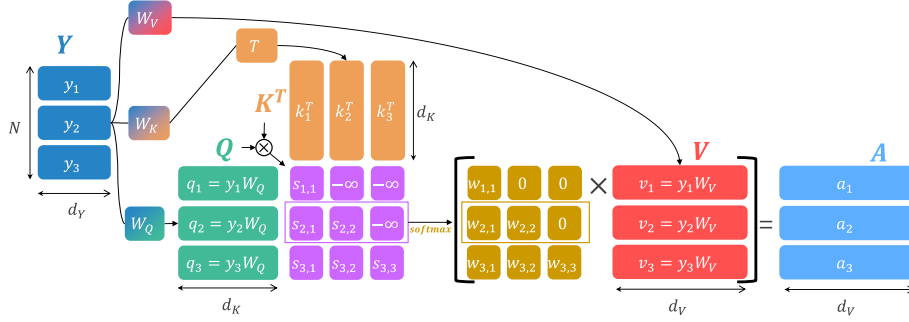
In other words, with a single pass, it is expected that each input token generates the correct output token.

**Remark.** Without changes to the self-attention layer, a token at position  $i$  in the input is able to attend to future tokens at position  $\geq i + 1$ . This causes a data leak as, during inference, autoregressive generation do not have access to future tokens.

**Masked self-attention** Modification to self-attention to prevent tokens to attend at future positions (i.e., at their right). This can be done by either setting the

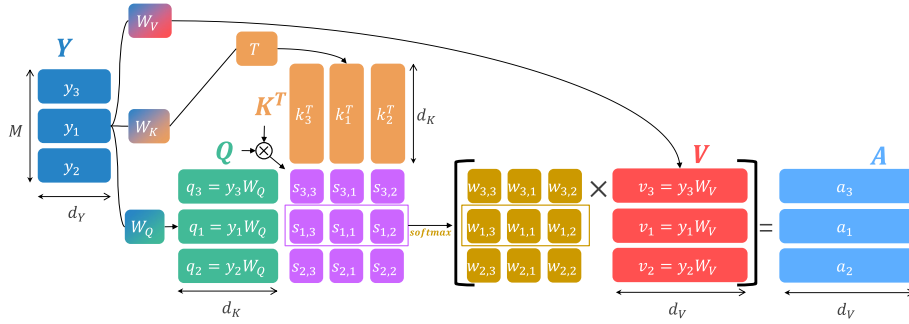
Masked self-attention

similarity scores with future tokens to  $-\infty$  or directly setting the corresponding attention weights to 0 (i.e., make the attention weights a triangular matrix).



### 3.1.5 Positional encoding

**Remark** (Self-attention equivariance to permutation). By permuting the input sequence of a self-attention layer, the corresponding outputs will be the same as if it were the original sequence, but it is affected by the same permutation. Therefore, self-attention alone does not have information on the ordering of the tokens.



**Positional encoding** Vector of shape  $d_Y$  added to the embeddings to encode positional information. Positional encoding can be:

Positional encoding

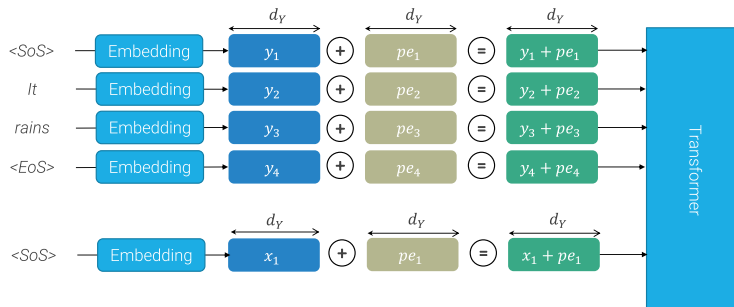
**Fixed** The vector associated to each position is fixed and known before training.

**Example.** The original transformer paper proposed the following encoding:

$$\text{pe}_{\text{pos}, 2i} = \sin\left(\frac{\text{pos}}{10000^{2i/d_Y}}\right) \quad \text{pe}_{\text{pos}, 2i+1} = \cos\left(\frac{\text{pos}}{10000^{2i/d_Y}}\right)$$

where  $\text{pos}$  indicates the position of the token and  $i$  is the dimension of the position encoding vector (i.e., even indexes use  $\sin$  and odd indexes use  $\cos$ ).

**Learned** The vector for position encoding is learned alongside the other parameters.



**Remark** (Transformer vs recurrent neural networks). Given a sequence of  $n$  tokens with  $d$ -dimensional embeddings, self-attention and RNN can be compared as follows:

- The computational complexity of self-attention is  $O(n^2 \cdot d)$  whereas for RNN is  $O(n \cdot d^2)$ . Depending on the task,  $n$  might be a big value.
- The number of sequential operations for training is  $O(1)$  for self-attention (parallel training) and  $O(n)$  for RNN (not parallelizable).
- The maximum path length (i.e., maximum number of operations before a token can attend to all the others) is  $O(1)$  for self-attention (through the multi-head self-attention layer) and  $O(n)$  for RNN (it needs to process each token individually while maintaining a memory).

## 3.2 Vision transformer

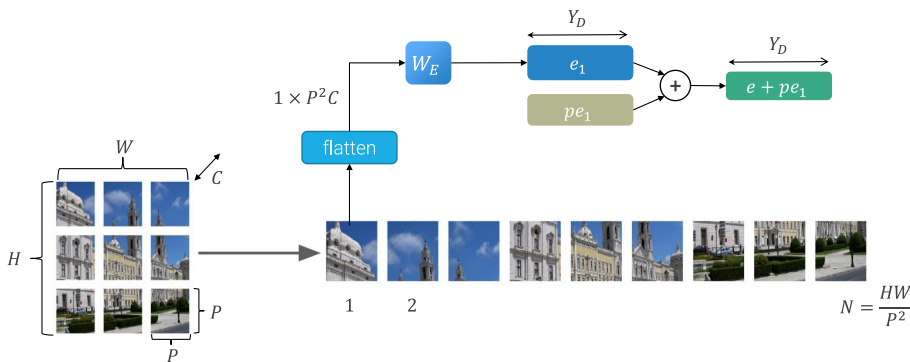
**Remark.** Using single pixels as tokens is unfeasible due to the complexity scaling of transformers as an  $H \times W$  image results in an attention matrix of  $(HW)^2$  entries.

**Example.** Consider an ImageNet image with shape  $224 \times 224$ . The attention weights will have  $(224^2)^2 = 2.5$  bln entries which would require 5 GB to store them in half-precision. A classic 12 layers with 8 heads transformer would require 483 GB of memory to only store all attention matrices.

**Remark.** Compared to text, image pixels are more redundant and less semantically rich. Therefore, processing all of them together is not strictly necessary.

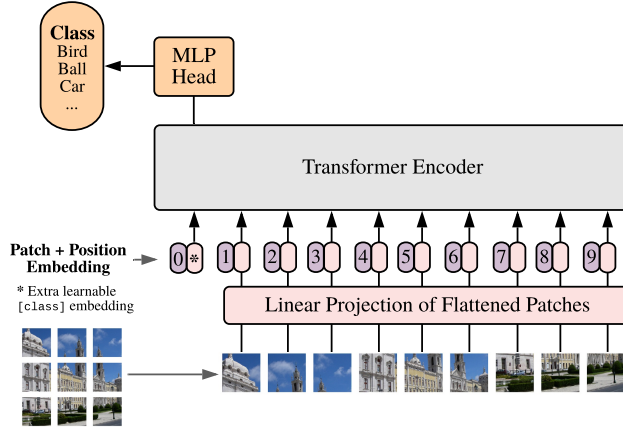
**Patch** Given an image of size  $C \times H \times W$ , it is divided into patches of size  $P \times P$  along the spatial dimension. Each patch is converted into a  $Y_D$ -dimensional embedding for a transformer as follows:

1. Flatten the patch into a  $P^2 C$  vector.
2. Linearly transform it through a learned projection matrix  $W_E \in \mathbb{R}^{P^2 C \times Y_D}$ .
3. Add positional information.



**Vision transformer (ViT)** Transformer encoder that processes embedded patches. A special classification token ([CLS], as in BERT) is appended at the beginning of the sequence to encode the image representation and its embedding is passed through a traditional classifier to obtain the logits.

**Remark.** The (pre-norm) transformer encoder used in vision is the same one as in NLP.



**Remark.** Differently from convolutional neural networks where convolutions are the major source of FLOPs, in ViT the number of FLOPs heavily depends on the length of the input sequence due to the quadratic complexity of the attention mechanism.

**ViT variants** The main size-wise variants of ViT are the following:

| Model     | Layers | Heads | Hidden size | MLP size | Parameters |
|-----------|--------|-------|-------------|----------|------------|
| ViT-base  | 12     | 12    | 768         | 3072     | 86 M       |
| ViT-large | 24     | 16    | 1024        | 4096     | 307 M      |
| ViT-huge  | 32     | 16    | 1280        | 5120     | 632 M      |

Note that, by convention, the MLP size is four times the hidden size.

Moreover, ViT models can also vary depending on the size of the input patch.

The overall notation to denote size and patch is: ViT-**<size>/<patch size>**.

**Results** The main experimental observations and results using vision transformer are the following:

- The first embedding projection  $W_E$  for RGB images shows a similar behavior to convolutions as they tend to recognize edges and color variations.

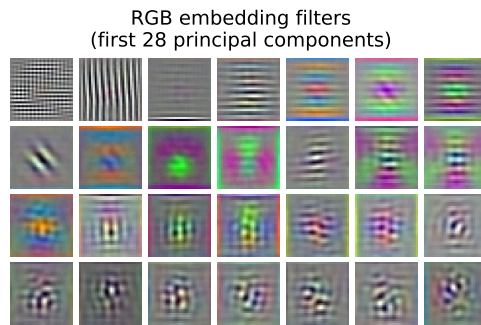


Figure 3.8: Visualization of the columns of the patches linear projection matrix  $W_E$ . Each column has shape  $3P^2$  and can be reshaped to be a  $3 \times P \times P$  image.

- The learned positional embeddings are able to encode information about the row and column positioning of the patches.

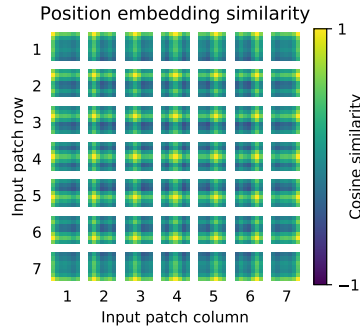


Figure 3.9: Cosine similarity of the positional encoding of each patch compared to all the others

- Attention heads at the lower layers attend at both positions around the patch and far from them. Higher layers, as with convolutions, attend to distant patches.

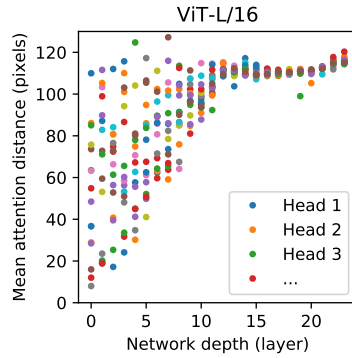


Figure 3.10: Mean attention distance of the heads of ViT-large/16

- On ImageNet top-1 accuracy, ViT outperforms a large ResNet only when pre-trained on a large dataset.

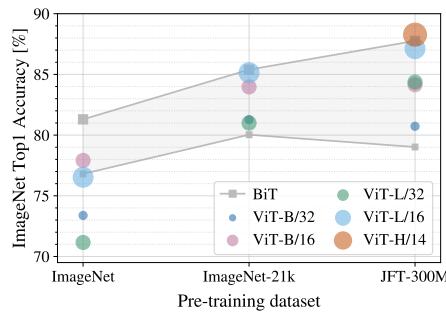


Figure 3.11: ImageNet top-1 accuracy with different pre-training datasets. BiT represents ResNet (two variants).

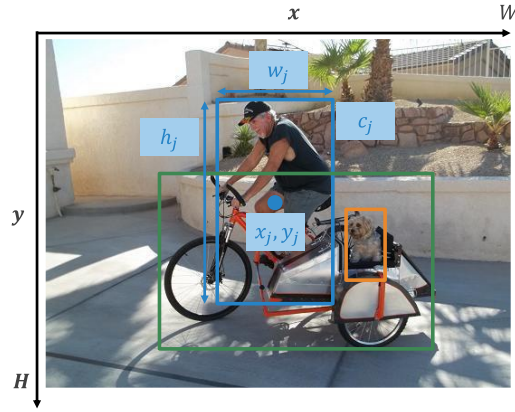
**Remark.** Comparison between convolutional neural networks and vision transformer is not straightforward.

**Remark.** On an execution efficiency point-of-view, the currently more common inference hardware is more optimized for convolutions.

## 4 Object detection

**Object detection** Given an RGB  $W \times H$  image, determine a set of objects  $\{o_1, \dots, o_n\}$  contained in it. Each object  $o_j$  is described by: Object detection

- A category  $c_j \in \{1, \dots, C\}$  as in image classification.
- A bounding box  $BB_j = [x_j, y_j, w_j, h_j]$  where  $x_j, w_j \in [0, W - 1]$  and  $y_j, h_j \in [0, H - 1]$ .  $(x_j, y_j)$  is the center and  $(w_j, h_j)$  is the size of the box.
- A confidence score  $\rho_j$ .



**Remark.** Differently from classification, a model has to:

- Be able to output a variable number of results.
- Output both categorical and spatial information.
- Work on high resolution input images.

### 4.1 Metrics

**Intersection over union (IoU)** Measures the amount of overlap between two boxes computed as the ratio of the area of intersection over the area of union:

Intersection over union (IoU)

$$\text{IoU}(BB_i, BB_j) = \frac{|BB_i \cap BB_j|}{|BB_i| + |BB_j| - |BB_i \cup BB_j|}$$

**True/false positive criteria** Given a threshold  $\rho_{\text{IoU}}$ , a detection  $BB_i$  is a true positive (TP) w.r.t. a ground-truth  $\widehat{BB}_j$  if it is classified with the same class and:

$$\text{IoU}(BB_i, \widehat{BB}_j) > \rho_{\text{IoU}}$$

**Remark.** Confidence can also be considered when determining a match through a threshold  $\rho_{\text{min}}$ .

**Recall** Measures the number of ground-truth objects that have been found:

$$\text{recall} = \frac{|\text{TP}|}{|\text{ground-truth boxes}|}$$

**Precision** Measures the number of correct detections among all the predictions:

$$\text{precision} = \frac{|\text{TP}|}{|\text{model detections}|}$$

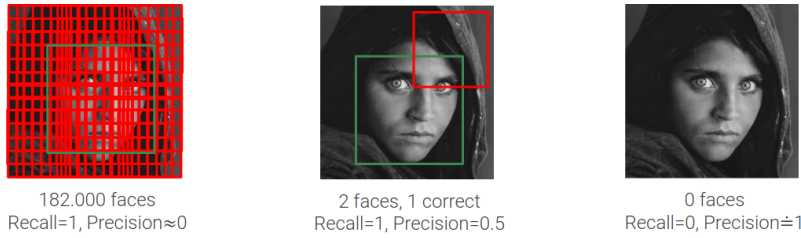


Figure 4.1: Recall and precision in different scenarios

**Precision-recall curve** Plot that relates all possible precisions and recalls of a detector.

**Example.** Consider the following image and the bounding boxes found by a detector:

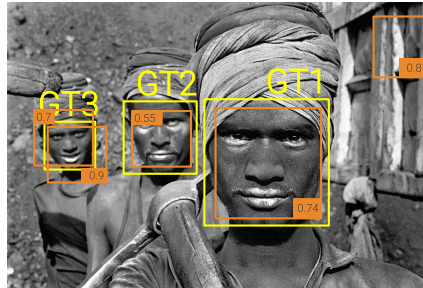
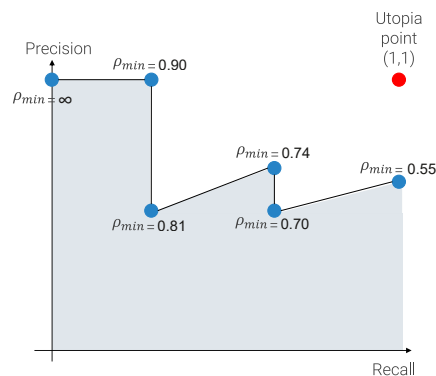


Figure 4.2: Ground-truth (yellow boxes) and predictions (orange boxes) with their confidence score

By sorting the confidence scores, it is possible to plot the precision-recall curve by varying the threshold  $\rho_{\min}$ :



**Remark.** Recall is monotonically decreasing, while precision can both decrease and increase.



**Average precision (AP)** Area under the precision-recall curve.

Average precision  
(AP)  
Mean AP (mAP)

**Mean average precision (mAP)** Mean AP over the possible classes.

**COCO mean average precision** Compute for each class the average AP over varying  $\rho_{\text{IoU}}$  (e.g., in the original paper,  $\rho_{\text{IoU}} \in [0.5, 0.95]$  with 0.05 steps) and further average them over the possible classes.

COCO mAP

**Remark.** Higher COCO mAP indicates a detector with good localization capabilities.

## 4.2 Viola-Jones

**Viola-Jones** General framework for object detection, mainly applied to faces.

Viola-Jones object  
detection

It is one of the first successful applications of machine learning in computer vision and has the following basis:

- Use AdaBoost to learn an ensemble of features.
- Use multi-scale rectangular features computed efficiently using integral images.
- Cascade to obtain real-time speed.

### 4.2.1 Boosting

**Weak learner** Classifier with an error rate slightly higher than a random classifier (i.e., in a balanced binary task, accuracy slightly higher than 50%).

Weak learner

**Decision stump** Classifier that learns a threshold for a single feature (i.e., decision tree with depth 1).

Decision stump

**Strong learner** Classifier with an accuracy strongly correlated with the ground-truth.

Strong learner

**Adaptive boosting (AdaBoost)** Ensemble of  $M$  weak learners  $\text{WL}_i$  that creates a strong learner  $\text{SL}$  as the linear combination of their predictions (i.e., weighted majority vote):

Adaptive boosting  
(AdaBoost)

$$\text{SL}(x) = \left( \sum_{i=1}^M \alpha_i \text{WL}_i(x) > 0 \right)$$

**Training** Given  $N$  training samples  $(x^{(i)}, y^{(i)})$  and  $M$  untrained weak learners  $\text{WL}_i$ , training is done sequentially by tuning a learner at the time:

Boosting training

1. Uniformly weigh each sample:  $w^{(i)} = \frac{1}{N}$ .
2. For each weak learner  $\text{WL}_j$  ( $j = 1, \dots, M$ ):
  - a) Fit the weak learner on the weighted training data.
  - b) Compute its error rate:

$$\varepsilon_j = \sum_{i: x^{(i)} \text{ misclassified}} w^{(i)}$$

- c) Compute the reweigh factor:

$$\beta_j = \frac{1 - \varepsilon_j}{\varepsilon_j}$$

d) Increase the weight of misclassified samples:

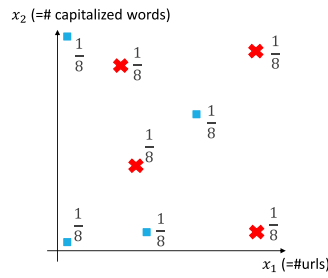
$$w^{(i)} = w^{(i)} \beta_j$$

and re-normalize all samples so that their weights sum to 1.

3. Define the strong classifier as:

$$\text{SL}(x) = \left( \sum_j \ln(\beta_j) \text{WL}_j(x) > 0 \right)$$

**Example.** Consider the problem of spam detection with two features  $x_1$  and  $x_2$  (number of URL and capitalized words, respectively). The training samples and their initial weights are the following:

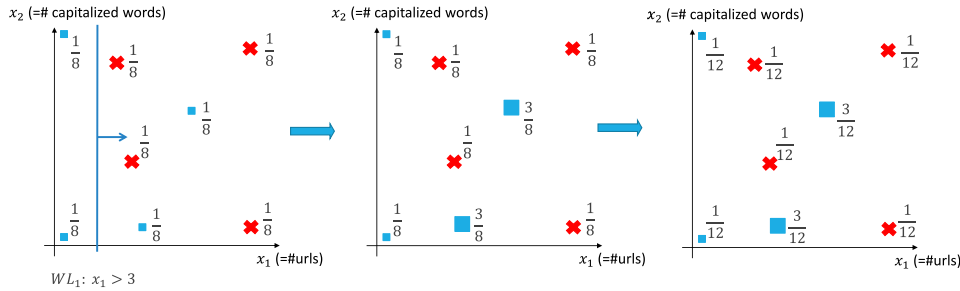


We want to train an ensemble of 3 decision stumps  $\text{WL}_j$ .

Let's say that the first weak classifier learns to detect spam using the criteria  $x_1 > 3$ . The error rate and reweigh factor are:

$$\varepsilon_1 = \frac{1}{8} + \frac{1}{8} \quad \beta_1 = \frac{1 - \varepsilon_1}{\varepsilon_1} = 3$$

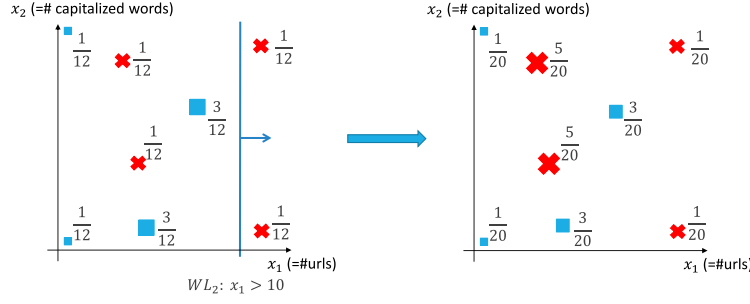
The new reweighed and normalized samples are:



Now, assume that the second classifier learns  $x_1 > 10$ . The error rate and reweigh factor are:

$$\varepsilon_2 = \frac{1}{12} + \frac{1}{12} \quad \beta_2 = \frac{1 - \varepsilon_2}{\varepsilon_2} = 5$$

The new reweighed and normalized samples are:



Finally, the third classifier learns  $x_2 > 20$ . The error rate and reweigh factor are:

$$\varepsilon_3 = \frac{1}{20} + \frac{1}{20} + \frac{3}{20} \quad \beta_3 = \frac{1 - \varepsilon_3}{\varepsilon_3} = 3$$

The strong classifier is defined as:

$$SL(x) = \begin{cases} 1 & \text{if } (\ln(3)WL_1(x) + \ln(5)WL_2(x) + \ln(3)WL_3(x)) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

**Haar-like features** For face detection, a  $24 \times 24$  patch of the image is considered (for now) and the weak classifiers define rectangular filters composed of 2 to 4 subsections applied at fixed positions of the patch.

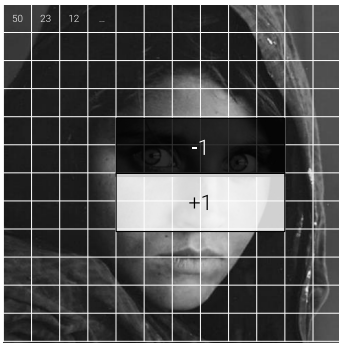
Haar-like features

Given a patch  $x$ , a weak learned  $WL_j$  classifies it as:

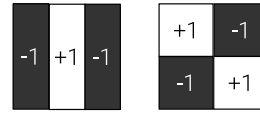
$$WL_j(x) = \begin{cases} 1 & \text{if } s_j f_j \geq s_j \rho_j \\ -1 & \text{otherwise} \end{cases}$$

where the learned parameters are:

- The size and position of the filter ( $f_j$  is the result of applying the filter).
- The polarity  $s_j$ .
- The threshold  $\rho_j$ .



(a) Filter applied on a patch



(b) Other possible filters

Figure 4.3: Example of filters

**Remark.** AdaBoost is used to select a subset of the most effective filters.

## 4.2.2 Integral images

**Integral image** Given an image  $I$ , its corresponding integral image  $II$  is defined as:

Integral image

$$II(i, j) = \sum_{i' \leq i, j' \leq j} I(i', j')$$

In other words, the value at coordinates  $(i, j)$  in the integral image is the sum of all the pixels of the original image in an area that starts from the top-left corner and has as bottom-right corner the pixel at  $(i, j)$ .

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 6 | 3 |
| 5 | 4 | 1 | 4 |
| 2 | 1 | 3 | 3 |

Input image  $I$

|    |    |    |    |
|----|----|----|----|
| 3  | 5  | 11 | 14 |
| 8  | 14 | 21 | 27 |
| 10 | 17 | 27 | 36 |

Integral image  $II$

Figure 4.4: Example of integral image

**Remark.** In practice, the integral image can be computed recursively as:

$$II(i, j) = II(i, j - 1) + II(i - 1, j) - II(i - 1, j - 1) + I(i, j)$$

**Fast feature computation** Given an image  $I$  and its integral image  $II$ , the sum of the pixels in a rectangular area of  $I$  can be computed in constant time as:

Fast feature computation

$$II(A) - II(B) - II(C) + II(D)$$

where  $A$ ,  $B$ ,  $C$ , and  $D$  are coordinates defined as in Figure 4.5.

|  |  |  |   |  |  |  |  |   |  |  |  |
|--|--|--|---|--|--|--|--|---|--|--|--|
|  |  |  |   |  |  |  |  |   |  |  |  |
|  |  |  |   |  |  |  |  |   |  |  |  |
|  |  |  | D |  |  |  |  | B |  |  |  |
|  |  |  |   |  |  |  |  |   |  |  |  |
|  |  |  |   |  |  |  |  |   |  |  |  |
|  |  |  |   |  |  |  |  |   |  |  |  |
|  |  |  | C |  |  |  |  | A |  |  |  |
|  |  |  |   |  |  |  |  |   |  |  |  |

Figure 4.5: Summation of the pixels in the blue area

**Multi-scale sliding window** During inference, Viola-Jones is a sliding window detector that scans the image considering patches of fixed size.

Multi-scale sliding window

To achieve scale-invariance, patches of different size are used, scaling the rectangular filters accordingly.

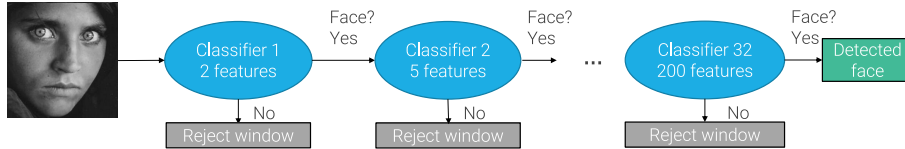
**Remark.** The integral image allows to compute the features in constant time independently of the patch size.

### 4.2.3 Cascade

**Cascade** To obtain real-time predictions, a hierarchy of classifiers is used to quickly reject background patches. The first classifier considers a few features while the following ones use more.

Cascade

| **Remark.** The simpler classifiers have a high recall so that they do not discard faces.



### 4.2.4 Non-maximum suppression

**Non-maximum suppression (NMS)** Algorithm to obtain a single bounding box from several overlapping ones. Given the set of all the bounding boxes with their confidence that a detector found, NMS works as follows:

Non-maximum suppression (NMS)

1. Until there are unchecked boxes:
  - a) Consider the bounding box with the highest confidence.
  - b) Eliminate all boxes with overlap higher than a chosen threshold (e.g.,  $\text{IoU} > 0.5$ ).

| **Remark.** If two objects are close, NMS might detect them as a single instance.

## 4.3 CNN for object detection

### 4.3.1 Object localization

**Object localization** Subset of object detection problems where it is assumed that there is only a single object to detect.

Object localization

**CNN for object localization** A pre-trained CNN can be used as feature extractor with two heads:

CNN for object localization

**Classification head** Used to determine the class.

**Regression head** Used to determine the bounding box.

Given:

- The ground-truth class  $c^{(i)}$  and bounding box  $BB^{(i)}$ ,
- The predicted class logits  $\text{scores}^{(i)}$  and bounding box  $\widehat{BB}^{(i)}$ ,

training is a multi-task learning problem with two losses:

$$\mathcal{L}^{(i)} = \mathcal{L}_{\text{CE}}(\text{softmax}(\text{scores}^{(i)}), \mathbb{1}[c^{(i)}]) + \lambda \mathcal{L}_{\text{MSE}}(\widehat{BB}^{(i)}, BB^{(i)})$$

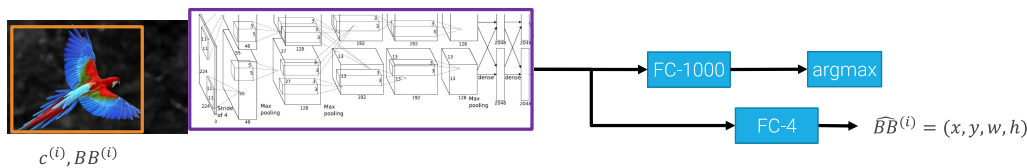


Figure 4.6: Localizer with AlexNet as feature extractor and 1000 classes

**Remark.** A localization CNN can be used as a sliding window detector to detect multiple objects.

An additional background class (**bg**) has to be added to mark patches without an object. Moreover, when a patch belongs to the background, the loss related to the bounding box should be ignored. Therefore, the loss becomes:

$$\mathcal{L}^{(i)} = \mathcal{L}_{\text{CE}}\left(\text{softmax}(\text{scores}^{(i)}), \mathbb{1}[c^{(i)}]\right) + \lambda \mathbb{1}[c^{(i)} \neq \text{bg}] \mathcal{L}_{\text{MSE}}\left(\widehat{BB}^{(i)}, BB^{(i)}\right)$$

where  $\mathbb{1}[c^{(i)} \neq \text{bg}]$  is 1 iff the ground-truth class  $c^{(i)}$  is not the background class.

This approach has two main problems:

- Background patches are usually more frequent, requiring additional work to balance the dataset or mini-batch.
- There are too many patches to check.

### 4.3.2 Region proposal detectors

**Region proposal** Class of algorithms to find regions likely to contain an object.

Region proposal

**Selective search** Region proposal algorithm that works as follows:

Selective search

1. Segment the image into superpixels (i.e., uniform regions).
2. Merge superpixels based on similarity of color, texture, or size. Each aggregation generates a proposed region.
3. Repeat until everything collapses in a single region.

**Remark.** Region proposal algorithms should have a high recall.

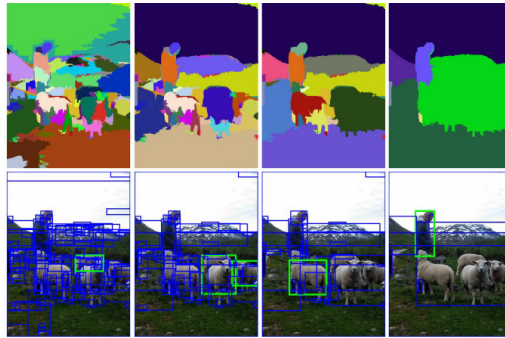


Figure 4.7: Example of some iterations of selective search

**Region-based CNN (R-CNN)** Use a CNN for object localization with selective search. The workflow is the following:

Region-based CNN  
(R-CNN)

1. Run selective search to get the proposals.
2. For each proposal:
  - a) Warp the proposed crop to the input shape of the CNN.
  - b) Feed the warped crop to the CNN to get:
    - A class prediction.
    - A bounding box correction (as selective search already gives a box).

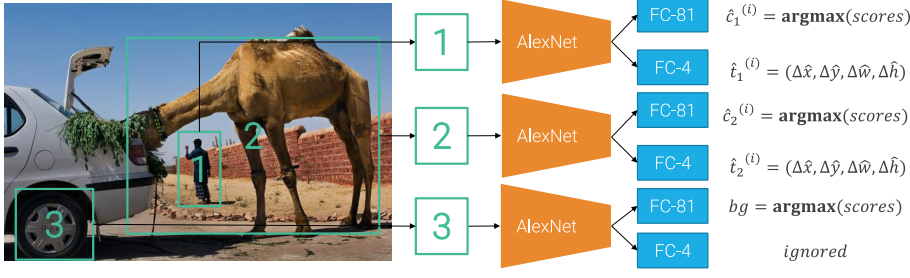


Figure 4.8: Example of R-CNN using AlexNet

**Bounding box correction** Given a selective search bounding box  $BB_{SS}$  and the network predicted correction  $\hat{t}$ :

Bounding box correction

$$BB_{SS} = (x_{SS}, y_{SS}, w_{SS}, h_{SS}) \quad \hat{t} = (\Delta\hat{x}, \Delta\hat{y}, \Delta\hat{w}, \Delta\hat{h})$$

the output box  $BB_{out}$  is given by:

$$BB_{out} = (x_{SS} + w_{SS}\Delta\hat{x}, y_{SS} + h_{SS}\Delta\hat{y}, w_{SS}\exp(\Delta\hat{w}), h_{SS}\exp(\Delta\hat{h}))$$

where the center is a translation relative to the box size and the dimensions are log-space scaled.

**Remark.** This formulation is due to the fact that a neural network tend to output smaller values, so overall it results an easier task to learn.

**Training** Given a training sample  $x^{(i)}$  with class  $c^{(i)}$  and bounding box  $BB^{(i)} = [x_{GT}, y_{GT}, w_{GT}, h_{GT}]$ , the selective search box  $BB_{SS}^{(i)}$  associated to it during training is the one with the most overlap, while the others are considered background. The target correction  $t^{(i)} = [\Delta x, \Delta y, \Delta w, \Delta h]$  is computed as:

$$\Delta x = \frac{x_{GT} - x_{SS}}{w_{SS}} \quad \Delta y = \frac{y_{GT} - y_{SS}}{h_{SS}} \quad \Delta w = \ln\left(\frac{w_{GT}}{w_{SS}}\right) \quad \Delta h = \ln\left(\frac{h_{GT}}{h_{SS}}\right)$$

The loss is then defined as:

$$\mathcal{L}^{(i)} = \mathcal{L}_{CE}(\text{softmax}(\text{scores}^{(i)}), \mathbb{1}[c^{(i)}]) + \lambda \mathbb{1}[c^{(i)} \neq \text{bg}] \mathcal{L}_{MSE}(\hat{t}^{(i)}, t^{(i)})$$

**Remark.** Empirically, it has been observed that feature computation, fine-tuning, bounding box correction, and architecture are important to increase mAP.

**Remark.** Instead of AlexNet, any other CNN can potentially be used.

**Remark.** R-CNN is slow as it requires to process each proposed crop.

**Fast R-CNN** Optimization to R-CNNs that avoids processing overlapping pixels of the proposed crops with the CNN multiple times:

Fast R-CNN

1. Process the original image with the feature extractor section of the CNN.
2. Compute the proposed crops from the feature extractor activations and adjust to the correct shapes through pooling.
3. Feed each crop to the remaining fully-connected layers of the CNN and the task-specific heads.

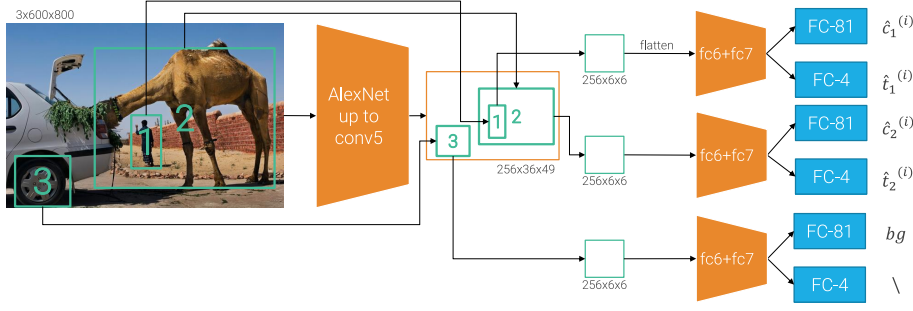


Figure 4.9: Example of fast R-CNN using AlexNet

**Region of interest pool (RoIPool)** Given an input activation of shape  $C_a \times H_a \times W_a$  and the desired output spatial dimension  $H_o \times W_o$ , RoIPool allows to obtain an output of shape  $C_a \times H_o \times W_o$  as follows:

Region of interest pool (RoIPool)

1. Project the proposed region from the original image to the feature extractor activations.
2. Snap the projection to grid (i.e., apply rounding).

**Remark.** As a single pixel in the activation encodes multiple pixels of the input image, snapping might lose some information.

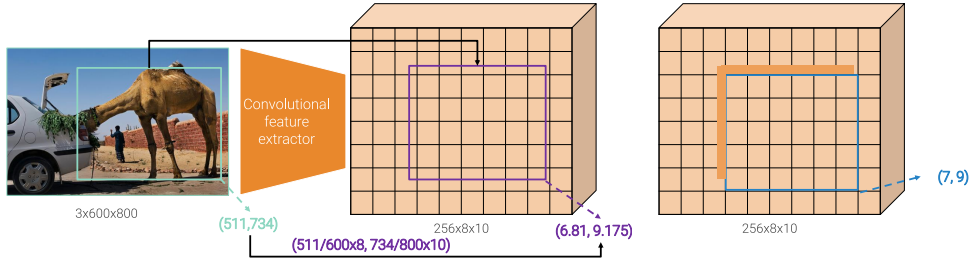


Figure 4.10: Project and snap operations

3. Apply max pooling with kernel of approximately size  $\left\lceil \frac{H_r}{H_o} \right\rceil \times \left\lceil \frac{W_r}{W_o} \right\rceil$  and stride approximately  $\left\lfloor \frac{H_r}{H_o} \right\rfloor \times \left\lfloor \frac{W_r}{W_o} \right\rfloor$ .

**Remark.** Approximations are needed as the spatial dimension of the crop might not be directly convertible to the desired output shape. So, some iterations might not use the precise kernel size or stride.

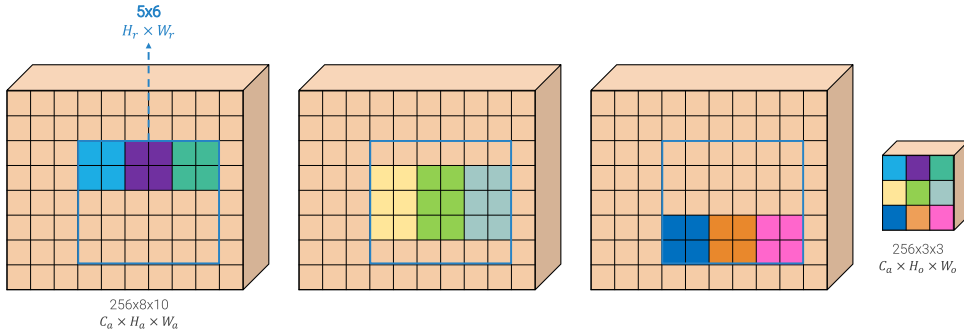


Figure 4.11: Pooling operation with varying kernel size



**Remark.** Snapping and approximate pooling introduce two sources of quantization.

**Huber loss** Instead of L2, fast R-CNN uses the Huber (i.e., smooth L1) loss to compare bounding boxes: Huber loss

$$\mathcal{L}_{BB}^{(i)} = \sum_{d \in \{x, y, w, h\}} \mathcal{L}_{\text{huber}} \left( \Delta \hat{d}^{(i)} - \Delta d^{(i)} \right)$$

$$\mathcal{L}_{\text{huber}}(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } |a| \leq 1 \\ |a| - \frac{1}{2} & \text{otherwise} \end{cases}$$

**Remark.** L2 grows quadratically with the loss which makes it sensitive to outliers. Smooth L1 maintains the gradient constant to 1 for big values.

**Remark.** Fast R-CNN reduces the number of FLOPs when applying the convolutions but moves the bottleneck to the feed-forward layers.

|                   | Conv FLOPs               | FF FLOPs                |
|-------------------|--------------------------|-------------------------|
| <b>R-CNN</b>      | $n \cdot 2154 \text{ M}$ | $n \cdot 117 \text{ M}$ |
| <b>Fast R-CNN</b> | 16 310 M                 | $n \cdot 117 \text{ M}$ |

Table 4.1: FLOPs comparison with AlexNet as CNN and  $n$  proposals

**Remark.** The slowest component of fast R-CNN is selective search.

**Faster R-CNN** Selective search is dropped and a region proposal network (RPN) is used: Faster R-CNN

1. Pass the input image through the feature extractor section of the CNN.
2. Feed the activations to the RPN to determine the regions of interest.
3. Continue as in fast R-CNN.

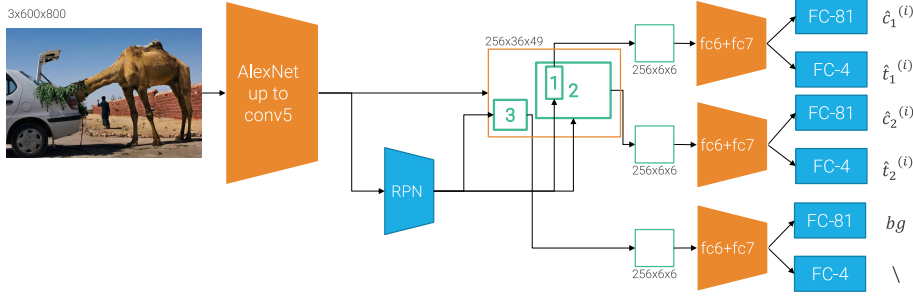


Figure 4.12: Example of faster R-CNN using AlexNet

**Region proposal network (RPN)** Network that takes as input the image activations of shape  $C_L \times H_L \times W_L$  and outputs:

Region proposal network (RPN)

- The objectness scores of shape  $2 \times H_L \times W_L$ .

**Remark.** The two channels are due to the fact that the original paper uses a two-way softmax, which in practice is equivalent to a sigmoid.

- The proposed boxes of shape  $4 \times H_L \times W_L$ .

In other words, an RPN makes a prediction at each input pixel.

**Remark.** RPN has a small fixed receptive field (that should roughly be the size of an object), but can predict boxes larger than it.

**Remark.** As is, RPN is basically solving object detection as it has to determine the exact box for the objects, which might be a difficult task.

**Anchor** Known bounding box with fixed scale and aspect-ratio.

Anchor

**Anchor correction** Make an RPN predict a correction for a known anchor whose center is positioned at the center of the receptive field.

Anchor correction

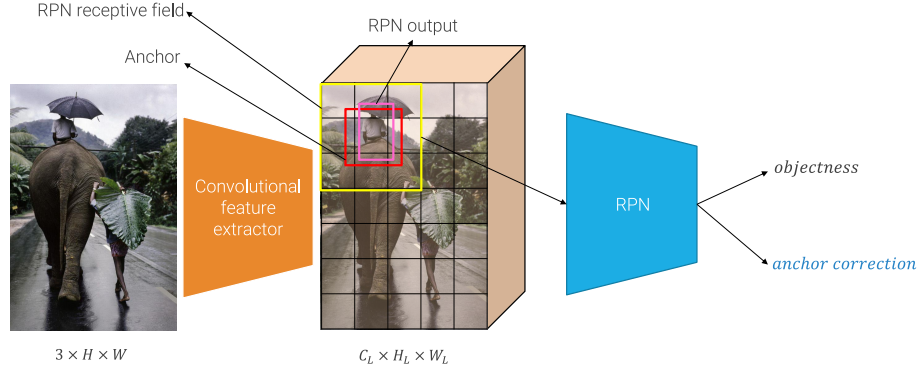


Figure 4.13: Example of an iteration of a 1-anchor RPN

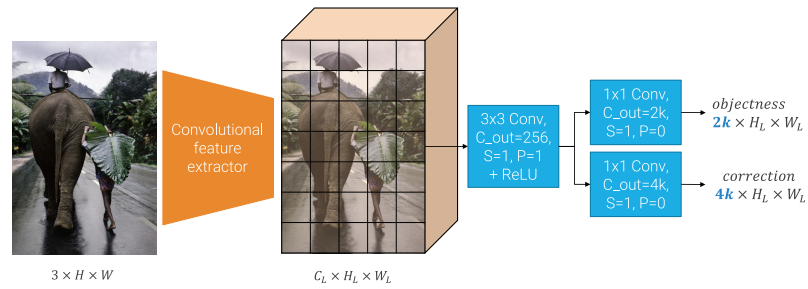
**$k$  anchors correction** Consider  $k$  different anchors so that the RPN outputs  $k$  objectness scores (overall shape of  $2k \times H_L \times W_L$ ) and  $k$  corrections (overall shape of  $4k \times H_L \times W_L$ ) at each pixel.

$k$  anchors correction

**Remark.** Virtually, this can be seen as putting together the outputs of  $k$  different 1-anchor RPN (with different anchors).

**Architecture** An RPN is implemented as a two-layer CNN:

1. A  $3 \times 3$  convolution with padding 1, stride 0, 256 output channels, and ReLU as activation.
2. Two parallel  $1 \times 1$  convolutions with no padding and stride 1 with  $2k$  and  $4k$  output channels, respectively.



**Remark.** Only the proposals with the highest objectness scores are considered at training and test time.

**Training** Given a training image  $x^{(i)}$  and a bounding box  $BB_{GT}$ , the  $j$ -th anchor  $BB_A$  can be a:

**Negative anchor**  $BB_A$  has objectness score  $o^{(i,j)} = 0$  (i.e., it contains background) if  $\text{IoU}(BB_{GT}, BB_A) < 0.3$ .

**Positive anchor**  $BB_A$  has objectness score  $o^{(i,j)} = 1$  (i.e., it contains an object) whether:

- $\text{IoU}(BB_{GT}, BB_A) \geq 0.7$ .
- $\text{IoU}(BB_{GT}, BB_A)$  is the largest and none of the others are  $\geq 0.7$ .

**Ignored anchor**  $BB_A$  is not considered for this sample in all other cases.

A mini-batch is composed of all the positive anchors and it is filled with negative anchors to reach the desired size.

**Remark.** Differently from R-CNN, multiple boxes have a positive label as it is ambiguous to determine which anchor is responsible for recognizing a particular object.

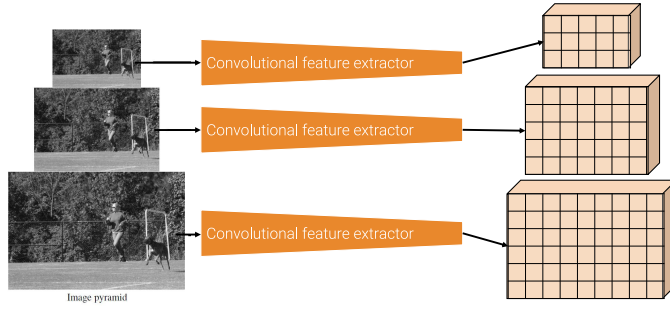
**Remark.** R-CNN is unable to detect objects smaller than the grid size.

### 4.3.3 Multi-scale detectors

**Image pyramid multi-scale detection** Obtain a feature pyramid by feeding the input image to the convolutional feature extractor at different scales.

Image pyramid  
multi-scale detection

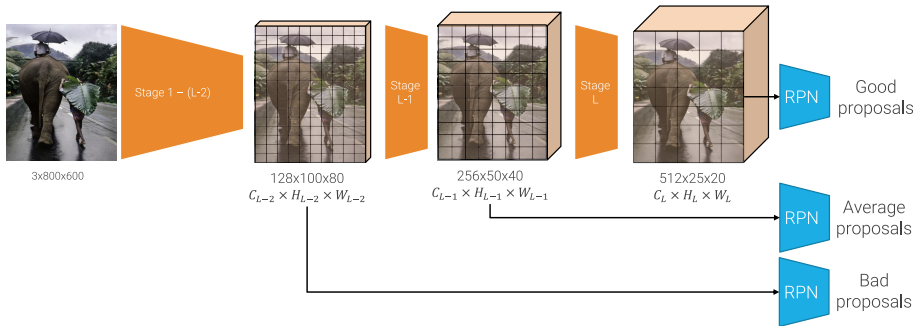
**Remark.** This approach creates effective features at different scales, but it is computationally expensive.



**CNN pyramid multi-scale detection** CNNs naturally produce a pyramid of features composed of the activations at each stage.

CNN pyramid  
multi-scale detection

**Remark.** This approach do not affect computational cost, but features at smaller scales have bad semantic quality as they are at the beginning of the network.



**Feature pyramid network (FPN)** Network that enhances small scale features (at the beginning of the network) by combining them with high resolution and semantically rich features (at the end of the network).

Feature pyramid  
network (FPN)

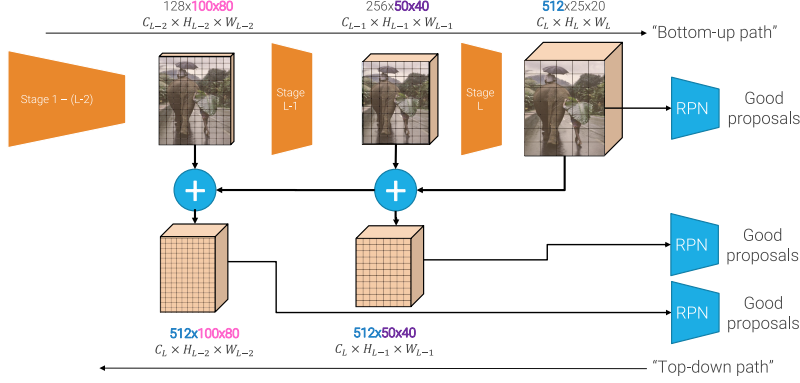


Figure 4.14: General FPN flow

**Top-down path** Given the activation  $A^{(L)}$  at layer  $L$  and the activation  $A^{(L-1)}$  at layer  $L - 1$ , the top-down path computes the enhanced features as follows:

1. Obtain  $\bar{A}^{(L)}$  by upsampling  $A^{(L)}$  using nearest neighbor to match the spatial dimension of  $A^{(L-1)}$ .
2. Obtain  $\bar{A}^{(L-1)}$  by applying a  $1 \times 1$  convolution to  $A^{(L-1)}$  to match the number of channels of  $A^{(L)}$ .
3. Sum  $\bar{A}^{(L)}$  and  $\bar{A}^{(L-1)}$ , and apply a  $3 \times 3$  convolution to reduce aliasing artifacts caused by upsampling.

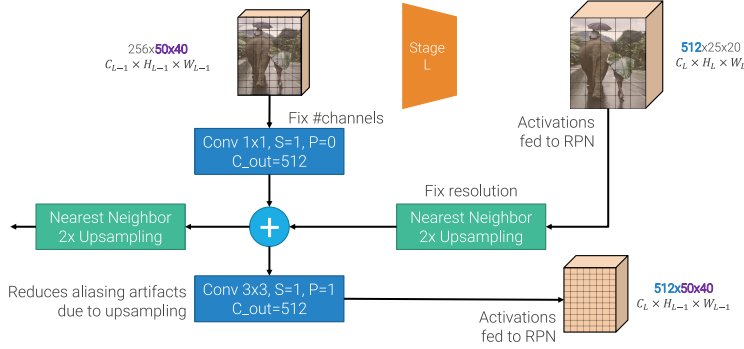


Figure 4.15: FPN top-down flow

**Faster R-CNN with FPN** The FPN is used with the feature extractor to obtain a pyramid of features  $P_1, \dots, P_n$ . A proposal of the RPN is assigned to the most suited level of the pyramid  $P_k$ .

Faster R-CNN with FPN

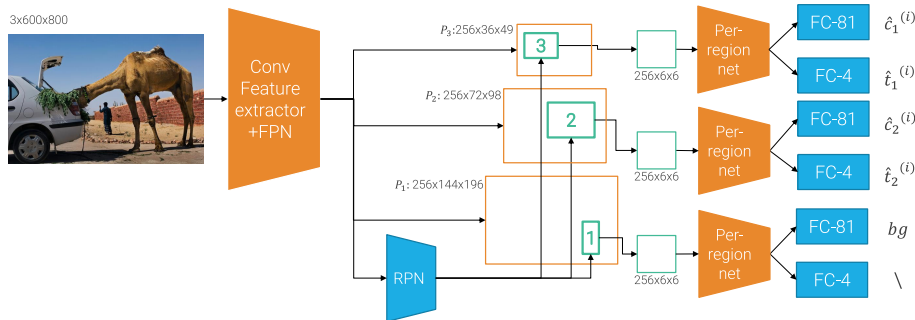


Figure 4.16: Example of faster R-CNN with FPN

**Remark.** Given a proposal of the RPN with size  $w \times h$ , the most suited level of the pyramid  $P_k$  is determined by the following formula:

$$k = \left\lfloor k_0 + \log_2 \left( \frac{\sqrt{wh}}{224} \right) \right\rfloor$$

where  $k_0$  is the level of the feature map at which a  $224 \times 224$  proposal should be mapped to.

**Remark.** R-CNN and its improvements can be seen as a detector with two stages:

**Stage 1** Passes the input image into the feature extractor and RPN to obtain the activations and proposals.

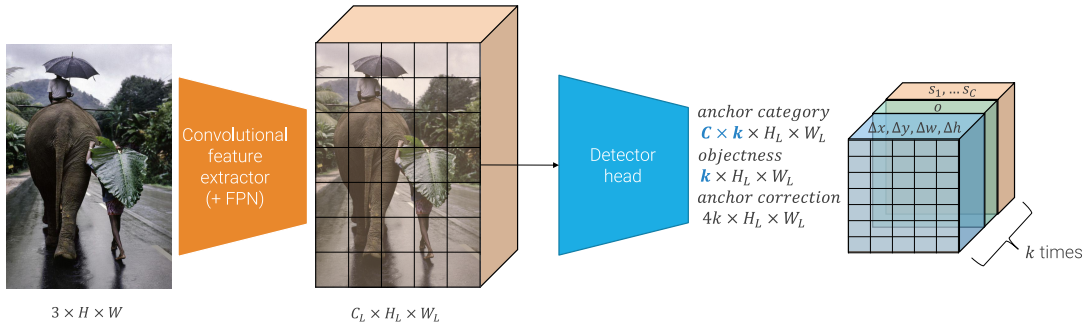
**Stage 2** Passes each proposal through RoI pooling and per-region classification and correction.

#### 4.3.4 One-stage detectors

**One-stage detector** Drop the second stage of R-CNNs and let the RPN determine both the bounding box and the class.

One-stage detector

As an objectness score is available, it is used to determine whether the region contains background (i.e., there is no need to add a background class).



**Multi-label classification** Task in which classes are not mutually exclusive (i.e., multiple classes can be assigned to the same object).

Multi-label classification

With  $C$  classes, the architecture of a multi-label detector has  $C$  independent sigmoid functions at the output layer. Given the predictions for the  $j$ -th box of the  $i$ -th sample  $\mathbf{s}^{(i,j)} \in \mathbb{R}^C$  and the ground-truth  $\mathbf{y}^{(i,j)} \in \{0, 1\}^C$ , the classification loss is defined as:

$$\mathcal{L}(\mathbf{s}^{(i,j)}, \mathbf{y}^{(i,j)}) = \sum_{k=1}^C \text{BCE} \left( \sigma(\mathbf{s}_k^{(i,j)}), \mathbf{y}_k^{(i,j)} \right)$$

**Remark.** Not assigning a class (i.e.,  $\forall k = 1 \dots C : \mathbf{y}_k^{(i,j)} = 0$ ) can also be used to model the background class.

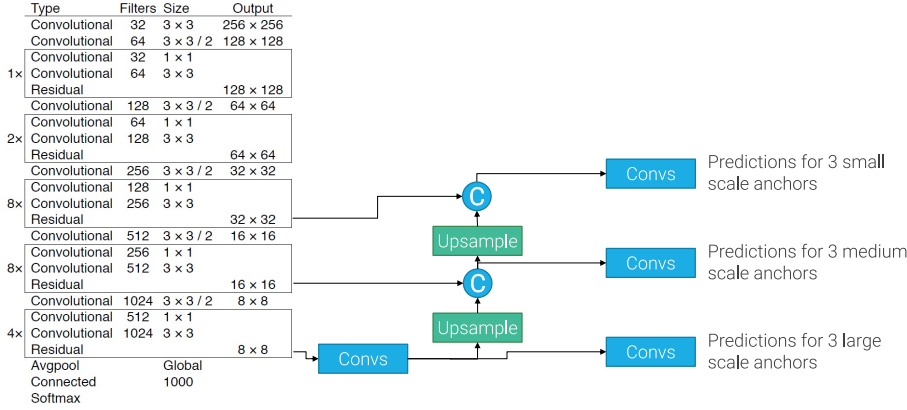
**Remark.** Even if the task is multi-class classification, it has been observed that multiple sigmoids perform better.

**YOLOv3** One-stage detector that uses DarkNet-53 as backbone for feature extraction and learned anchors.

YOLOv3

**DarkNet-53** Architecture based on bottleneck residual blocks and multi-scale concatenated features (in FPN they were summed).

It is optimized to obtain a good accuracy-speed trade-off.



**Learned anchors** Size and aspect-ratio of the anchors are determined using  $k$ -means on the training set boxes. The distance between a box  $BB$  and a centroid  $BB_{\text{centroid}}$  is computed as:

$$1 - \text{IoU}(BB_{\text{centroid}}, BB)$$

**Remark.** YOLOv2 uses  $k = 5$ , while YOLOv3 uses  $k = 9$ .

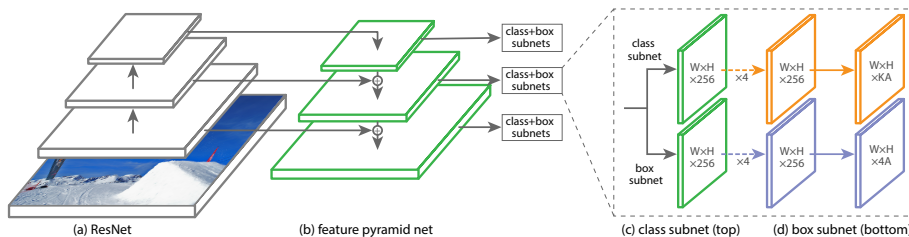
**Remark** (Class unbalance). The object detection task is usually unbalanced towards (easy) negative boxes (i.e., background). RPNs and one-stage detectors are particularly sensitive to this problem as they always have to evaluate each possible anchor (two-stage detectors are less affected as they only consider top-scored proposals). This unbalance may cause:

- Suboptimal models as easy negative boxes still have a non-zero loss and, being the majority, they make gradient descent unnecessarily consider them.
- Inefficient training due to the fact that random sampled mini-batches will mostly contain easy negative boxes that do not provide useful learning information.

**Hard negative mining** Sort negative anchors by classification loss and apply NMS. Only top scoring anchors are used in mini-batches.

**Remark.** This approach allows to reduce the effect of unbalancing, but only a subset of negative examples are used to train the model.

**RetinaNet** One-stage detector that uses ResNet with FPN as feature extractor. The classification and regression heads are  $3 \times 3$  convolutions that do not share parameters (differently from the traditional RPN). Moreover, several tweaks have been considered to deal with class unbalance.



**Remark.** The standard cross-entropy has a non-negligible magnitude even for correctly classified examples. Therefore, easy negative boxes over-weigh the overall loss even if they are already well-classified (i.e., with output probability  $p \gg 0.5$ ).

**Binary focal loss** Given the output probability  $p$  and ground-truth label  $y$ , consider the following notation:

Binary focal loss

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise} \end{cases}$$

$$\text{BCE}(p, y) = \text{BCE}(p_t) = -\ln(p_t) = \begin{cases} -\ln(p) & \text{if } y = 1 \\ -\ln(1 - p) & \text{otherwise} \end{cases}$$

Binary focal loss down-weighs the loss as follows:

$$\text{BFL}_\gamma(p_t) = (1 - p_t)^\gamma \text{BCE}(p_t) = -(1 - p_t)^\gamma \ln(p_t)$$

where  $\gamma$  is a hyperparameter ( $\gamma = 0$  is equivalent to the standard unweighted BCE).

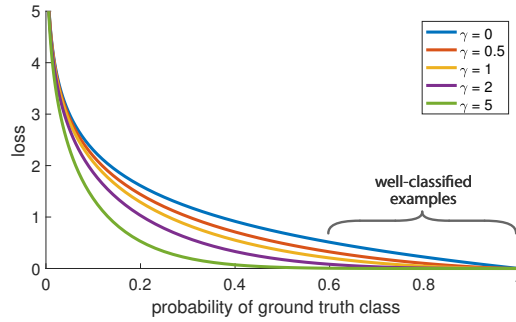


Figure 4.17: Focal loss for varying  $\gamma$

**Example.** Consider a focal loss with  $\gamma = 2$ . The down-weigh factors for varying  $p_t$  are:

| $p_t$ | $(1 - p_t)^\gamma$                   |
|-------|--------------------------------------|
| 0.9   | $(0.1)^2 = 0.01 = \frac{1}{100}$     |
| 0.6   | $(0.4)^2 = 0.16 \approx \frac{1}{6}$ |
| 0.4   | $(0.6)^2 = 0.36 \approx \frac{1}{3}$ |
| 0.1   | $(0.9)^2 = 0.81 \approx \frac{4}{5}$ |

**Binary class weights** Weigh the loss value of the positive class by  $\alpha \in [0, 1]$  and the negative class by  $(1 - \alpha)$ .

Binary class weights

Consider the following notation:

$$\alpha_t = \begin{cases} \alpha & \text{if } y = 1 \\ 1 - \alpha & \text{otherwise} \end{cases}$$

Binary cross-entropy with class weights can be defined as:

$$\text{WBCE}(p_t) = \alpha_t \text{BCE}(p_t) = -\alpha_t \ln(p_t)$$

On the same note,  $\alpha$ -balanced binary focal loss can be defined as:

$$\text{WBFL}_\gamma(p_t) = \alpha_t \text{BFL}_\gamma(p_t) = -\alpha_t (1 - p_t)^\gamma \ln(p_t)$$

**Remark.** Class weights and focal loss are complementary: the former balances the importance of positive and negative classification errors, while the latter focuses on the hard examples of each class.

**RetinaNet loss** RetinaNet is applied to all anchors  $A$  using the  $\alpha$ -balanced binary focal loss for classification and the same bounding box loss as R-CNN:

$$\mathcal{L}^{(i)} = \sum_{j=1}^A \left( \sum_{k=1}^C \text{WBFL}_{\gamma} \left( \sigma(\mathbf{s}_k^{(i,j)}), \mathbf{y}_k^{(i,j)} \right) + \lambda \mathbb{1}[\mathbf{y}^{(i,j)} \neq \bar{\mathbf{0}}] \mathcal{L}_{\text{huber}} \left( \hat{t}^{(i)} - t^{(i)} \right) \right)$$

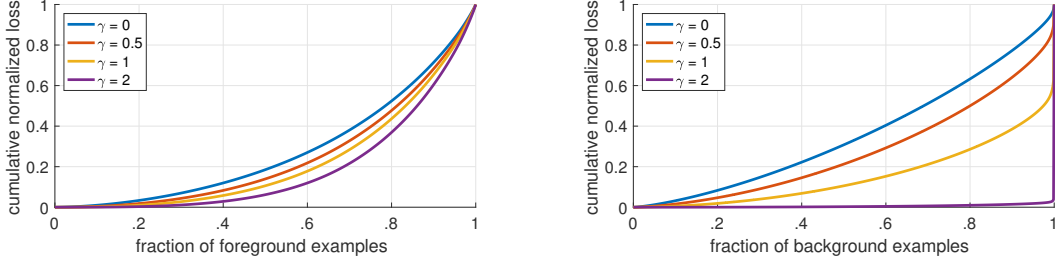


Figure 4.18: Cumulative loss contribution for varying  $\sigma$  of the focal loss.

Note that for the background examples, the contribution to the loss becomes more relevant only when the majority of the samples (i.e., the most difficult ones) has been considered.

**Model initialization** Instead of initializing biases to 0 (i.e., equiprobable classes), set them to account for unbalanced classes.

Model initialization

Consider a newly initialized network with input  $\mathbf{x}$ , weights  $\mathbf{w} \sim \mathcal{N}(\mu = 0, \sigma)$ , and bias  $b$ . The output activation  $s$  is computed as:

$$s = \mathbf{w}\mathbf{x} + b \approx r + b \quad \text{with } r \sim \mathcal{N}(\mu = 0, \sigma)$$

Assume that we want to force the output probability to be  $\psi$ , we have that:

$$\begin{aligned} \sigma(r + b) &= \psi \\ \sigma(b) &= \psi & \text{as } \mathbb{E}[r] = 0 \\ \frac{1}{1 + e^{-b}} &= \psi \\ b &= -\ln \left( \frac{1 - \psi}{\psi} \right) \end{aligned}$$

A reasonable value for  $\psi$  is the empirical ratio of the positive class:

$$\psi = \frac{|\text{positives}|}{|\text{positives}| + |\text{negatives}|}$$

**Example.** Assume an imbalanced dataset with a ratio 1:100 of positive to negative examples. We would like to initialize the bias  $b$  so that the model outputs  $\psi = \frac{1}{100} = 0.01$  (i.e., more likely to classify as negative at the beginning). Therefore, we have that:

$$b = -\ln \left( \frac{1 - 0.01}{0.01} \right) \approx -4.6$$



**Remark.** Anchor-based detectors have the following limitations:

- Anchors are a subset of all possible boxes. Their application is inefficient as they are sort of brute forcing the problem.
- Multiple anchors are viable boxes for the same object and NMS is needed to post-process the predictions.
- During training, the anchor assigned to the ground-truth is selected through hand-crafted rules and thresholds.

#### 4.3.5 Keypoint-based detectors

**CenterNet** Anchor-free object detector based on keypoints.

CenterNet

Given an input image of size  $3 \times H \times W$ , CenterNet outputs:

**Heatmap** A matrix  $\hat{Y} \in [0, 1]^{C \times \frac{H}{R} \times \frac{W}{R}}$  where  $R$  is the output stride (usually small, e.g.,  $R = 4$ ). Each value scores the “keypoint-ness” of that pixel for a given class.

**Offset** A matrix  $\hat{O} \in \mathbb{R}^{2 \times \frac{H}{R} \times \frac{W}{R}}$  that indicates an offset for each point of the heatmap to map them from the strided shape back to the original one.

**Bounding box size** A matrix  $\hat{S} \in \mathbb{R}^{2 \times \frac{H}{R} \times \frac{W}{R}}$  that indicates the width and height of the bounding box originated from each point of the heatmap.



**Architecture** The backbone of CenterNet is a convolutional encoder-decoder (i.e., first downsample with the encoder and then upsample with the decoder) also used for keypoint detection or semantic segmentation. The output of the backbone is fed to three different branches composed of a  $3 \times 3$  convolution, ReLU, and a  $1 \times 1$  convolution.

**Inference** Predictions are determined as follows:

1. Find the local maxima of the heatmap  $\hat{Y}$ .
2. For each local maximum at channel  $c^{(m)}$  and position  $(x^{(m)}, y^{(m)})$  of  $\hat{Y}$ :
  - a) Determine the bounding box center coordinates using the corresponding offsets in  $\hat{O}$ :
 
$$(x^{(m)} + \hat{O}_x^{(m)}, y^{(m)} + \hat{O}_y^{(m)})$$
  - b) Determine the box size using the corresponding values  $(\hat{S}_W^{(m)}, \hat{S}_H^{(m)})$  in  $\hat{S}$ .

**Remark.** NMS is implicit with the local maxima search operation.

**Training** Given a ground-truth keypoint (i.e., the center of a box)  $p = (x_p, y_p)$  of class  $c$ , its coordinates are projected onto the output heatmap coordinate system as  $\tilde{p} = (\lfloor \frac{x_p}{R} \rfloor, \lfloor \frac{y_p}{R} \rfloor)$ .

The target heatmap  $Y_c^{(p)}$  for class  $c$  is created as follows:

$$Y_c^{(p)}[x, y] = \begin{cases} 1 & \text{if } (x, y) = \tilde{p} \\ \exp\left(-\frac{(x-x_{\tilde{p}})^2 + (y-y_{\tilde{p}})^2}{2\sigma_p^2}\right) & \text{otherwise} \end{cases}$$

In other words, to help training, the heatmap at  $\tilde{p}$  is 1 and it smoothly decreases to 0 while moving away from  $\tilde{p}$ .

The loss is the following:

$$\mathcal{L}^{(p)} = \mathcal{L}_{\text{heatmap}}^{(p)} + \mathcal{L}_{\text{box}}^{(p)}$$

where:

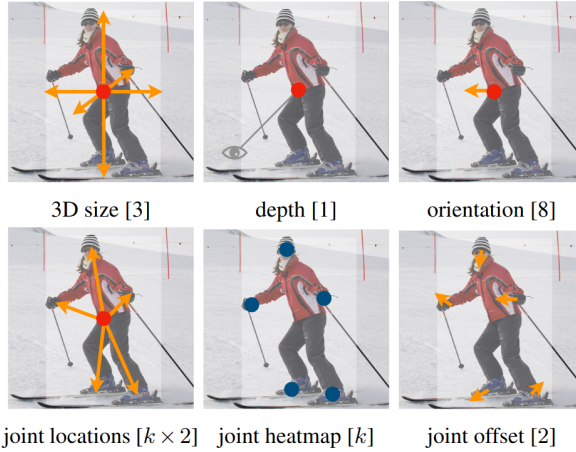
- $\mathcal{L}_{\text{heatmap}}^{(p)}$  is the binary focal loss applied pixel-wise with an additional weighting factor:

$$\mathcal{L}_{\text{heatmap}}^{(p)} = \sum_{x,y,c} \begin{cases} -(1 - \hat{Y}_c[x, y])^\gamma \ln(\hat{Y}_c[x, y]) & \text{if } Y_c[x, y] = 1 \\ -(1 - Y_c[x, y])^\beta \cdot (\hat{Y}_c[x, y])^\gamma \ln(1 - \hat{Y}_c[x, y]) & \text{otherwise} \end{cases}$$

where  $(1 - Y_c[x, y])^\beta$  reduces the loss at positions close to the ground truth.  $\gamma$  and  $\beta$  are hyperparameters ( $\gamma = 2$  and  $\beta = 4$  in the original paper).

- $\mathcal{L}_{\text{box}}^{(p)}$  is the Huber loss to compare bounding boxes.

**Remark.** CenterNet can solve other tasks such as 3D bounding box estimation or pose estimation.



**Remark.** Keypoints can be seen as a special case of anchors, but:

- They are only based on location and not overlap.
- They do not rely on manual thresholds.
- There is no need for NMS.

However, CenterNet still relies on some hand-crafted design decisions such as the windows size to detect local maxima and the output stride.

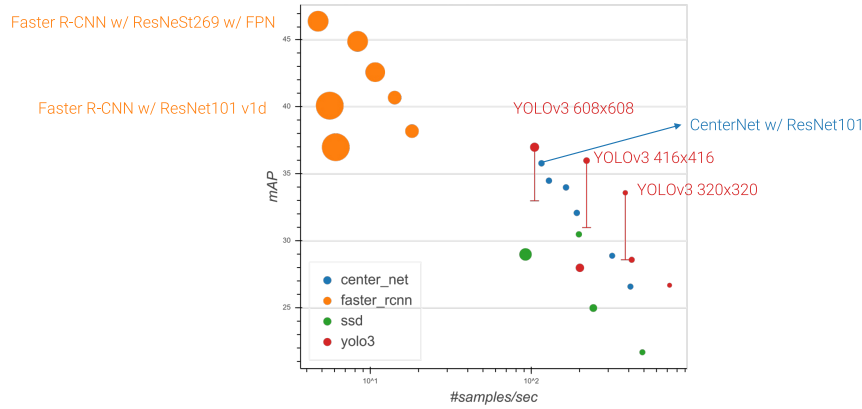


Figure 4.19: mAP – speed comparison of the various object detection approaches

### 4.3.6 Transformer-based detectors

**Detection transformer (DETR)** Method based on two ideas:

- Use transformers to predict a set of objects in a single pass (i.e., solve a set prediction problem).
- Use the Hungarian (i.e., bipartite matching) loss as set prediction loss that forces a unique matching between predictions and ground-truths.

Detection transformer (DETR)

**Parallel decoding** Generate all outputs of a decoder in a single pass (instead of doing it autoregressively).

Parallel decoding

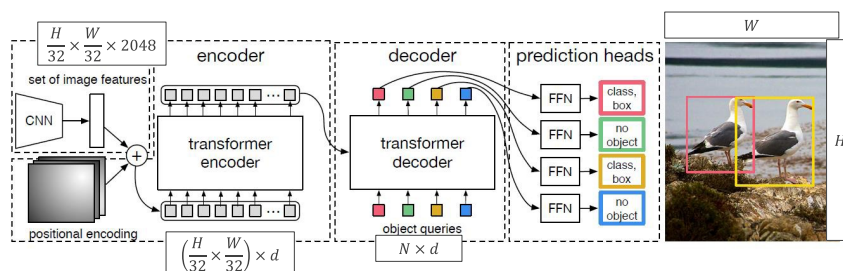
**Object queries** Learned positional encodings that are fed to the decoder.

Object queries

**Remark.** In object detection, there are no other generated output tokens to feed into the decoder for parallel decoding. Therefore, only the learned positional encodings are used.

**Architecture** Given an input image  $I$  of size  $H \times W \times 3$ , DETR does the following:

1. Pass the input  $I$  through a CNN feature extractor to obtain an activation  $A$  of shape  $\frac{H}{32} \times \frac{W}{32} \times 2048$ .
2. Use a  $1 \times 1$  convolution to adjust the number of channels of  $A$  from 2048 to  $d$ .
3. Add positional encoding to  $A$ .
4. Pass the activation  $A$  through the transformer encoder to obtain the keys and values for the decoder.
5. Pass the learned object queries through the decoder to obtain the outputs  $O_i$ .
6. Pass each output  $O_i$  through an MLP to obtain class and box predictions.



**Hungarian loss** Consider for simplicity a problem with 2 classes (plus background).  
Given:

Hungarian loss

- $N$  predictions  $\{\hat{y}_i = (\hat{p}_i, \hat{b}_i)\}_{i=1}^N$  where  $\hat{p}_i$  is the class probability distribution and  $\hat{b}_i$  describes the bounding box normalized w.r.t. the image size.
- $O$  ground-truth boxes padded to  $N$  with background classes  $\{\hat{y}_i = (c_i, b_i)\}_{i=1}^O \cup \{\hat{y}_i = \emptyset\}_{i=N-O}^N$  where  $c_i$  is the class,  $b_i$  describes the bounding box normalized w.r.t. the image size, and  $\emptyset$  represents the background class.

The Hungarian loss is defined in two steps:

1. Solve the bipartite matching problem of finding the optimal permutation  $\sigma^*$  that associates each prediction to a unique ground-truth box while minimizing the matching loss  $\mathcal{L}_{\text{match}}$  defined as follows:

$$\mathcal{L}_{\text{match}}(\hat{y}_i, y_j) = \begin{cases} -\hat{p}_i[c_j] + \mathcal{L}_{\text{box}}(\hat{b}_i, b_j) & \text{if } c_j \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

where  $\mathcal{L}_{\text{box}}$  is a loss based on the linear combination of the Huber loss and IoU.

The overall problem is the following:

$$\sigma^* = \arg \min_{\sigma} \sum_{i=1}^N \mathcal{L}_{\text{match}}(\hat{y}_{\sigma(i)}, y_i)$$

2. Given the optimal permutation  $\sigma^*$ , compute the loss as:

$$\mathcal{L}_{\text{hungarian}}(\hat{y}, y) = \sum_{i=1}^N \left( -\ln(\hat{p}_{\sigma^*(i)}(c_i)) + \mathbb{1}[c_i \neq \emptyset] \mathcal{L}_{\text{box}}(\hat{b}_{\sigma^*(i)}, b_i) \right)$$

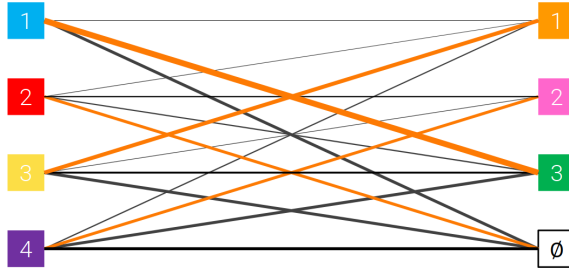


Figure 4.20: Possible permutations and optimal permutation (in orange).

**Remark.** Results show that faster R-CNN + FPN is better for smaller object and DETR performs best with larger objects.

**Remark (Visualization).** By analyzing the main components of DETR, the following can be observed:

**Encoder** The encoder tend to solve a segmentation problem (i.e., determine what the object is).

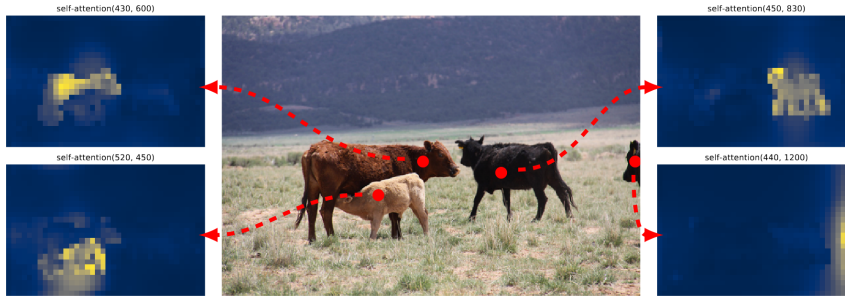


Figure 4.21: Self-attention map of some pixels at the last encoder. Yellow tiles indicate that the analyzed pixel attends to that patch.

**Decoder** The decoder tend to attend at object boundaries (i.e., determine where the object is).

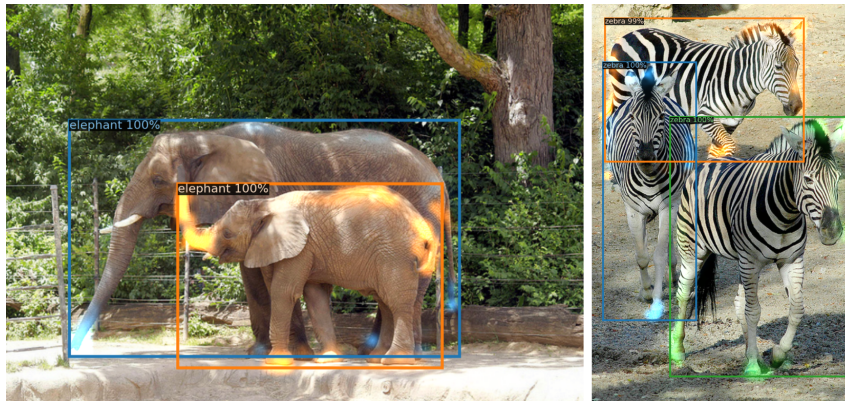


Figure 4.22: Decoder attention. Highlighted areas have a higher attention weight.

**Object query** Each object query tend to be specialized in recognizing objects in specific areas.

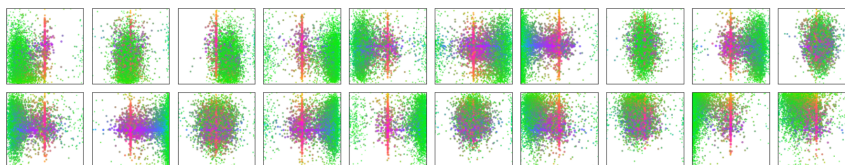


Figure 4.23: Position of the predictions of each object query. Green dots represent small boxes, red large horizontal boxes, and blue large vertical boxes.

## 4.A Appendix: EfficientDet

**Remark.** When working with object detection, there are many options to scale a model:

**Backbone** Change the CNN to refine the input image.

**Image resolution** Change the resolutions produced by the CNN.

**Multi-scale feature representation** Change the architecture of the multi-scale detector.

**Detector head** Change the classification and regression heads.

**EfficientDet** Similarly to MobileNet and EfficientNet, EfficientDet uses a compound scaling coefficient  $\phi$  to scale up the model (using heuristic rules).

EfficientDet

**Remark.** Before EfficientDec, other multi-scale feature representations have been introduced:

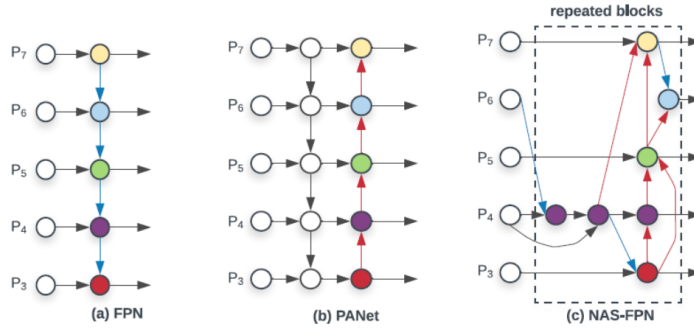
**FPN** As described in Section 4.3.3.

**PANet** Based on FPN with an additional bottom-up path to merge higher resolution features with coarser ones.

PANet

**NAS-FPN** Base block found using neural architecture search. Multiple blocks are repeated to obtain the multi-scale features.

NAS-FPN



**Weighted bi-directional feature pyramid network (BiFPN)** Architecture to represent multi-scale features with the following characteristics:

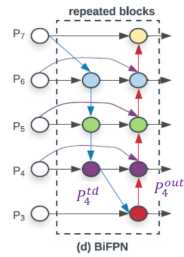
Weighted  
bi-directional feature  
pyramid network  
(BiFPN)

- Applied as repeated blocks (i.e., as NAS-FPN).
- Based on PANet with the following changes:
  - Activations generated by only one parent activation are removed and the connections are adjusted.
  - Add a connection between input and output node of a block.
  - Use depth-wise separable convolutions.
  - Add learnable weights to weigh features at different resolutions.

The output  $P_i^{\text{out}}$  at the  $i$ -th scale is therefore computed as:

$$P_i^{\text{out}} = \text{conv} \left( \frac{w_{i,1}P_i^{\text{in}} + w_{i,2}P_i^{\text{top-down}} + w_{i,3}\text{downsample}(P_{i-1}^{\text{out}})}{w_{i,1} + w_{i,2} + w_{i,3} + \varepsilon} \right)$$

$$P_i^{\text{top-down}} = \text{conv} \left( w'_{i,1}P_i^{\text{in}} + w_{i+1,\text{td}}\text{upsample}(P_{i+1}^{\text{top-down}}) \right)$$





# 5 Segmentation

## 5.1 Semantic segmentation

**Semantic segmentation** Given an input image, output a category for each pixel.

Semantic  
segmentation  
Pixel-wise IoU

**Pixel-wise IoU** IoU generalized to pixel-wise segmentation masks. Given a class  $c$ , the ground-truths  $y^{(i)}$ , and predictions  $\hat{y}^{(i)}$ , the IoU w.r.t.  $c$  is computed as follows:

$$TP_c = \sum_{i=1} |\text{pixels } (u, v) : y_{(u,v)}^{(i)} = c \wedge \hat{y}_{(u,v)}^{(i)} = c|$$
$$\text{IoU}_c = \frac{TP_c}{\sum_{i=1} (|(u, v) : \hat{y}_{(u,v)}^{(i)} = c| + |(u, v) : y_{(u,v)}^{(i)} = c|) - TP_c}$$

The mean IoU is computed as:

$$\text{mIoU} = \frac{1}{C} \sum_{c=1}^C \text{IoU}_c$$

**Remark.** Mean IoU is not strictly correlated to human's perception of segmentation. A small gain in mIoU might correspond to a large visual improvement.

### 5.1.1 Kinect human pose estimation

**Human pose detection** Task of detecting the position and orientation of a person.

Human pose  
detection

**Pipeline** Kinect pose detection is done in three phases:

1. Capture a depth image and remove the background to obtain the depth map of a person.
2. Classify each pixel into a body part.
3. Determine the position of the joints (i.e., skeleton) by finding local modes (i.e., center of mass).

**Synthetic annotated data** The data used to create the model is artificially generated. 100k poses were captured using motion capture devices. Then, different mock-up body models (for which the ground-truth is known) were simulated and recorded through a virtual camera with the same intrinsic parameters of the Kinect camera. This allows to obtain robustness with different body and clothing shapes.

Synthetic annotated  
data

**Remark.** The workflow of the original paper consists of iteratively training the model and creating new training data for poses that the current model struggles on.



**Depth comparison features** Given a depth image  $D$  and the offsets  $\theta = (\Delta p, \Delta n)$ , each pixel  $x$  of  $D$  produces a feature as follows:

Depth comparison features

$$f(x; D, (\Delta p, \Delta n)) = D \left[ x + \frac{\Delta p}{D[x]} \right] - D \left[ x + \frac{\Delta n}{D[x]} \right]$$

In other words, each  $x$  is described by the difference in depth between two points offset from  $x$ . The depth at background pixels is a large positive number.

**Remark.** As real-time processing is required, this approach allows to quickly compute features to discriminate parts of the body. However, it does not always produce a correct response.

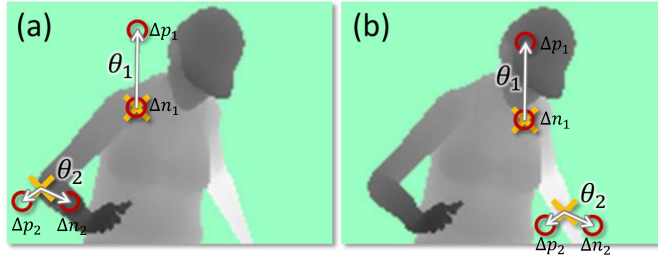
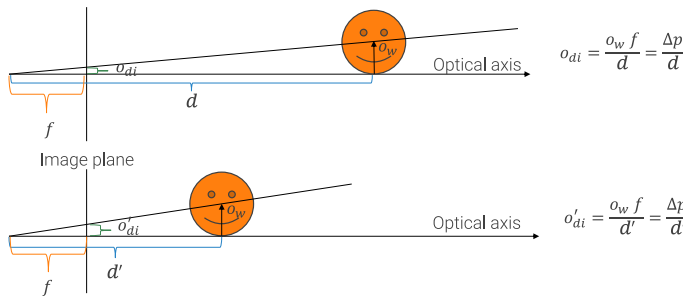


Figure 5.1: Examples of feature computation

**Depth-invariant offsets** The denominator ( $D[x]$ ) applied to the offsets allows obtaining depth-invariant offsets.

Consider two objects at different depths. The focal length of the camera is  $f$  and the world offset we want to apply is  $o_w$ . By changing depth  $d$ , the offset in the image plane  $o_{di}$  changes due to the perspective projection rules. Therefore, to obtain the offset in the image place, we have that:

$$o_{di} : f = o_w : d \Rightarrow o_{di} = \frac{o_w f}{d} = \frac{\Delta p}{d}$$



**Decision tree** Depth comparison features are used to train decision trees.

Decision tree

**Remark.** Decision trees are unstable robust classifiers. They are able to achieve good performance but significantly change in structure if the training data is slightly perturbed (i.e., high variance).

**Random forest** Ensemble of  $N$  decision trees that aims to reduce variance by averaging their predictions.

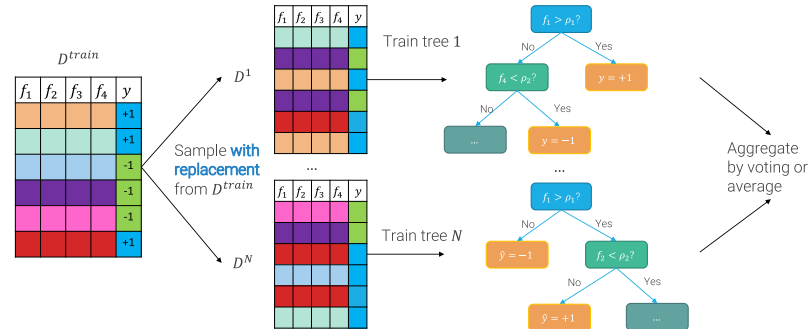
Random forest



**Remark.** For a random forest to be effective, its decision trees should be uncorrelated so that when averaging, the average of their errors tend to 0.

**Bootstrap aggregating (bagging)** Train each decision tree using a replica of the training set obtained by sampling with replacement.

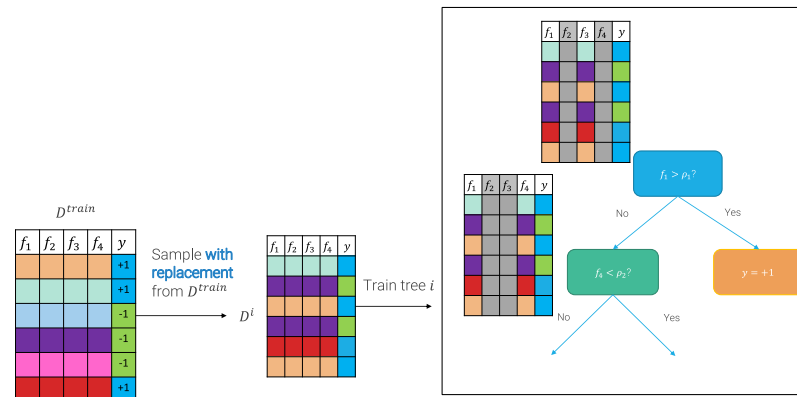
Bootstrap aggregating (bagging)



**Random splitting** Even though bagging reduces variance, if there is a subset of particularly predictive features, the resulting trees will be correlated.

Random splitting

To avoid this, in a random forest, tree splitting is done on a different random subset of features each time.



**Remark.** Random forests are:

- Fast and parallelizable in both training and inference.
- Robust to hyperparameters change.
- Interpretable.

### 5.1.2 R-CNN

**R-CNN for segmentation** Slide a window across each pixel of the input image. Each slice is passed through an R-CNN to determine the class of the pixel at the center of the window.

R-CNN for segmentation

The loss for an example  $i$  is the sum of the cross-entropy losses at each pixel  $(u, v)$ :

$$\mathcal{L}^{(i)} = \sum_{(u,v)} \mathbb{1} \left( c_{(u,v)}^{(i)} \right) \mathcal{L}_{\text{CE}} \left( \text{softmax}(\text{scores}_{(u,v)}) \right)$$

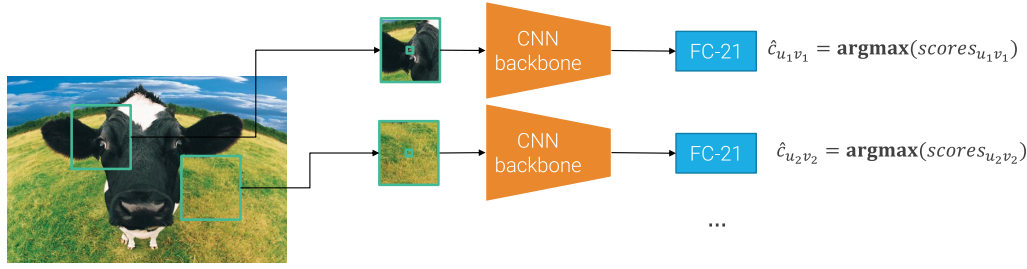


Figure 5.2: R-CNN for segmentation with 20 (+1) classes

### 5.1.3 Fully convolutional network

**Standard up-sampling** Non-learned operator to increase the spatial dimension of the input image. Possible approaches are:

**Nearest neighbor** Fill the new pixels with the value of the nearest one.

**Bilinear interpolation** Fill the new pixels by interpolating the nearest ones.

**Fully convolutional network (FCN)** Pass the whole input image into a CNN and use the activation to determine the class of each pixel. The flow is the following:

**Backbone** Pass the input image through a CNN.

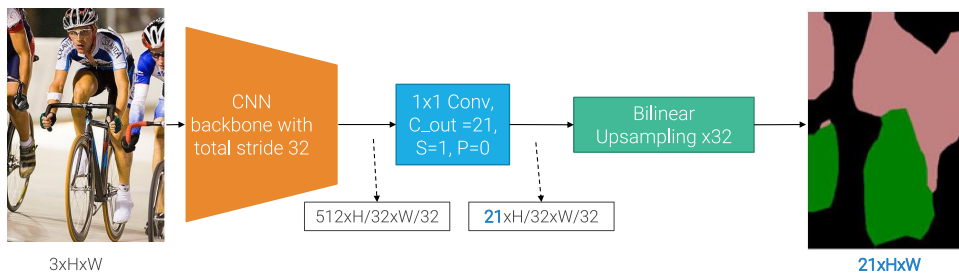
**Scoring layer** Apply a  $1 \times 1$  convolution on the output activation to obtain the correct number of channels (one per class).

**Up-sampling** Apply an up-sampling operator to restore the input spatial dimension.

**Remark.** Without learned non-linear up-sampling operators, the output mask is dependent on the total stride of the CNN. Therefore, a coarse final activation will result in a coarse segmentation mask.

**Pyramid of FCNs** Stack of FCNs with skips (i.e., merges) to up-sample multiple activations at different resolutions.

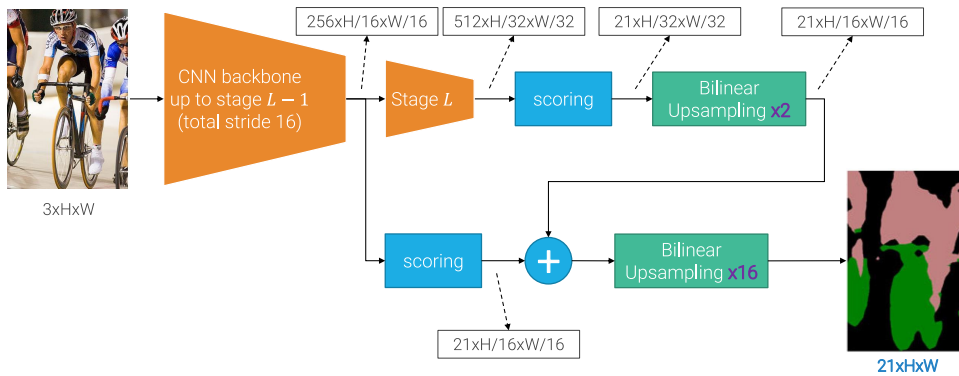
**FCN-32S** FCN that applies the backbone CNN up until the last layer  $L$  obtaining a total stride of 32. This results in a very coarse mask.



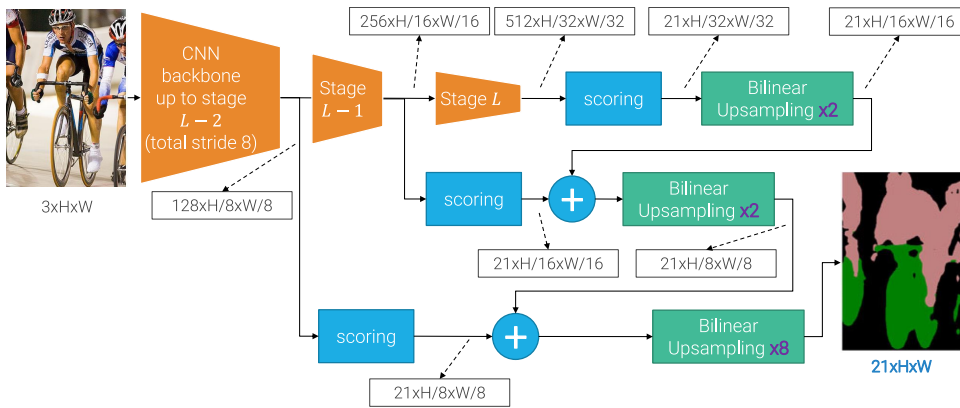
**FCN-16S** FCN that applies the backbone CNN up until the layer  $L - 1$  and then branches into two paths:

- A branch continues into the  $L$ -th layer as in FCN-32S. Up-sampling is done to match the spatial dimension of the other branch so that it can be used as a skip.

- A branch passes through a scoring layer and is summed to the output of the other branch before up-sampling to the image spatial dimension.



**FCN-8S** As FCN-16S where the skip comes from FCN-16S.



**Remark.** Ablation study found out that:

- Results do not show significant improvements after stride 8 (i.e., the initial stages of the backbone CNN are less relevant for the segmentation task).
- Fine-tuning the backbone is important.
- There is no significant difference in training end-to-end (i.e., directly FCN-8S) and coarse-to-fine (i.e., progressively train starting from FCN-32S up to FCN-8S)
- Skips between resolutions to merge them are important.

#### 5.1.4 U-Net

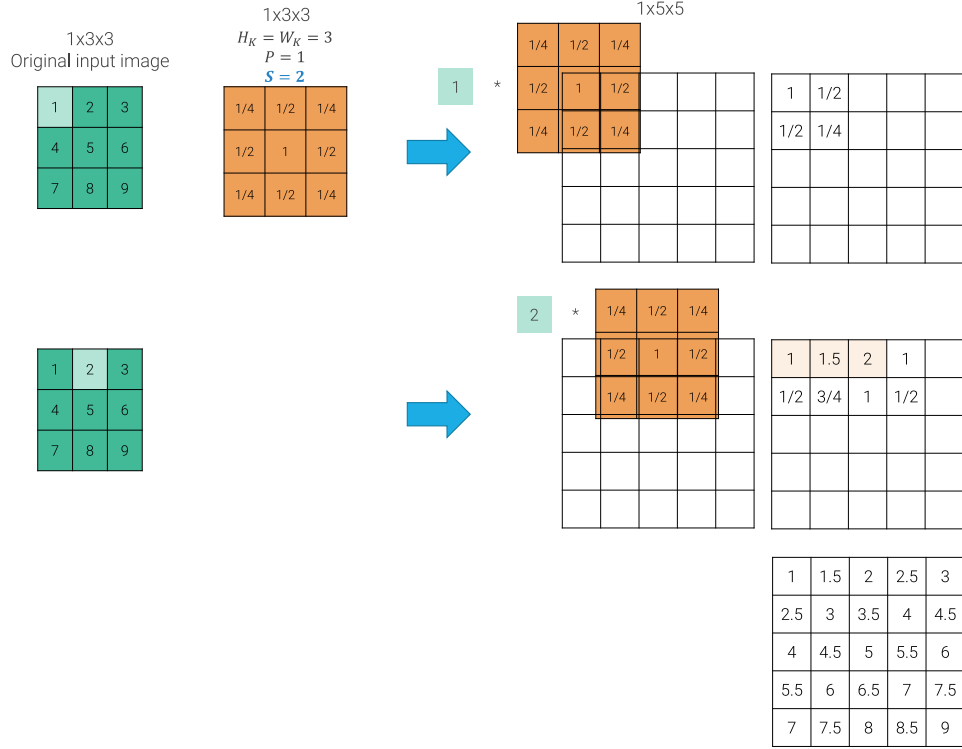
**Transposed convolution** Operator to invert the spatial down-sampling of a convolution.

Transposed convolution

Given a convolution with convolutional matrix  $K$ , a transposed convolution is obtained by applying  $K^T$  to the image.

In practice, a transposed convolution can be obtained by sliding the kernel on the output activation (instead of input). The values at the output activation correspond to the product between the input pixel and the kernel. If multiple kernels overlap at the same output pixel, its value is obtained as the sum of all the values that end up in that position.

**Example.** Consider images with 1 channel. Given a  $3 \times 3$  input image and a  $3 \times 3$  transposed convolution kernel with stride 2, the output activation has spatial dimension  $5 \times 5$  and is obtained as follows:



**Remark.** A transposed convolution is usually initialized as a bilinear interpolator. For instance, a  $3 \times 3$  kernel is initialized as:

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$$

**Remark.** Transposed convolutions can be seen as convolutions with a fractional stride.

**Remark.** Transposed convolutions are also called deconvolutions (this is technically wrong as deconvolutions are the full inverse of convolutions), up-convolutions, fractionally/backward strided convolutions.

**Remark.** In generative tasks, transposed convolutions generate checkerboard artifacts. Darker pixels are due to the fact that they result from the summation of multiple kernel applications.

**U-Net** Encoder-decoder architecture for segmentation:

U-Net

**Encoder** Backbone CNN that down-samples the input image.

**Decoder** Symmetric structure to the encoder that up-samples the activation using transposed convolutions and further refines them with normal  $3 \times 3$  convolutions.

Each level of the encoder is connected to its corresponding level in the decoder through a skip connection that concatenates the activations.

The scoring layer is only applied at the end to adjust the number of channels.

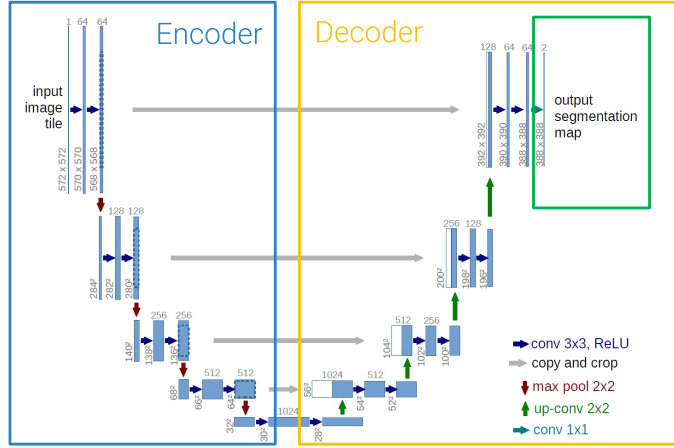


Figure 5.3: U-Net structure. Note that in the original paper, convolutions have padding `valid` and the input is provided from a sliding window. Modern implementations have padding `same` and same input and output spatial dimension.

**Remark.** Intuitively, the two components of a U-Net do the following:

**Encoder** Determine what is in the image.

**Decoder** Determine where the objects are.

### 5.1.5 Dilated CNN

**Remark.** The standard classification backbone approach has some shortcomings:

- Predictions made using the activation at higher layers are semantically rich but spatially coarse.
- Predictions made using the activation at lower layers are semantically poorer but have a higher spatial resolution.

**Dilated/atrous convolution** Given a dilation rate  $r$ , a dilated convolution is equivalent to applying a kernel with an  $r - 1$  gap between weights.

Dilated/atrous convolution

Formally, a dilated convolution with kernel  $K$  and dilation rate  $r$  applied on an image  $I$  is computed as follows:

$$[K * I](j, i) = \sum_{n=1}^C \sum_m \sum_l K_n(m, l) I_n(j - r \cdot m, i - r \cdot l) + b$$

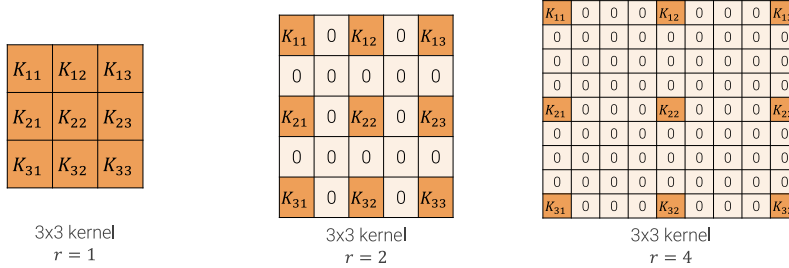
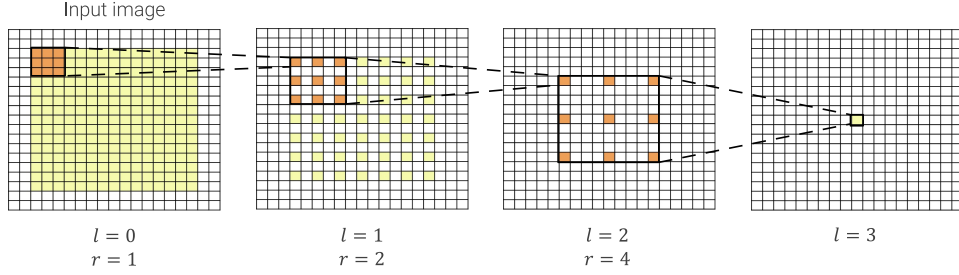


Figure 5.4: Example of  $3 \times 3$  dilated convolutions with increasing dilation rate

**Remark.** By stacking dilated convolutions with exponentially increasing dilation rate  $r_l = 2^l$ , it is possible to achieve the following effects:

- Exponential growth in receptive field. At the  $l$ -th level the receptive field is  $(2^{l+1} - 1) \times (2^{l+1} - 1)$ .
- Linear growth in the number of parameters.
- Unchanged resolution.



**Dilated ResNet** ResNet composed of standard stages up until a certain layer. The remaining ones are dilated stages.

Dilated ResNet

**Dilated bottleneck residual block** Standard bottleneck residual block composed of convolutions with kernels  $1 \times 1 \mapsto 3 \times 3 \mapsto 1 \times 1$  where the middle  $3 \times 3$  convolution is a dilated convolution.

**Dilated stage** Given a dilation rate  $r$ , a dilated stage is built as follows:

- The first bottleneck block has stride 1 (instead of 2) and dilation rate  $\frac{r}{2}$ .
- The remaining blocks have stride 1 and dilation rate  $r$ .

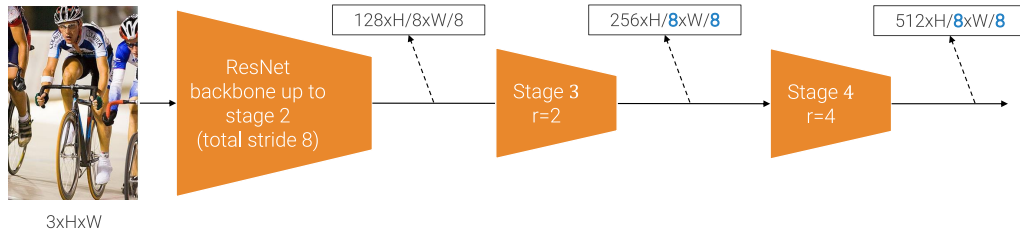
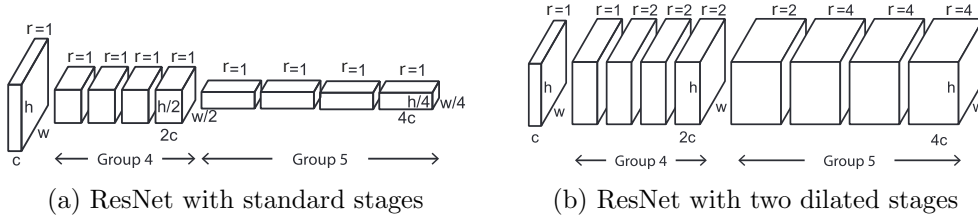


Figure 5.6: Dilated ResNet with total stride 8

**Remark.** In principle, it is possible to process the input image without altering the resolution. However, this approach is computationally more expensive as the activations are larger. Therefore, the first standard ResNet stages are kept to apply some stride. As in FCN, a total stride of 8 (up to the second stage) or 16 (up to the third stage) is used.

**DeepLab v3** Architecture based on dilated ResNet that considers objects at multiple scales.

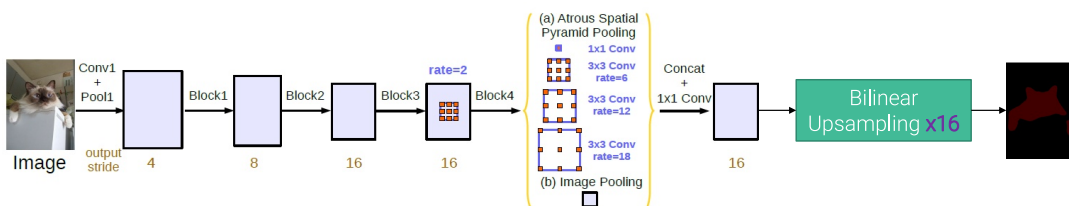
DeepLab v3

**Atrous spatial pyramid pooling (ASPP)** Module that takes as input the activation of the dilated ResNet backbone and applies separate  $3 \times 3$  convolutions with large dilation rates to encode spatial context. The final output is the concatenation of the activations of each convolution.

Atrous spatial pyramid pooling (ASPP)

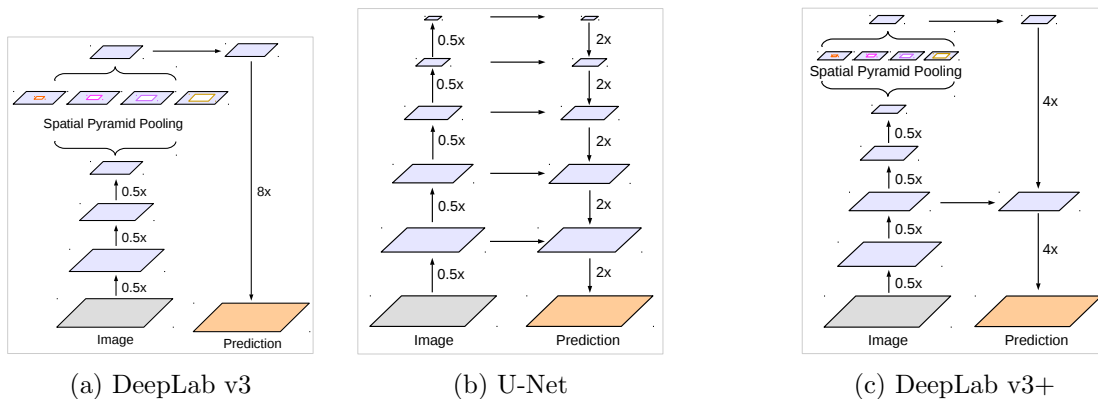
**Curriculum training** Training is done in two steps:

1. Train with a total stride of 16 (i.e., only the last ResNet stage is dilated). This allows for a faster training with larger batches.
2. Fine-tune with a total stride of 8 (i.e., the last two stages are dilated). This results in spatially larger and more detailed activations.



**DeepLab v3+** Architecture based on the ASPP of DeepLab v3 and the decoder of U-Net.

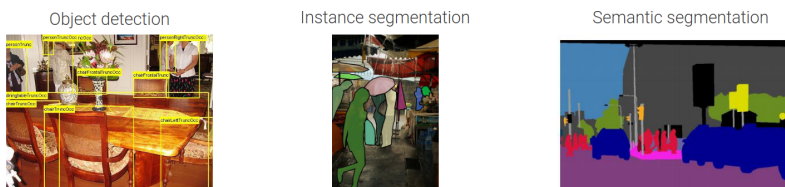
DeepLab v3+



## 5.2 Instance segmentation

**Instance segmentation** Task of segmenting and classifying all instances of the objects of interest (i.e., intersection between object detection and semantic segmentation).

Instance segmentation



### 5.2.1 Mask R-CNN

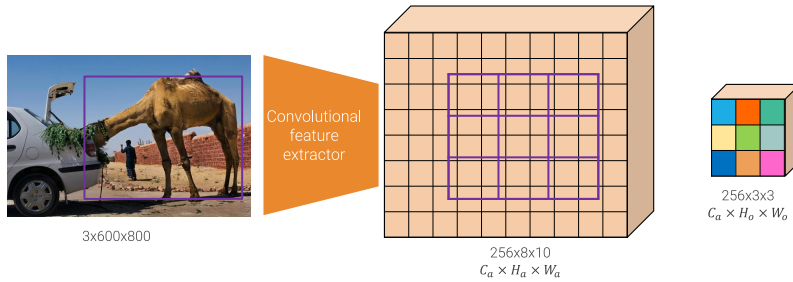
**Mask R-CNN** Architecture based on faster R-CNN with a modification to RoI pool and the addition of a CNN head to predict the segmentation mask.

Mask R-CNN

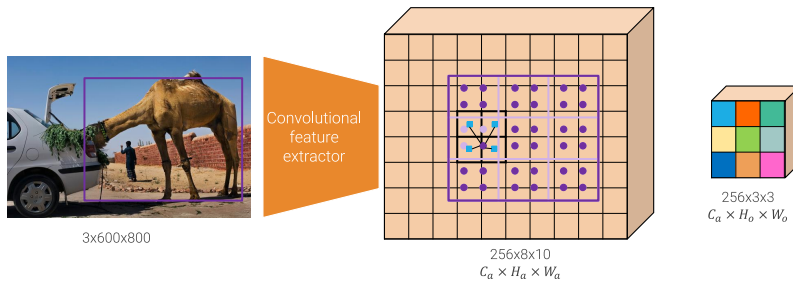
**RoI align** Modification to RoI pool to avoid quantization. It works as follows:

RoI align

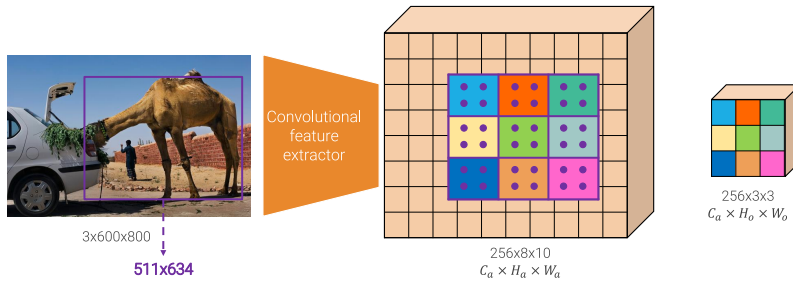
1. Divide the proposal into equal subregions without snapping to grid.



2. Sample some values following a regular grid within each subregion. Use bilinear interpolation to determine the values of the sampled points (as they are most likely not be pixel-perfect).



3. Max or average pool the sampled values in each subregion.

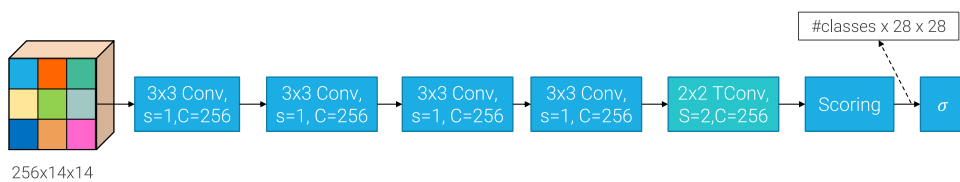


**Mask head** Fully-convolutional network that takes the output of RoI align and predicts a binary mask with resolution  $28 \times 28$  for each class. It is composed of convolutions, a transposed convolution, and a scoring layer.

Mask head

In practice, the bounding box predicted by the standard R-CNN flow is used to determine how to warp the segmentation mask onto the original image.

**Remark.** Empirical experiments show that solving a multi-label problem (i.e., multiple sigmoids) is better than a multi-class problem (i.e., softmax).





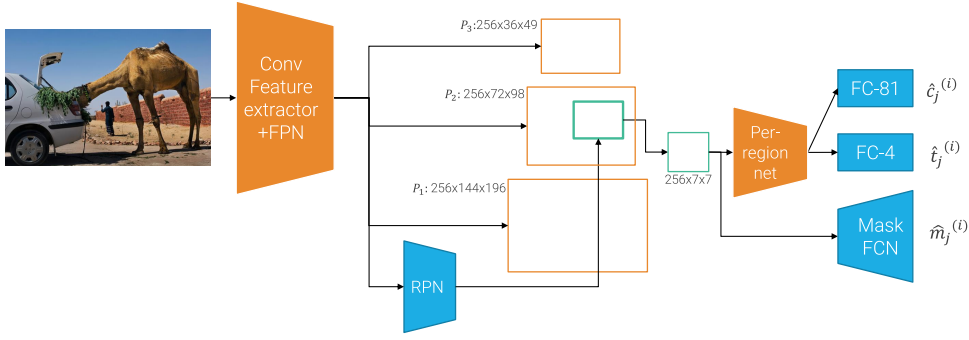


Figure 5.8: Overall architecture of mask R-CNN

**Training** The R-CNN flow is trained in the standard way.

Given the ground-truth class  $c$  and mask  $m$ , and the predicted mask  $\hat{m}$ , the mask head is trained using the following loss:

$$\mathcal{L}_{\text{mask}}(c, m) = \frac{1}{28 \times 28} \sum_{u=0}^{27} \sum_{v=0}^{27} \mathcal{L}_{\text{BCE}}(\hat{m}_c[u, v], m[u, v])$$

In other words, the ground-truth mask is compared against the predicted mask for the correct class.

**Inference** The class prediction of R-CNN is used to select the correct channel of the mask head. The bounding box of R-CNN is used to decide how to warp the segmentation mask onto the image.

**Remark.** By attaching a different head to predict keypoints, this framework can be adapted for human pose estimation.

## 5.3 Panoptic segmentation

**Things** Countable object categories that can be split into individual instances.

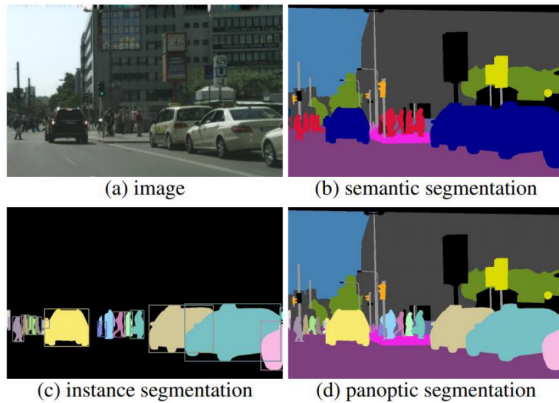
Things

**Stuff** Uncountable amorphous regions (e.g., background)

Stuff

**Panoptic segmentation** Task of classifying each pixel of the image. Objects of interest (i.e., things) are treated as in instance segmentation. Background and non-relevant textures (i.e., stuff) are treated as in semantic segmentation.

Panoptic segmentation



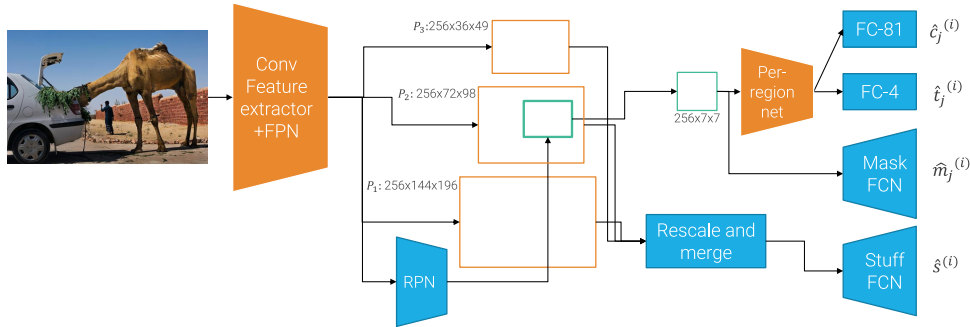
### 5.3.1 Panoptic feature pyramid network

**Panoptic feature pyramid network** Modification of mask R-CNN with an additional path for stuff prediction that works as follows:

Panoptic feature pyramid network

1. Rescale and merge FPN feature maps.
2. Pass the result through a fully-convolutional network to predict stuff masks.

When generating the predictions, the mask head (for things prediction) has the priority over the stuff head.



### 5.3.2 MaskFormer

**Remark.** With convolutions, semantic segmentation has been solved by classifying each pixel into a class. On the other hand, instance and panoptic segmentation partition the image into masks and classify them.

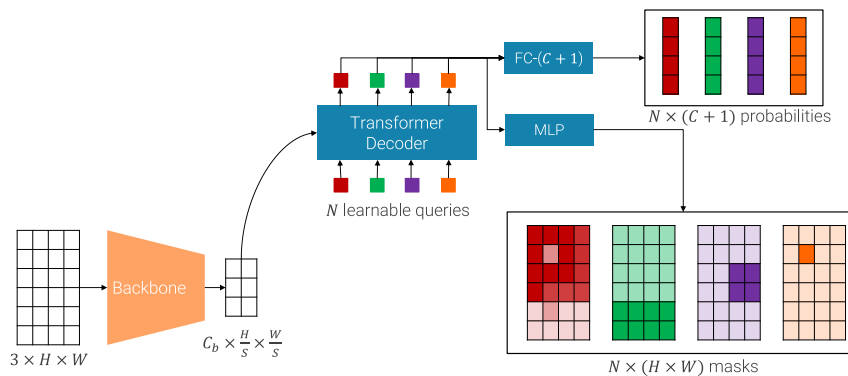
It is intuitive to show that solving a mask classification problem can solve all types of segmentation.

**MaskFormer** Modification of DETR for pixel-wise predictions.

MaskFormer

**Architecture (naive)** An additional path is inserted at the output of the transformer decoder that passes each of its output into a multi layer perceptron to predict a binary mask.

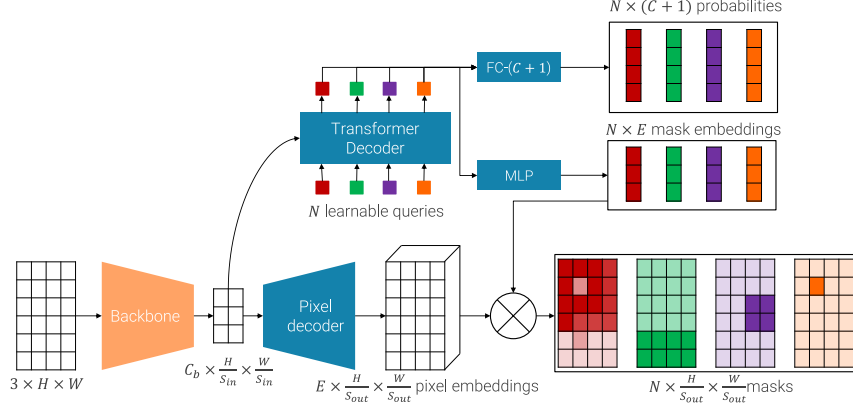
**Remark.** This approach attempts to predict full-resolution masks from spatially coarse features.



**Architecture (pixel decoder)** The following operations are added:

1. A pixel decoder is used to compute pixel-wise embeddings from the output of the CNN backbone.

2. An MLP is added to compute mask embeddings from the outputs of the transformer decoder.
3. The dot product between mask and pixel embeddings allows to compute the output binary masks.



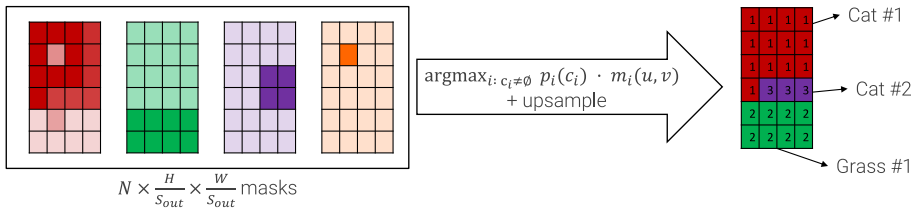
**Inference** Given the output class probabilities  $\mathbf{p}_1, \dots, \mathbf{p}_N$  and masks  $\mathbf{m}_1, \dots, \mathbf{m}_N$ , inference is done as follows:

1. Determine the class  $c_i$  of the  $i$ -th mask from the distribution  $\mathbf{p}_i$ .
2. Classify each pixel  $(u, v)$  as follows:
  - a) For each mask  $i$  that is not background, compute the following score:

$$s_i = \mathbf{p}_i(c_i) \cdot \mathbf{m}_i(u, v)$$

- b) Select the mask with the highest score  $s_i$  as the one associated to the pixel.

**Remark.** Up-sampling might be needed to match the shape of the mask to the original image.



**Remark.** Although it is able to solve all types of segmentation, MaskFormer do not have state-of-the-art results and it is hard to train.

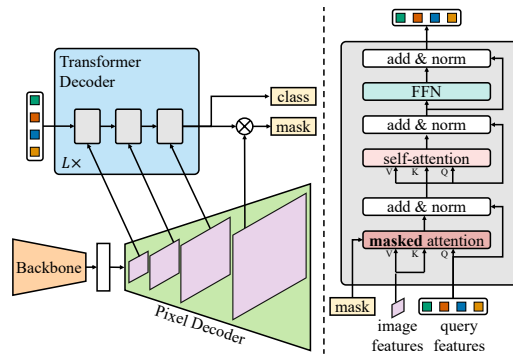
### 5.3.3 Masked-attention mask transformer (Mask2Former)

**Mask2Former** Modification of MaskFormer with the addition of:

Mask2Former

- Masked attention in the transformer decoder for faster training and better results.
- Multi-scale high-resolution features in the pixel-decoder for multi-scale detection.

- Training speed-up by not supervising all pixels (i.e., not all pixels of the mask are trained at once, like a sort of dropout).



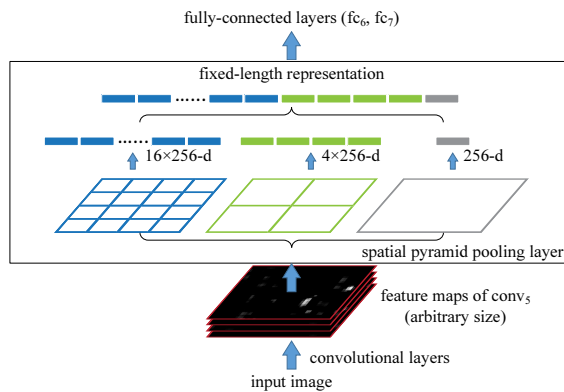
## 5.A Appendix: Spatial pyramid pooling layer

**Spatial pyramid pooling layer (SPP)** Method to obtain a representation with a fixed-resolution containing different scales.

Spatial pyramid pooling layer (SPP)

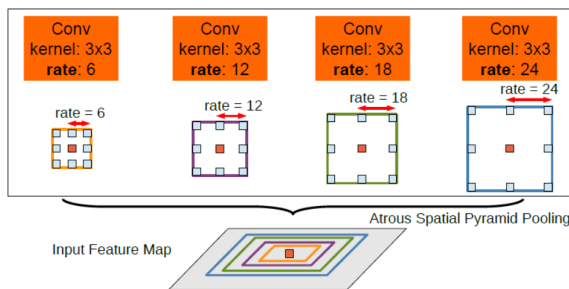
The input image is first passed through the CNN backbone. Then, the feature map is max-pooled with a fixed number of variable-size windows before being flattened.

**Remark.** This can be seen as an extension of global average pooling to preserve spatially localized information.

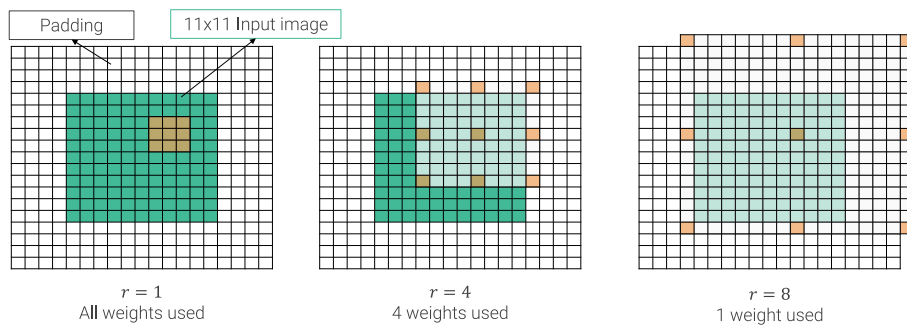


### 5.A.1 DeepLab v2 ASPP

ASPP in DeepLab v2 emulates SPP through dilated convolutions with increasing rate that are concatenated and aggregated through a  $1 \times 1$  convolution.



**Remark.** When the dilation rate grows, the actual number of relevant weights that are not applied to the padding decreases.



### 5.A.2 DeepLab v3 ASPP

ASPP on DeepLab v3 is formed by dilated convolutions and a path that computes global image features to emulate larger dilation rates (i.e., avoid wasting computation on padding with dilated convolutions with a large rate).

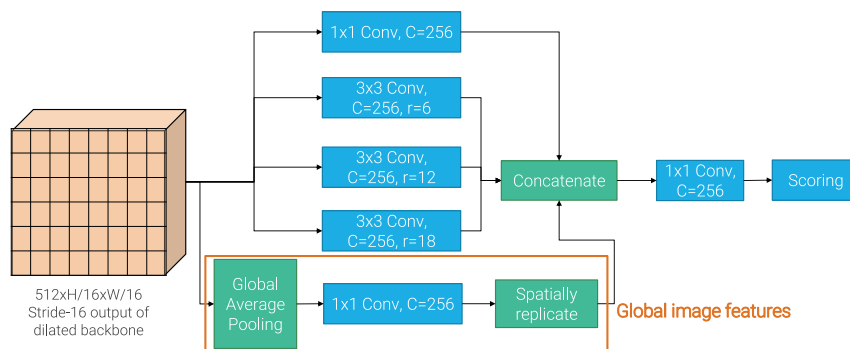


Figure 5.9: ASPP with stride-16. With stride-8, rates are doubled.

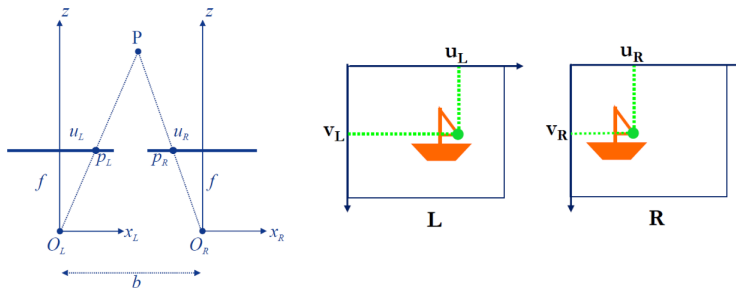
## 6 Depth estimation

**Stereo correspondence** Given an ideal stereo setup, the depth of each point in the world can be obtained by solving a correspondence problem along rows. Given two points  $(u_L, v_L)$  and  $(u_R, v_R = v_L)$  in the left and right image, respectively, representing the projection of the same 3D point, its distance  $z$  from the camera can be determined using the disparity  $d$ :

$$d = u_L - u_R$$

$$z = \frac{bf}{d}$$

where  $b$  is the baseline (i.e., camera distance) and  $f$  is the focal length.



Stereo  
correspondence

**Remark.** Due to the lack of data, existing models are trained on synthetic data and fine-tuned afterwards on real data.

### 6.1 Monocular depth estimation

**Monocular (single-view) depth estimation** Reconstruct the 3D structure of a scene from a single image.

Monocular  
(single-view) depth  
estimation

**Remark.** In principle, this is an ill-posed problem. However, humans are able to solve it through learning.

**Remark.** Traditional supervised frameworks (e.g., encoder-decoder) to solve monocular depth estimation have some limitations:

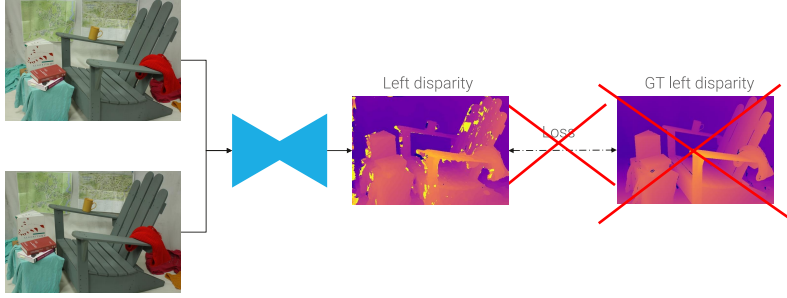
- They require a large amount of realistic synthetic data.
- They require expensive hardware for depth measurement when fine-tuning on real data.

#### 6.1.1 Monodepth

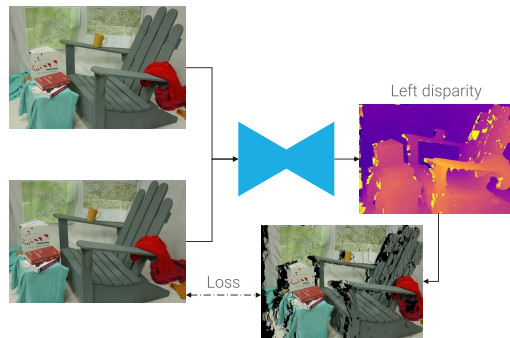
##### Supervised stereo pipeline

Supervised stereo  
pipeline

**Naive approach** A possible solution for depth estimation is to feed a CNN with a pair of synchronized images and make it predict the left (or right) disparity. However, this approach requires to know the ground-truth disparity, which is expensive to obtain.



**Reconstruction approach** Make the model predict the left disparity which is then used to reconstruct the right image. This works as a pixel  $(u, v)$  in the left image with disparity  $\tilde{d}$  should appear as the same to the pixel  $(u + \tilde{d}, v)$  in the right image.



**Monodepth** Network that takes as input the left (or right) image of a stereo vision system and predicts the left (or right) disparity.

**Training (naive)** Once the left disparity has been predicted, it is used to reconstruct the right image.

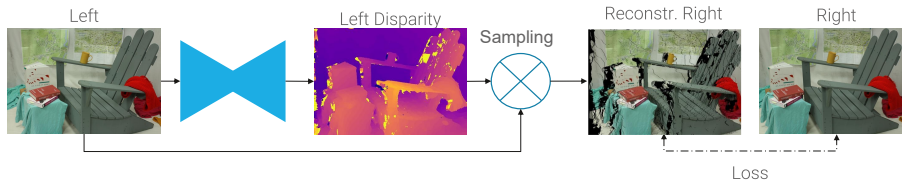


Figure 6.1: Naive training flow

**Remark.** Forward mapping creates holes and is ambiguous as disparities are non-integer values.

**(Backward) bilinear sampling** Compute the right image by determining each pixel of the output backwards and by interpolating it in the left image.

**Remark.** By reconstructing the right image backwards, the estimated disparity will be with respect to the right image, which is not available during inference.

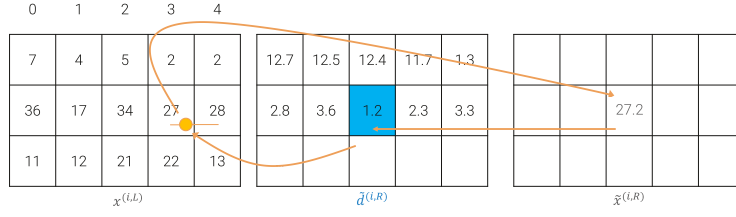


Figure 6.2: Backward reconstruction from the right image

**Training (correct)** Once the left disparity has been predicted, it is used to reconstruct the left image by backward mapping from the right image (which is available at train time).

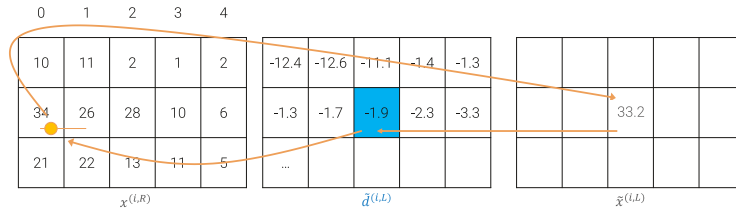


Figure 6.3: Backward reconstruction from the left image

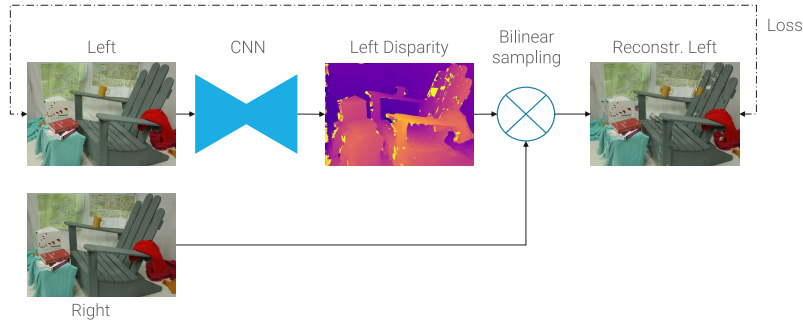


Figure 6.4: Actual training flow