Machine Learning for Computer Vision

Matteo Donati

September 30, 2022

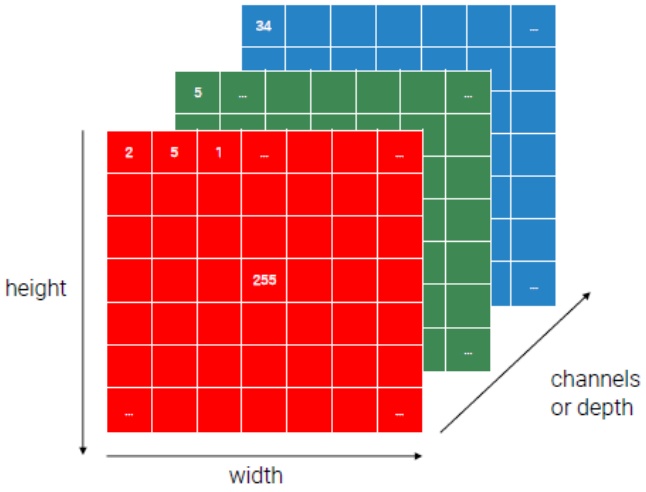
Contents

1 Introduction

Computer vision (CV) is the science of making computers gain a high-level understanding of images, and research in this field is highly influential. During the 2000s, machine learning entered the field of CV mainly with the usage of SVMs and random forests. In the 2010s, thanks to the deep learning revolution, also neural networks entered such field. In general, machine learning techniques that are deployed in the field of computer vision can be applied and re-used to solve other tasks, such as processing audio data and detecting malware.

2 Machine Learning Basics and k Nearest Neighbour

Image classification a CV task that takes in input an image, displaying a specific object, and returns one category among a set of categories. Some challenges of image classification are: intraclass variations, background clutter, occlusions, viewpoint variations, illumination challenges, and general weirdness of the world. In general, images are tensors in a computer:



and categories are numbers. In general:

- Traditional CV techniques (e.g. handcrafted rules based on edges) need a controlled environment, usually available or feasible in industrial vision applications. When these assumptions are not met, one can rely on machine learning techniques. In general, one can think of machine learning as a new way to instruct computers about what one wants them to do, where data and datasets are crucial in order to train the proposed models.
- When applying machine learning methods, one is given the following:
  - A training set $D^{train} = \{(x^{(i)}, y^{(i)}) \mid i = 1, \dots, N\}$.
  - A test set $D^{test} = \{(x^{(i)}, y^{(i)}) \mid i = 1, \dots, M\}$.

  where $x^{(i)} \in \mathbb{R}^f$ are the features representing the real world items one cares about, and $y^{(i)}$ are the outputs one wants to predict for that item. The two sets contains i.i.d. samples from the same unknown distribution $p_{data}(x, y)$.

2.1 k Nearest Neighbour

k Nearest Neighbour (kNN) is a technique that allows to solve a classification problem. In particular, such method works as follows:

1. Training.

- All the training samples and labels are stored.

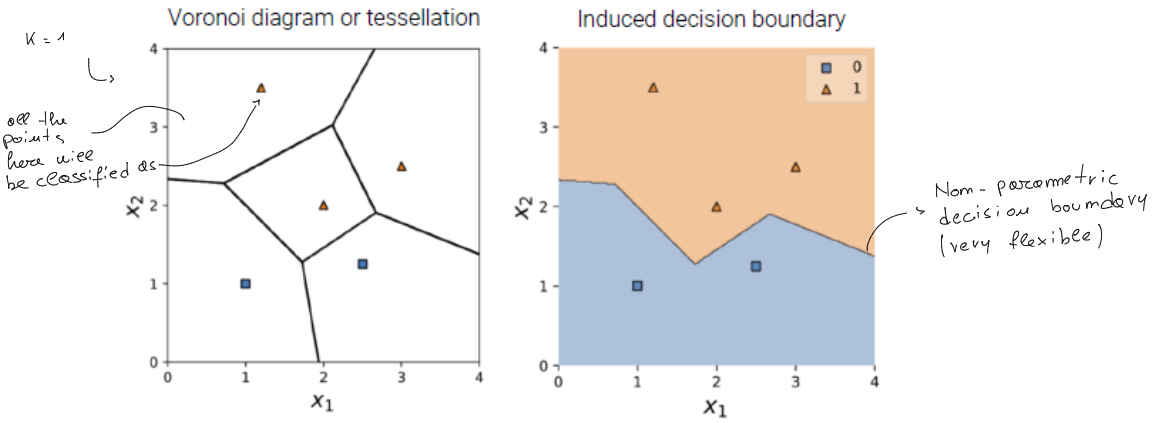
This step does not include learning, only memorization. Moreover, kNN is a lazy algorithm: all the computations are postponed to testing time.

2. Testing.

   - The distance in feature space between a test sample and all the training samples is computed.
   - The labels for the closest k training samples are retrieved.
   - Such labels are aggregated (e.g. majority vote) to define the predicted label for the specific test sample.

   kNN is only as good as the features and the distance in feature space are. Moreover, kNN is a non-parametric instance-based algorithm, which can model complex data distributions.

kNN is able to learn an induced decision boundary:



Moreover, the parameter k and the distance function are examples of hyper-parameters of a machine learning model:

- By changing k one can change the model capacity of the classifier. Informally, one can think of the model capacity as the measurements of how complex/flexible the functions that can be learned are. In particular, $k = 1$ gives the model with the largest capacity, which can perfectly classify any (training) dataset. By growing k, one reduces such capacity: decision boundaries between classes become smoother and one can misclassify more and more points of the training set (but, of course, one is only interested in classifying test points).
- Usually, distance used with kNN classifiers is an l_p norm of the difference between features:

channels first

$$d(x^{(i)}, x^{(j)}) \doteq \|x^{(i)} - x^{(j)}\|_p = \left(\sum_k \left| x_k^{(i)} - x_k^{(j)} \right|^p \right)^{1/p} \tag{1}$$

and, in particular, either the l_1 norm or the l_2 norm. To apply such norms to RGB images ($3 \times M \times N$ images), one can first flatten the image so to obtain $(3 * M * N) \times 1$ images.

- Model selection 2.2 helps one in choosing the right hyper-parameters.

Lastly, using raw pixel values as features often focuses on context and it is not semantically meaningful (e.g. a deer on a blue background can be classified as a plane). Moreover, kNN classifiers are, in principle, universal approximators, but to provide reliable predictions they would need a huge dataset (curse of dimensionality).

if one uses the test set to set hyper-parameters, then there is data-leakage

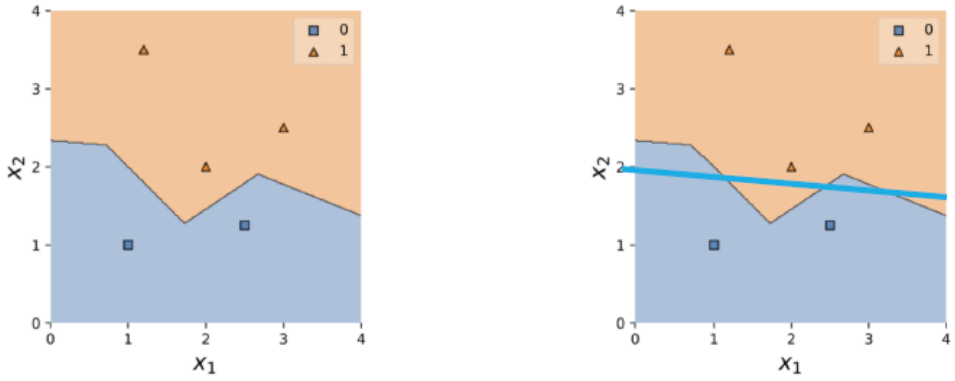
2.2 Model Selection

- Grid-search on the entire training set and test set is usually what is used to test the performances of different machine learning models. However, this approach will largely overestimate how the algorithm will perform on unseen data.
- A smarter approach would be to held out a small part of the training set, called *validation set*, which can be used to find (always through grid-search) the best combination of hyper-parameters that then can be used, only once, to evaluate the model on the test set. In general, the validation error will underestimate the generalization (or test) error.

To achieve the best generalization, one wants the model to work in the sweep spot where the training error is small, and the gap between the training and validation error is also small. When the training error is large, the algorithm is **underfitting** the training data, while when the gap between the training error and the validation error is large, the algorithm is **overfitting** the training data.

3 Linear Classifier

kNN computes a different decision boundary for every region of the space. This is very powerful but it needs a lot of samples. One could try to use just one global parametric model:



Such parametric approach could be formalized as:

$$f(\underbrace{x}_{32\times32\times3=3072\times1};\underbrace{W}_{10\times3072}) = Wx = \underbrace{y}_{10\times1} \tag{2}$$

Usually, in machine learning "linear" means "affine":

$$f(\underbrace{x}_{32\times32\times3=3072\times1};\theta=(\underbrace{W}_{10\times3072},\underbrace{b}_{10\times1})) = Wx+b = \underbrace{y}_{10\times1} \tag{3}$$

To detect which is the set possible θ one can see learning as an optimization procedure. The space of the functions that a machine learning model can produce is its **hypothesis space** $\mathbb{H}$. Learning means solving an optimization problem that finds the best function $h \in \mathbb{H}$:

$$h^* = \underset{h\in\mathbb{H}}{\operatorname{argmin}} L(h, D^{train}) \tag{4}$$

or equivalently:

$$\theta^* = \underset{\theta\in\Theta}{\operatorname{argmin}} L(\theta, D^{train}) \tag{5}$$

where L is the loss function. Indeed, instead of optimizing accuracy, one usually optimizes a proxy measure: the loss function, which is easier to optmize but still correlated with how good the classifier is. In particular, the loss value is defined as the average loss over the entire training set:

$$L(\theta, D^{train}) = \frac{1}{N}\sum_i L(\theta, (x^{(i)}, y^{(i)})) \tag{6}$$

There are different loss functions among which one can choose:

- **RMSE**, which can be implemented as follows:

$$L(\theta, (x^{(i)}, y^{(i)})) = L(Wx^{(i)} + b, y^{(i)}) = \|\overbrace{Wx^{(i)} + b}^{\text{scores}} - \overbrace{\mathbb{1}(y^{(i)})}^{\text{true labels}}\|_2 \tag{7}$$

  where $\mathbb{1}(y^{(i)})$ is the one-hot encoding of the distribution $y^{(i)}$.

- **Cross-entropy** which first allows to transform the scores (s_j) computed by the classifier into probabilities and then performs the maximum likelihood estimation of θ:

  1. The softmax activation function is computed as follows:

$$p_{model}(Y = j|X = x^{(i)}; \theta) = \underbrace{\text{softmax}_j(s)}_{\text{j-th entry of the output of the softmax}} = \frac{\exp(s_j)}{\sum_{k=1}^{C} \exp(s_k)} \tag{8}$$

  2. The maximum likelihood estimation of θ is computed as follows:

$$\begin{aligned} \theta^* &= \underbrace{\underset{\theta}{\text{argmax}}}_{\text{maximizing the probability of producing the right set of labels}} p_{model}(y^{(i)}, \dots, y^{(N)}|x^{(i)}, \dots, x^{(i)}; \theta) \\ &= \underset{\theta}{\text{argmax}} \prod_{i=1}^{N} p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta) \\ &= \underset{\theta}{\text{argmax}} \sum_{i=1}^{N} \log p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta) \\ &= \underbrace{\underset{\theta}{\text{argmin}}}_{\text{per-sample cross-entropy}} \sum_{i=1}^{N} -\log p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta) \end{aligned} \tag{9}$$

If p and q are discrete probability distributions, then this becomes:

$$H(p, q) = -\sum_{x \in \mathbb{X}} p(x) \log q(x) \tag{10}$$

Indeed, in classification, the set of events $\mathbb{X}$ is the set of classes $1, \dots, C$. One can think of the one-hot encoding of the true label as the distribution p, and the output of the model after the softmax, as the distribution q:

$$\begin{aligned} H(\mathbb{1}(y^{(i)}), p_{model}(Y|x^{(i)}; \theta)) &= -\sum_{k=1}^{C} \mathbb{1}(y^{(i)})_k \log p_{model}(Y = k|x^{(i)}; \theta) \\ &= -\log p_{model}(Y = y^{(i)}|x^{(i)}; \theta) \end{aligned} \tag{11}$$

To find a good value for θ, one can exploit the backpropagation algorithm, which is based on gradient descent. In particular, for each epoch $e = 1, \dots, E$:

The search space is $\mathbb{R}^{|\theta|}$

0. $\theta^{(0)}$ is initialized randomly.

1. Forward pass: classify all the training data to get the predictions $\hat{y}^{(i)} = f(x^{(i)}; \theta^{(e-1)})$ and the loss $L(\theta^{(e-1)}, D^{train})$.

2. Backward pass: compute the gradient $g = \frac{\partial L}{\partial \theta}(\theta^{(e-1)}, D^{train})$.

3. Step: update the parameters $\theta^{(e)} = \theta^{(e-1)} - lr * g$. The learning rate lr is a key hyper-parameter needed to reach an optimum point of the loss function.

4 Improving Gradient Descent

4.1 Stochastic Gradient Descent

Gradient descent (GD) computes the total gradient $\nabla_\theta L$ as the sum of the gradients of the single training images:

$$L(\theta, D^{train}) = \sum_i L(\theta, (x^{(i)}, y^{(i)})) \;\Rightarrow\; \nabla_\theta L(\theta, D^{train}) = \sum_{i \sim \text{all training samples}} \nabla_\theta L(\theta, (x^{(i)}, y^{(i)})) \tag{12}$$

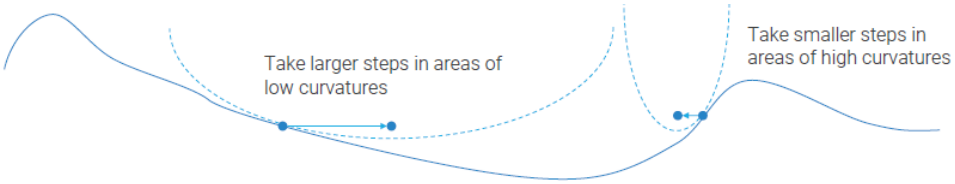
This is not feasible in practice. To avoid such problem, **stochastic gradient descent** (**SGD**) is used instead:

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{i \sim \text{mini-batch of samples}} L(\theta, (x^{(i)}, y^{(i)})) \tag{13}$$

To speed-up computation, SGD updates the network parameters after having approximated the true gradient with the gradient computed considering a mini-batch of training samples. When such mini-batch has size equal to one, then SGD is also called on-line gradient descent.

4.2 Second Order Methods

Newton's method, in the 1D case, tries to find the minimum of a function $f : \mathbb{R} \to \mathbb{R}$ by considering the Taylor expansion at point x_t: $f(x_t + \Delta x) \approx f(x_t) + f'(x_t)\Delta x + \frac{1}{2} f''(x_t)\Delta x^2$. This method fits a paraboloid at x_t having the same slope and curvature of f at x_t an then computes the update as the one yielding the minimum of the paraboloid. It turns out the update is $-\frac{f'(x_t)}{f''(x_t)}$ (i.e. a rescale of the first order derivative with the inverse of the second one).



If their theoretical conditions are met, second order methods are very effective. Indeed, they allow to converge to an optimum point without using the learning rate. However, second-order methods are usually impractical: one-step convergence only holds for perfectly quadratic functions. In general, the update rule in the multivariate and non-quadratic case becomes $-lrH_f^{-1}(x_t)\nabla f(x_t)$, where H_f is the matrix of second-order derivatives (Hessian matrix) that describes the local curvature of a multivariate function:

$$H_f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_k}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_k \partial x_1}(x) & \frac{\partial^2 f}{\partial x_k \partial x_2}(x) & \dots & \frac{\partial^2 f}{\partial x_k^2}(x) \end{bmatrix} \tag{14}$$

4.3 Momentum

Momentum is one of the most used modifications to SGD. One can think of momentum as adding a "velocity" term to the update rule of SGD:

averaging gradient (smoothing, removing noise)

$$v^{(t+1)} = \beta v^{(t)} - lr\nabla L(\theta^{(t)}) \tag{15}$$

$$\theta^{(t+1)} = \theta^{(t)} + v^{(t+1)} \tag{16}$$

To get an idea of how this works, one can think of a ball, rolling down a surface, that acquires speed (i.e. accumulates momentum) in the downward direction, which is not immediately lost when the surface changes curvature. This helps escaping local optimum. In particular, $\beta \in [0, 1)$ is the momentum coefficient. For $\beta = 0$, $v^{(t+1)}$ is equal to the update step of SGD. Otherwise, $v^{(t+1)}$ contains a running average of the previous update steps, $\{-lr\nabla L(\theta^{(0)}), \ldots, -lr\nabla L(\theta^{(t-1)})\}$.

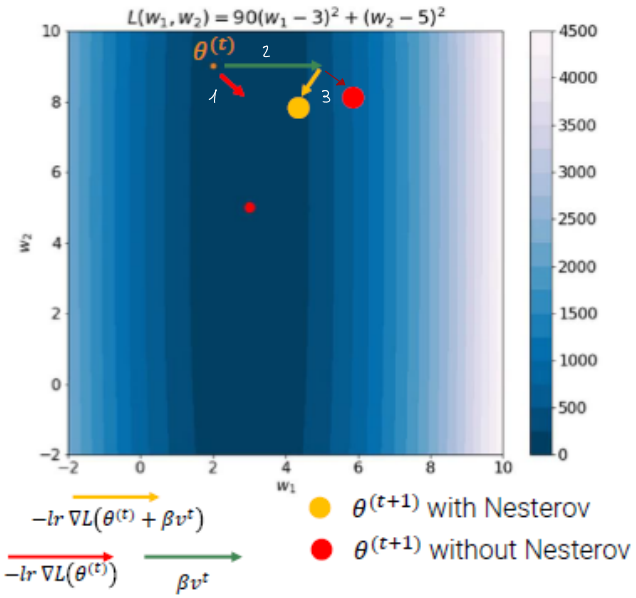
4.3.1 Nesterov Momentum

An important variant of SGD with momentum is **Nesterov Accelerated Gradients** (NAG):

$$v^{(t+1)} = \beta v^{(t)} - lr\nabla L(\theta^{(t)} + \beta v^{(t)}) \tag{17}$$

$$\theta^{(t+1)} = \theta^{(t)} + v^{(t+1)} \tag{18}$$

This is very similar to momentum, but the gradient is computed after having partially updated $\theta^{(t)}$ with $\beta v^{(t)}$. In other words, this update looks ahead to compute a more accurate gradient.



keep in "comyoms", when parameters have different weights

4.4 Adaptive Learning Rates

A different paradigm than momentum to improve over SGD is to define adaptive learning rates. In particular, there exist different formulations of such technique:

- **Adaptive Gradient** (**AdaGrad**) proposed to rescale each entry of the gradient with the inverse of the history of its squared values:

squaring the gradient (element-wise product)

history of the gradient for each dimension

$$s^{(t+1)} = s^{(t)} + \nabla L(\theta^{(t)}) \odot \nabla L(\theta^{(t)}) \tag{19}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{lr}{\sqrt{s^{(t+1)}} + \epsilon} \odot \nabla L(\theta^{(t)}) \tag{20}$$

the learning rate depends on the dimension

  In general, weights receiving small gradients have their effective learning rate increased, while weights receiving large gradients have their effective learning rate decreased. However, $s^{(t)}$ is monotonically increasing and it may reduce all learning rates too early, when one is far from a good minimum.

- **RMSProp** modifies AdaGrad to down-weigh history of the past and keep the optimizer responsive. In particular, it introduces a parameter, $\beta \in [0, 1]$, to govern the decay of $s^{(t)}$ through an exponential moving average:

$$s^{(t+1)} = \beta s^{(t)} + (1 - \beta)\nabla L(\theta^{(t)}) \odot \nabla L(\theta^{(t)}) \tag{21}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{lr}{\sqrt{s^{(t+1)}} + \epsilon} \odot \nabla L(\theta^{(t)}) \tag{22}$$

  Typically, $\beta = 0.9$ or higher.

- **Adaptive Moments** (**ADAM**) extends this idea to keep also a running average of gradients (i.e. a momentum-like term):

running average of gradients, a momentum-like term

$$g^{(t+1)} = \beta_1 g^{(t)} + (1 - \beta_1)\nabla L(\theta^{(t)}) \tag{23}$$

as in RMSProp

$$s^{(t+1)} = \beta_2 s^{(t)} + (1 - \beta_2)\nabla L(\theta^{(t)}) \odot \nabla L(\theta^{(t)}) \tag{24}$$

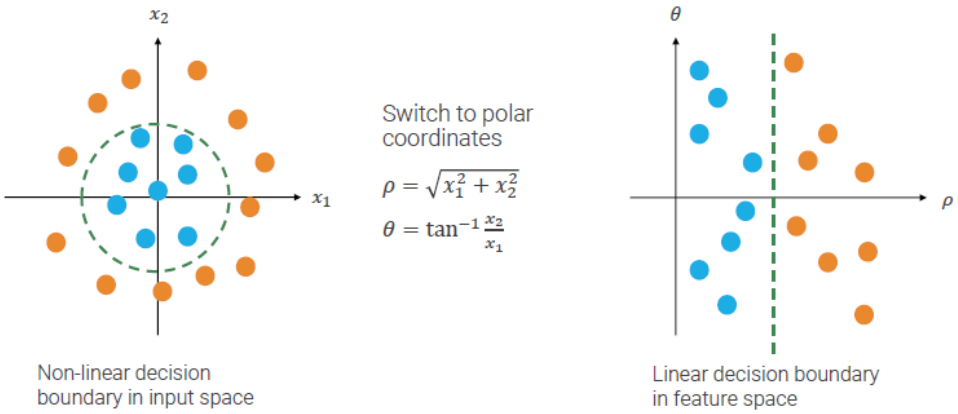
$$g^{debiased} = \frac{g^{(t+1)}}{1 - \beta_1^{(t+1)}}, \quad s^{debiased} = \frac{s^{(t+1)}}{1 - \beta_2^{(t+1)}} \tag{25}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{lr}{\sqrt{s^{debiased}} + \epsilon} \odot g^{debiased} \tag{26}$$

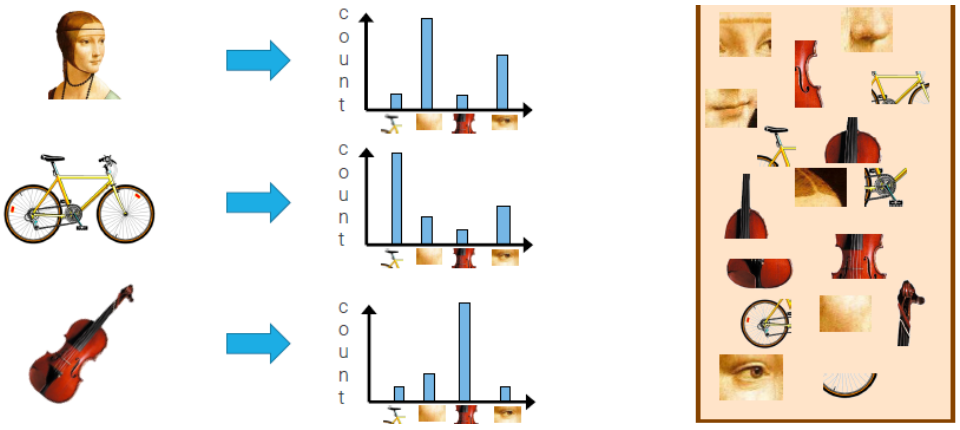
  Typically, $\beta_1 = 0.9$, $\beta_2 = 0.999$. Since $g^{(0)} = s^{(0)} = 0$, and β_1, β_2 are large, the first values of g and s will be very small, regardless of the values of the gradient. This is why bias-corrected terms are used.

5 Image Representations

kNN and linear classifiers are limited by the low effectiveness of input pixels as data features. One could, for example, represent data in polar coordinates:



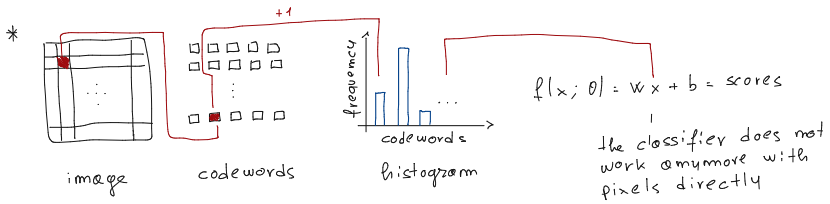
Or one could use another approach, called **Bag of visual words** (**BoVW**), which is used to define histograms as follows:



Given an image, one would like to represent it as an histogram, counting the frequency of appearance of every (code)word from a dictionary. However, images are not made of discrete elements. Words detection can be done either:

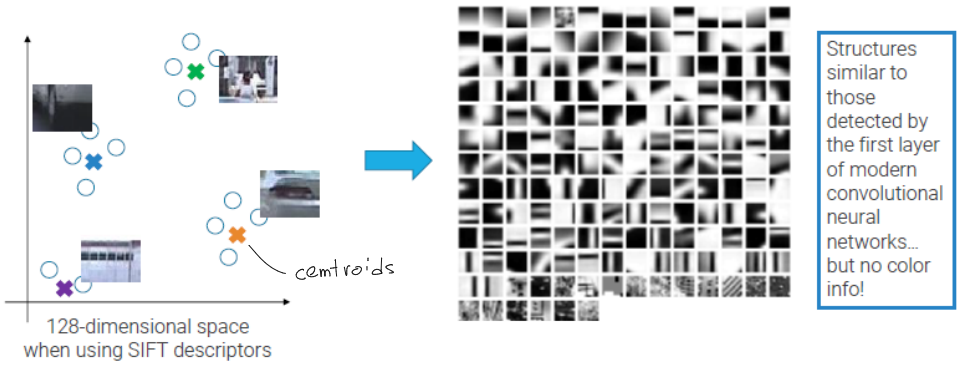
- By using a regular grid of positions and randomly sampling the scale. — image is split in patches, and each patch is a word
- Running an interest points detector (Harris, DoG, and similar).

Then, either:



- Use the patch around the keypoint as word.
- Compute its SIFT descriptor as word.
- Compute some color descriptor as word.

The dictionary, or codebook, is formed by clustering the extracted patches/descriptors in a finite number of centroids. Usually, k-means clustering is used:



Given an image, the same extraction pipeline used to create the codebook is applied to such image. Each extracted descriptor/patch is matched against the codebook and assigned to the nearest centroid in the codebook. In vanilla BoVW, the histogram entry corresponding to such codeword is incremented. There exist also variations to the standard BoW approach, which try to make more informative the assignment to a codeword. For example, if descriptors are D dimensional, and there are K codewords, more sophisticated methods (VLAD) create a $D \times K$ descriptor which stores the difference of input words with respect to the codewords.

5.1 Neural Networks

otherwise the output will still be linearly dependent from the input

Neural networks add a depth dimension to the standard linear classifier. Indeed, these networks are composed of multiple layers, and each neuron belonging to such layers applies a non-linear activation function to the weighted sum of its input and the corresponding weights. In particular, there exist many activation functions, such as:

- The **sigmoid** function: $\phi(a) = \frac{1}{1+\exp(-a)}$.
- The **rectified linear unit** function: $\phi(a) = \max(0, a)$.

Moreover, every layer is often called a **fully-connected layer**, as every neuron of a layer influences every neuron of the subsequent layer. A neural network with two or more layers is also called a **multi-layer perceptron**. Like kNN classifiers, neural networks enjoy a form of universal approximation property. Indeed, networks with a single hidden layer can be used to approximate any continuous function to any desired precision.

6 Convolutional Neural Networks

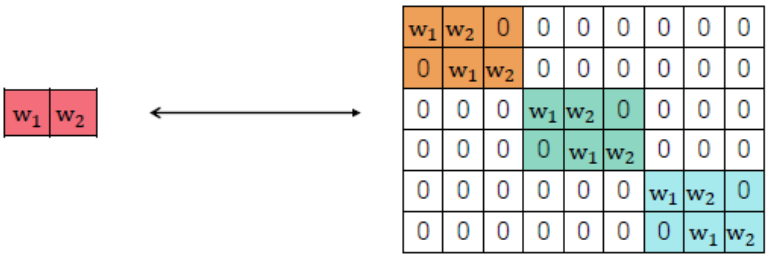
In general, fully connected layers require many parameters to compute simple and local features, while convolutional layers use kernels that are able to exploit local input features through convolution/correlation. Even though these layers are called "convolutional", they apply cross-correlation:

$$[K \star I](i,j) = \sum_l \sum_m K(l,m) I(i+l, j+m) \tag{27}$$

where the kernel K is not flipped with respect to the origin of the 2-dimensional space. Moreover, convolution can be interpreted as matrix multiplication, if one reshapes inputs and outputs. The resulting matrix is still a linear operator, which:

- Shares parameters across its rows.
- Is sparse.
- Naturally adapts to varying input sizes.
- Is ~~equivalent~~ to translations of the input.

For example, the following kernel can be transformed as follows:

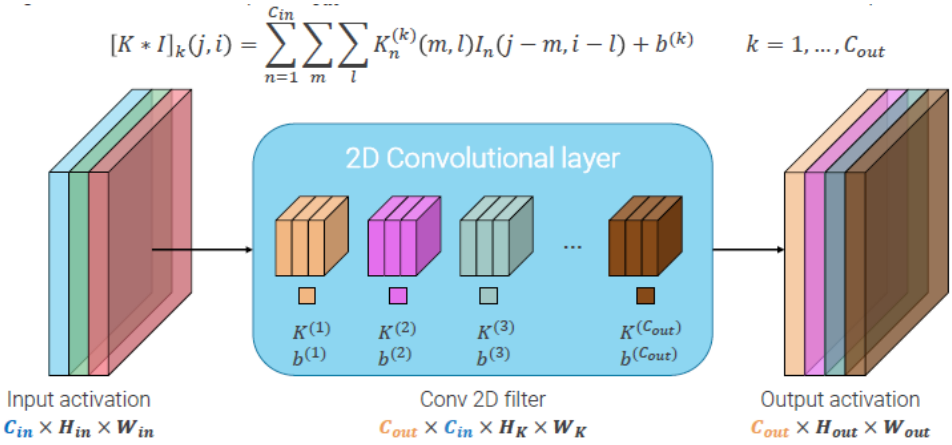


Images have three channels, so convolution kernels will be a 3-dimensional tensors of size $3 \times H_K \times W_k$:

$$[K * I](j,i) = \sum_{n=1}^{3} \sum_m \sum_l K_n(m,l) I_n(j-m, i-l) + b \tag{28}$$

When the filter is at position (j,i) on the input image, one computes one number. By sliding the filter over the image, one gets a single-valued output image, which is the output activation. Such image is also called **feature map**:

One can repeat this procedure with other filters, with different weights, and stack the new feature maps with the previous ones:



Convolutional layers can be interpreted as a constrained form of linear layers. Hence, to meaningfully compose them, one needs to insert a non-linear activation function between them.

6.1 Relationship Between Spatial Dimensions

In general:

$$H_{out} = H_{in} - H_K + 1 \tag{29}$$
$$W_{out} = W_{in} - W_K + 1 \tag{30}$$

If one stacks several convolutional layers, the feature maps will shrink after each layer. To avoid this, one can rely on zero-padding:

$$H_{out} = H_{in} - H_K + 1 + 2P \tag{31}$$
$$W_{out} = W_{in} - W_K + 1 + 2P \tag{32}$$

where, usually, $P = (H_K - 1)/2$ so to have an output with the same size of the input. Moreover, the **receptive field** of a hidden unit is the set of input pixels affecting such hidden unit. For example, if one applies a $H_K \times W_K$ kernel at each layer, the receptive field of an element in the L-th feature map will have size $[1 + L(H_K - 1)] \times [1 + L(W_K - 1)]$, where L is the number of layers. To compute attributes corresponding to a large portion of the input, one would need too many layers. Thus, to obtain larger receptive fields with a limited number of layers, one can down-sample the activations inside the network. This can be achieved using **strided convolutions**:

$$H_{out} = \left\lfloor \frac{(H_{in} - H_K + 2P)}{S} + 1 \right\rfloor \tag{33}$$
$$W_{out} = \left\lfloor \frac{(W_{in} - W_K + 2P)}{S} + 1 \right\rfloor \tag{34}$$

the receptive field at layer L is:

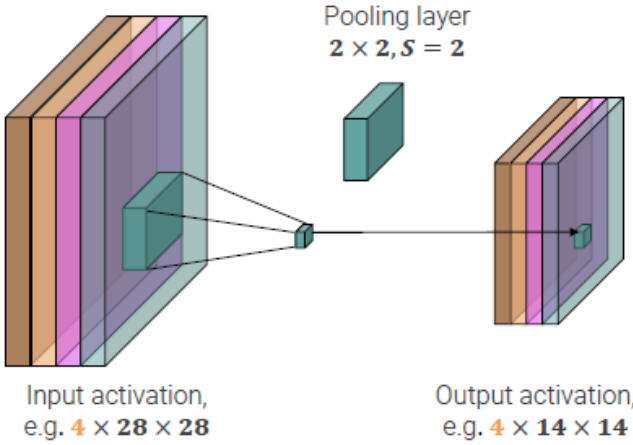
$$r_L = \left[\sum_{l=1}^{L} \left((H_K - 1) \prod_{i=1}^{l-1} s_i \right) + 1 \right] \times \left[\sum_{l=1}^{L} \left((W_K - 1) \prod_{i=1}^{l-1} s_i \right) + 1 \right]$$

6.2 Common Layers in CNNs

Convolutional neural networks are usually composed of convolutional layers, non-linear activations, fully-connected layers, pooling layers, and batch-normalization layers. The first portion of a CNN is usually composed of convolutional layers, non-linear activations and pooling layers. This first portion of the network is able to extract features from the input. The latter part of the network is usually fully-connected.

- **Pooling layers** aggregate several values into one output value with a pre-specified (i.e. not learned) kernel. The key difference with convolution is that each input channel is aggregated independently (i.e. the kernel works only along the spatical dimensions, $C_{out} = C_{in}$).

Provide invariance. E.g. in max-pool, one does not care where the max was found (i.e. invariance with respect to small spatial shifts)



- **Batch normalization layers** work by normalizing the output of a layer during training, so that each dimensioon has zero mean and unit variance in a batch.

imposing a normal distribution to avoid that activations of the different layers change in a non-controlled manner when updating the network's parameters

this distribution gets changed a bit thanks to two learnable parameters, so to introduce flexibility into the learned representations

this is performed considering the activations (before non-linearity) of the specific layer, produced with respect to each input sample in each mini-batch (indeed, batch-norm works better with large batches)

mean variance

Moreover, during training, a moving average for both μ and σ is constantly updated so to take into account the effect of different mini-batches. These two averages are used during testing.

- Layer normalization: alternative to batch normalization. It has the same behaviour at train and test time. It works by normalizing along the layer dimension (i.e. activation dimension), instead of along the batch dimension

normalizing at each training sample

7 CNN Architectures

Examples of famous architectures are the following:

- **AlexNet**, composed of five convolutional layers, each followed by a ReLU activation and optionally by max-pooling layers, and of three fully-connected layers. This network was introduced in 2012 and it was training on two GPUs for five/six days. The overall architecture is the following:

Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					224	3	150528	0	-	73,5	0,0
conv1	96	11	4	2	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216	0	0,0	0,0	0,0
fc6	4096	1	1	0	1	4096	4096	37758	9663,7	4,0	432,1
fc7	4096	1	1	0	1	4096	4096	16781	4295,0	4,0	192,0
fc8	1000	1	1	0	1	1000	1000	4100	1048,6	1,0	46,9
					Minibatch:	128	Totals:	62386,56	290766,6	763,4	714,0

For convolutional layers:
#params = $(W_K \times H_K \times C_{im} + 1) \times C_{out}$
flops = #activations $\times (W_K \times H_K \times C_{im}) \times 2$

For fully-connected layers:
#params = $C_{out} \times C_{im} + C_{im}$
flops = #minibatches $\times 2 \times C_{im} \times$ #activations

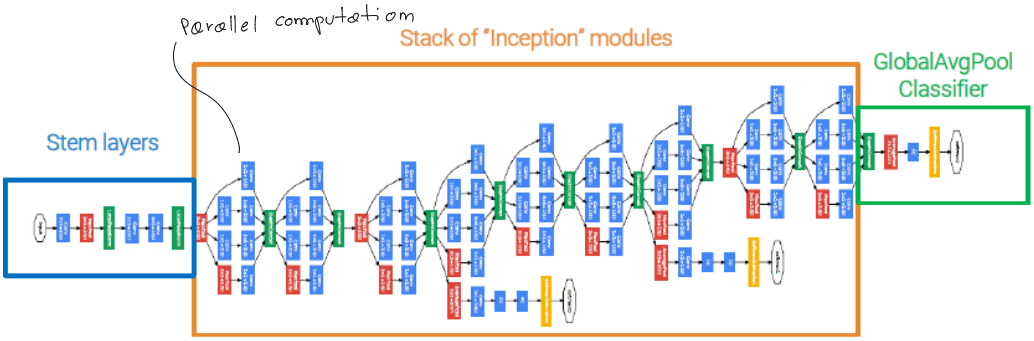
activations memory needs to also store gradients.
parameters memory needs to also store other parameters (e.g. optimizer parameters)

- **ZFNet / Clarifai**, tries to reduce the ~~trail~~ trial and error approach to network design, by introducing powerful visualizations (via deconvnets) for layers other than the first one. This architecture is based on visualizations and ablation studies.

- **VGG**, which is a commit to explore the effectiveness of simple design choices, by allowing the combination of 3×3 convolutions with $S = 1$ and $P = 1$, 2×2 max-pooling with $S = 2$, $P = 0$, where the number of channels doubles after each pool. Moreover, VGG introduced the idea of designing a network as repetitions of stages, where one stage has the same receptive field of larger convolutions but requires less parameters, and introduces more non-linearities. The overall architecture of VGG-16 is the following:

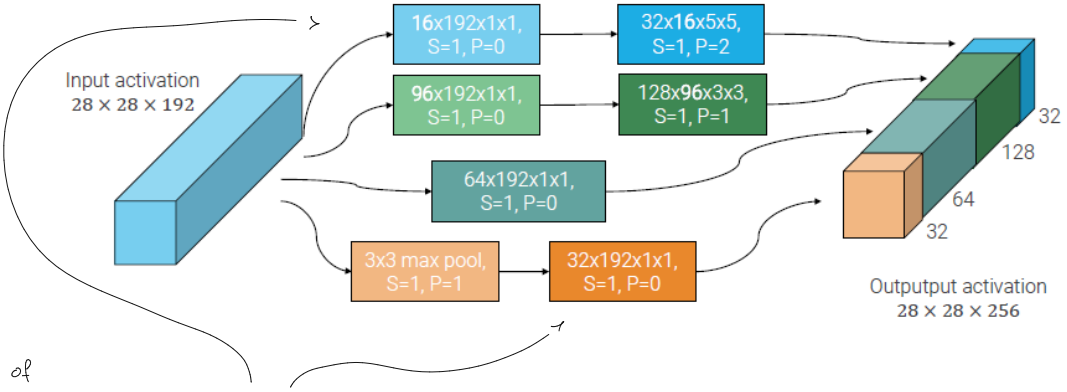
layer	Kernels	K_W/H	S	P	Actv H/W	Actv Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Params memory (MB)
input					224	3	150528	0	-	73,5	0,0
conv1	64	3	1	1	224	64	3211264	2	22196,3	3136,0	0,0
conv2	64	3	1	1	224	64	3211264	37	473520,1	3136,0	0,4
pool1	1	2	2	0	112	64	802816	0	411,0	784,0	0,0
conv3	128	3	1	1	112	128	1605632	74	236760,1	1568,0	0,8
conv4	128	3	1	1	112	128	1605632	148	473520,1	1568,0	1,7
pool2	1	2	2	0	56	128	401408	0	205,5	392,0	0,0
conv5	256	3	1	1	56	256	802816	295	236760,1	784,0	3,4
conv6	256	3	1	1	56	256	802816	590	473520,1	784,0	6,8
conv7	256	3	1	1	56	256	802816	590	473520,1	784,0	6,8
pool3	1	2	2	0	28	256	200704	0	102,8	196,0	0,0
conv8	512	3	1	1	28	512	401408	1180	236760,1	392,0	13,5
conv9	512	3	1	1	28	512	401408	2360	473520,1	392,0	27,0
conv10	512	3	1	1	28	512	401408	2360	473520,1	392,0	27,0
pool4	1	2	2	0	14	512	100352	0	51,4	98,0	0,0
conv11	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
conv12	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
conv13	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
pool5	1	2	2	0	7	512	25088	0	12,8	24,5	0,0
flatten	1	1	1	0	1	25088	25088	0	0,0	0,0	0,0
fc14	4096	1	1	0	1	4096	4096	102786	26306,7	4,0	1176,3
fc15	4096	1	1	0	1	4096	4096	16781	4295,0	4,0	192,0
fc16	1000	1	1	0	1	1000	1000	4100	1048,6	1,0	46,9
					Minibatch:	128	Totals:	138.382	3.961.171	14.733	1.584

- **Inception v1** (**GoogLeNet**), which improves utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for the increasing of the depth and width of the network while keeping the computational budged constant.



where:

  - The stem layers aggresively downsamples inputs: from 224 to 28 width/height in five layers. This convolutions only use $S = 2$, and the largest convolutional kernel is 7×7.
  - The inception modules are composed as follows:



number of FLOPs is greatly reduced

where adding 1×1 convolutions before larger convolutions and after max-pooling allows to control the number of output channels, by reducing the depth of the max-pooling output, and to control the time complexity of the larger convolutions by reducing the channel dimension.

  - Global average pooling layers are used to remove spatial dimensions. Lastly, a fully-connected layer is used to produce class scores.

The overall architecture of is the following:

stem layers

inception modules

global avg. pooling

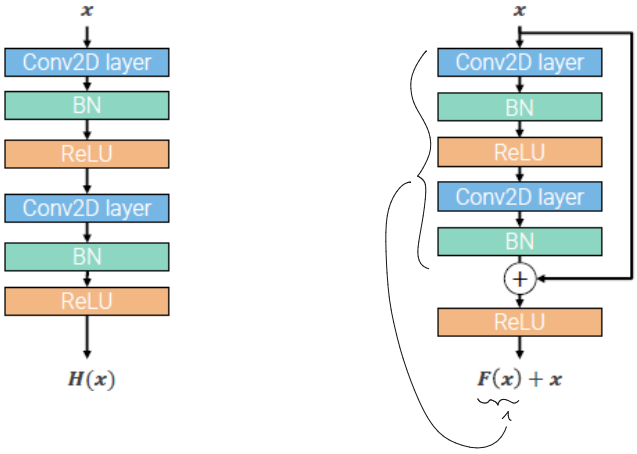
a lot of hyperparameters, but few parameters and few GB of memory

	inception 1x1				inception 3x3			Inception 5x5			Maxpool		Activations			#params		Acts memory	Params memory
Layer	Ks	H/W	S	P	C_out	1x1	H/W	C_out	1x1	H/W	1x1	H/W	H/W	Channels	#Activations	(K)	flops (M)	(MB)	(MB)
input													224	3	150528	0	-	73,5	0,0
conv1	64	7	2	3									112	64	802816	9	30211,6	784,0	0,1
pool1	1	3	2	1									56	64	200704	0	231,2	196,0	0,0
conv2	64	1	1	0									56	64	200704	4	3288,3	196,0	0,0
conv3	192	3	1	1									56	192	602112	111	88785,0	588,0	1,3
pool2	1	3	2	1									28	192	150528	0	173,4	147,0	0,0
incep1	64	1	1	0	128	96	3	32	16	5	32	3	28	256	200704	163	31380,5	196,0	1,9
incep2	128	1	1	0	192	128	3	96	32	5	64	3	28	480	376320	388	75683,1	367,5	4,4
pool3	1	3	2	1									14	480	94080	0	108,4	91,9	0,0
incep3	192	1	1	0	208	96	3	48	16	5	64	3	14	512	100352	376	17403,4	98,0	4,3
incep4	160	1	1	0	224	112	3	64	24	5	64	3	14	512	100352	449	20577,8	98,0	5,1
incep5	128	1	1	0	256	128	3	64	24	5	64	3	14	512	100352	509	23609,2	98,0	5,8
incep5	112	1	1	0	288	144	3	64	32	5	64	3	14	528	103488	605	28233,4	101,1	6,9
incep6	256	1	1	0	320	160	3	128	32	5	128	3	14	832	163072	867	41445,4	159,3	9,9
pool4	1	3	2	1									7	832	40768	0	47,0	39,8	0,0
incep7	256	1	1	0	320	160	3	128	32	5	128	3	7	832	40768	1042	11860,0	39,8	11,9
incep8	384	1	1	0	384	192	3	128	48	5	128	3	7	1024	50176	1443	16689,7	49,0	16,5
avgpool	1	1	1	0									1	1024	1024	0	6,4	1,0	0,0
fc1	1000	1	1	0									1	1000	1000	1025	262,1	1,0	11,7
												Minibatch:		128	Totals	6.992	389.996	3.251	80

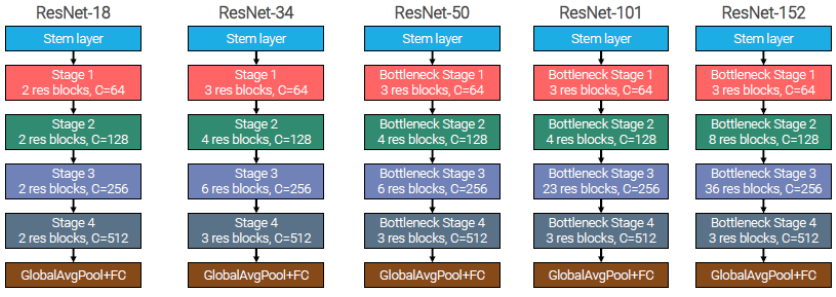
- **Residual networks**, which allow networks to learn identity functions. Indeed, in general, optimizing very deep networks is hard, and even though a network with, for example, twenty layers achieves certain performances, then it is not sure that by only adding identity layers one is able to maintain the same exact performances. Residual network try to solve this problem by introducing residual blocks that are implemented by adding skip connections:

bigger parameters space that SGD cannot navigate well

to solve such problem, identity is injected into the network



The weights of such blocks are usually initialized to zero, so to start the training having an output which is identical to the input (i.e. by starting with the implementation of the identity function), and then by learning an optimal perturbation of the initial function. Inspired by VGG regular design, ResNets are stacks of stages with fixed design rules:



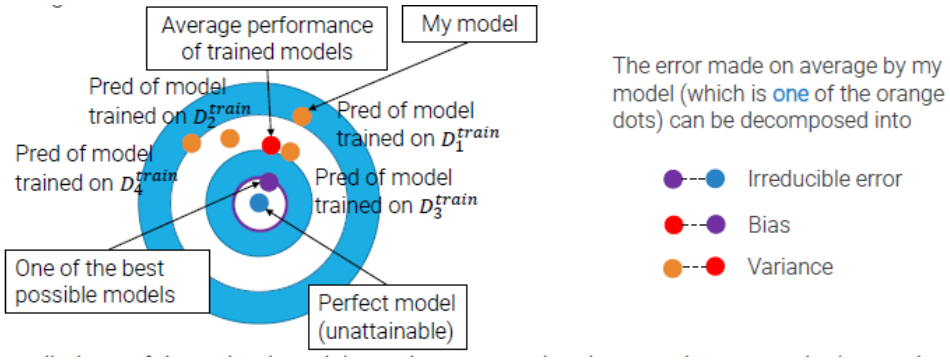
* The residual blocks described so far cannot be used as the first block of a new stage due to different tensor shapes. This can be solved by using 1x1 convolutions with stride 2 and 2C output channels

a variant of these layers are bottleneck residual blocks

8 Regularization

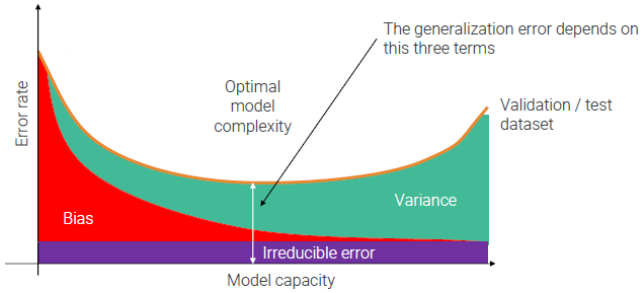
8.1 Bias and Variance

Supposing to repeat the process of gathering a training set and fitting a model, where each training dataset is a sample from a population, then, every time one collects a training dataset D_i^{train} and trains the same model, one would get different results:

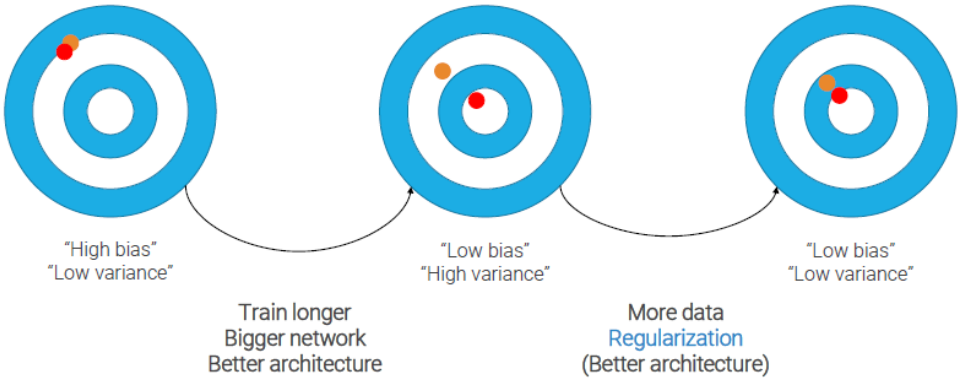


The closer one is to the center, the better will be the predictions of the trained model.

In general, for complex models the bias-variance trade-off is used mainly as a qualitative guide to improve generalization.



In practice one does not have n training datasets, but only one validation and one training dataset, so one can approximate bias with training loss/accuracy, and variance with the difference between training and validation loss/accuracy. In general, one cannot try all possible combinations of network width, depth, etc., so one can leverage high capacity models to reach a low bias, and then one can try reducing the variance.



8.2 Regularization

Regularization is any modification one makes to a learning algorithm that is intended to reduce the generalization (test) error. Examples of regularization are: parameter norm penalties, early stopping, label smoothing, dropout, data augmentation:

- **Parameter norm penalties** can be used to implicitly add a term to the loss:

$$L(\theta; D^{train}) = \underbrace{L^{task}(\theta; D^{train})}_{\text{task loss}} + \underbrace{\lambda L^{regularization}(\theta)}_{\text{regularization loss}} \tag{35}$$

  Two commonly used norms are:

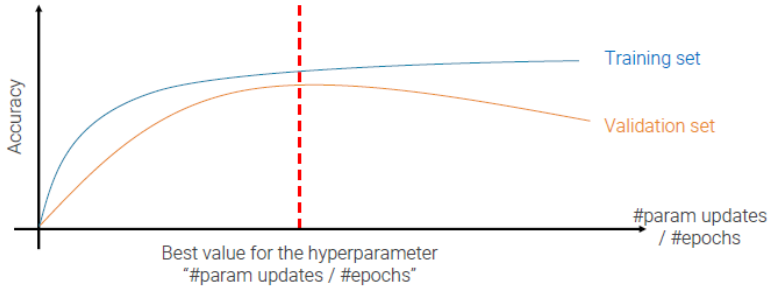
  - l_2 regularization: $L^{regularization}(\theta) = \sum_i \theta_i^2$.
  - l_1 regularization: $L^{regularization}(\theta) = \sum_i |\theta_i|$.

  l_2 regularization is also known as **weight decay** because every gradient descent step drives the weights toward the origin before apllying the same update used when regularization is not present:

$$\begin{aligned}
\theta^{(i+1)} &= \theta^{(i)} - lr \nabla_\theta L(\theta^{(i)}; D^{train}) \\
&= \theta^{(i)} - lr \nabla_\theta \left[L^{task}(\theta^{(i)}; D^{train}) + \frac{\lambda}{2} \|\theta^{(i)}\|_2^2 \right] \\
&= \underset{\emptyset}{\sout{theta}}^{(i)} - lr[\nabla_\theta L^{task}(\theta^{(i)}; D^{train}) + \lambda \theta^{(i)}]
\end{aligned}$$

$$= \underbrace{\theta^{(i)}(1 - lr\lambda)}_{\text{decayed parameter vector}} \quad -lr \quad \underbrace{\nabla_{theta} L^{task}(\theta; D^{train})}_{\text{gradient without regularization}}$$

- **Early stopping** can be used to stop training when the accuracy/loss on the validation set is optimal.

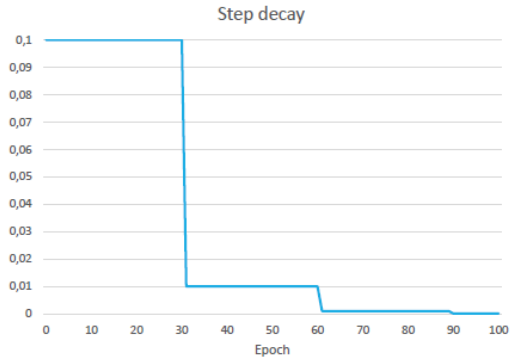


- **Dropout** can be used to randomly set some activations to zero. The probability of dropping is a hyper-parameter $p \in \mathbb{R}_{(0,1)}$. Every time the network is trained on a new mini-batch of examples, a new binary mask of active nodes is computed. The idea behind dropout is that each hidden unit must be able to perform well regardless of which other hidden units are in the model. This regularizes units to be useful in many contexts.
- **Data augmentation** can be used to add new data so to reduce variance. Examples of data augmentation are spatial flipping, spatial rotations, sampled random crops/scales, colour augmentations.

9 Practical Training and Testing

One of the biggest problems when training a neural network is how to select the proper learning rate. In particular, there exist different techniques that allows one to dynamically adjust such learning rate to find an optimal point of the considered loss function. Indeed, some examples of learning rate schedules are the following:

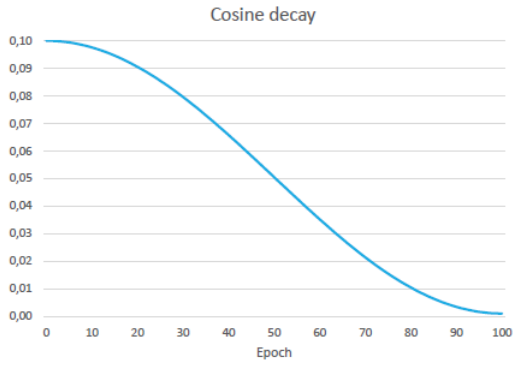
- **Step**. This schedule starts with a high learning rate (e.g. 0.1), and divides it by 10 when the error plateaus. This type of schedule is used in ResNets.



- **Cosine**. This schedule computes the learning rate for epoch e (where the total number of epochs is equal to E) as follows:

$$lr_e = lr_E + \frac{1}{2}(lr_0 - lr_E)\left(1 + \cos\left(\frac{e\pi}{E}\right)\right) \tag{36}$$

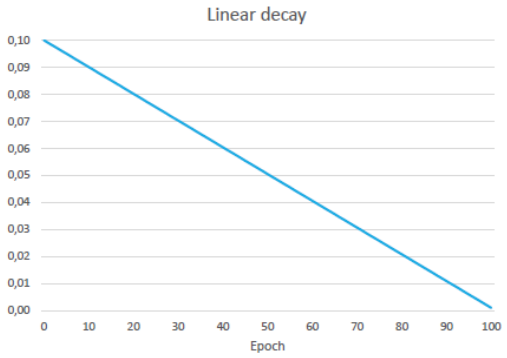
  It lets the optimizer follow a similar path to step decay, but with less parameters to tune. This schedule is mostly used in computer vision.



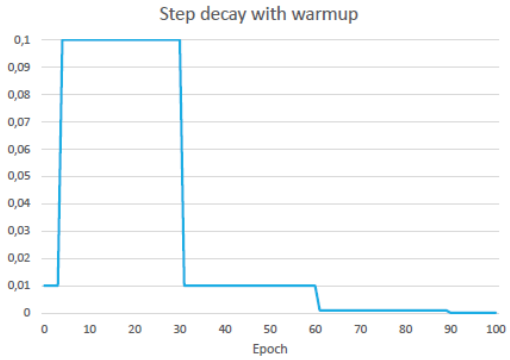
- **Linear**. This schedule computes the learning rate for epoch e as follows:

$$lr_e = lr_E + (lr_0 - lr_E)\left(1 - \frac{e}{E}\right) \tag{37}$$

This represents an even simpler alternative to coarsely emulate step decay. This schedule is mostly used in natural language processing.



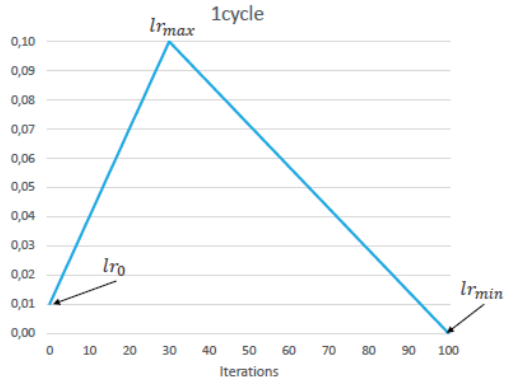
- **Warm-up**. This schedule starts with a lower learning rate, which is used for few epochs, so to avoid poops initialization. Indeed, high learning rates can slow down convergence at the beginning of training.



- **One cycle**. This schedule modifies the learning rate after each mini-batch. If the total number of iterations is I, the learning rate for each iteration i is computed as:

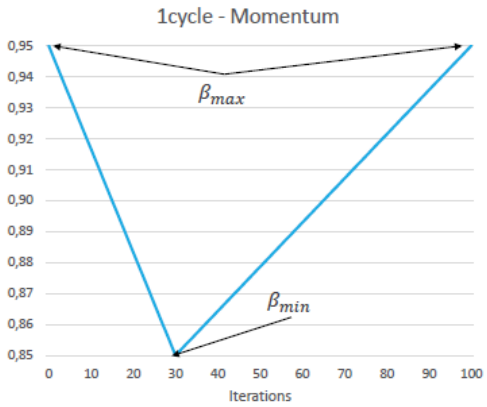
$$lr_i = \begin{cases} lr_{max} + (lr_0 - lr_{max})\left(1 - \frac{i}{pI}\right) & \text{if } i < pI \\ lr_{min} + (lr_0 - lr_{min})\left(1 - \frac{i-pI}{I-pI}\right) & \text{if } i \geq pI \end{cases} \tag{38}$$

where p is the fraction of iterations to increase the learning rate (e.g. 0.3).



This schedule also proposes to vary momentum β (β_1 in Adam) in the opposite way:

$$\beta_i = \begin{cases} \beta_{min} + (\beta_{max} - \beta_{min})\left(1 - \frac{i}{pI}\right) & \text{if } i < pI \\ \beta_{max} + (\beta_{min} - \beta_{max})\left(1 - \frac{i-pI}{I-pI}\right) & \text{if } i \geq pI \end{cases} \tag{39}$$

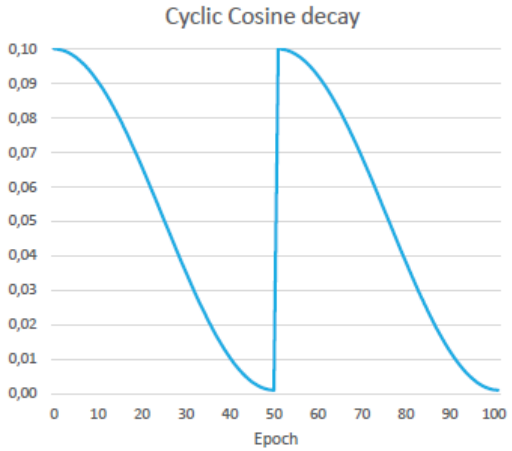


Other than learning rate, one could also apply a random hyper-parameters search (opposite to a grid search), which usually leads to a more efficient exploration of the space. Lastly, to improve the generalization capabilities for a given task, one could use **ensembling**:

1. Train multiple (randomly initialized) models on the same dataset.
2. Run each model over a test image.
3. Average the results.

This usually increases the overall performance by some percentage points: even if the networks have similar error rates, they tend to make different mistakes. However, this requires training different

models. One way to avoid to do so is to use **snapshot ensembling**: by using a cyclic learning rate schedule one can simulate M trainings in the time span of one, by taking snapshots of the parameters reached at the end of each cycle.



Moreover, at test time, one could store a vector of parameters which is updated with an exponential moving average at each step:

$$\theta^{(test)} = (1-\rho)\theta^{(i+1)} + \rho\theta^{(test)} \tag{40}$$

with $\rho \in [0, 1]$. This allows one to have a set of test parameters that is an average of the weights achieved at every snapshot (**Polyak average**). Other than Polyak average, one could compute a real average of best models:

$$\theta^{(test)} = \frac{\theta^{(cycle)} + n_{cycles}\theta^{(test)}}{n_{cycles} + 1} \tag{41}$$

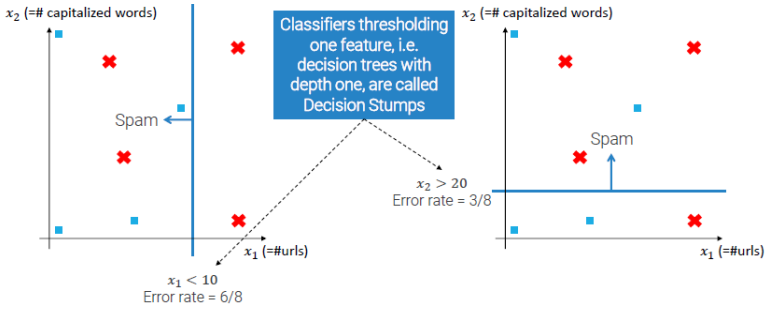
10 Object Detection

Object detection is the problem of taking in input a RGB image of size $W \times H$ and output a set of objects. In particular, for each object o_j one should process a category $c_j \in [1, \dots, C]$, a bounding box $BB_j = [x_j, y_j, w_j, h_j]$ where $x_j, w_j \in [0, W-1]$ and $y_j, h_j \in [0, H-1]$. One of the first successful applications is Viola Jones object detection, which enables real-time face detection in unconstrained environments. This detector introduced three main innovations: efficient multi-scale rectangular features computed by means of integral images, the use of AdaBoost to learn effective ensemble of features, and cascade to attain real-time speed. In particular:

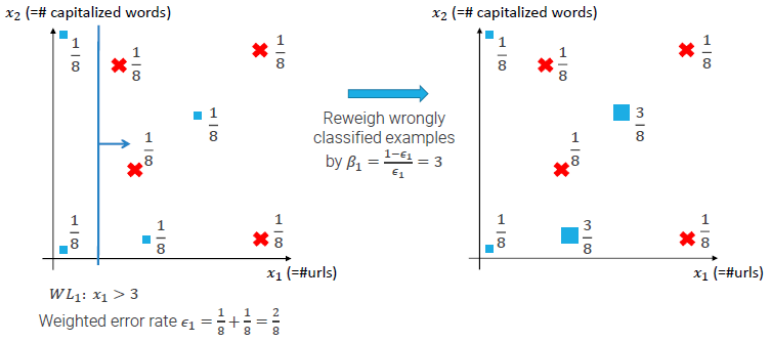
- **Boosting** is a way to train and build an ensemble of M weak learners (i.e. classifiers whose error is slightly better than random guessing) to obtain a strong learner (i.e. a classifier whose accuracy is strongly correlated with the true labels). The key idea is that weak classifiers are learned sequentially; after training a weak learner, the examples are re-weighted in order to emphasize those which were incorrectly classified by the previous weak classifier.

$$SL(x) = \sum_{j=1}^{M} \alpha_j WL_j(x) > 0 \tag{42}$$

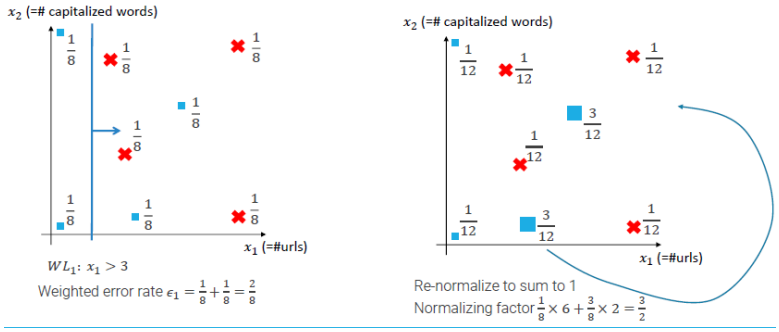
An example of weak learner is given by decision trees with depth equal to one, also called **decision stumps**:



As previously specified, during boosting, wrongly classified examples get re-weighted:



and then re-normalized:



This entire procedure is repeated for M different weak learners. Indeed, AdaBoost works as follows:

1. Given N training samples $(x^{(i)}, y^{(i)})$, initialize their weights $w^{(i)} = \frac{1}{N}$.
2. For $j = 1, \dots, M$ weak learners:
   (a) Fit the best classifier WL_j to the weighted training data.
   (b) Compute the weighted error rate $\epsilon_j = \sum_{i:x^{(i)} \text{ is misclassified}} w^{(i)}$.
   (c) Compute $\beta_j = \frac{1-\epsilon_j}{\epsilon_j}$.
   (d) Update weights $w^{(i)} \leftarrow w^{(i)}\beta_j$ for wrongly classified examples.
   (e) Re-normalize $w^{(i)}$ to sum up to 1.
3. Strong classifier is computed as $SL(x) = \sum_j \underbrace{\ln(\beta_j)}_{\alpha_j} WL_j(x) > 0$.

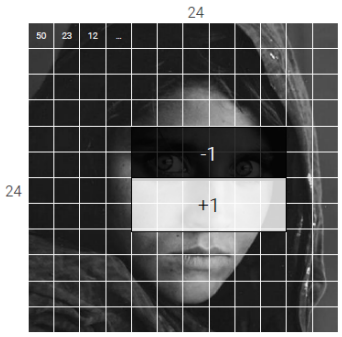
When considering binary weak learners, these predict a label in $\{-1, +1\}$. The specific strong learner will then output:

$$SL(x) = \begin{cases} 1 & \text{if } \sum_j \alpha_j WL_j(x) \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{43}$$

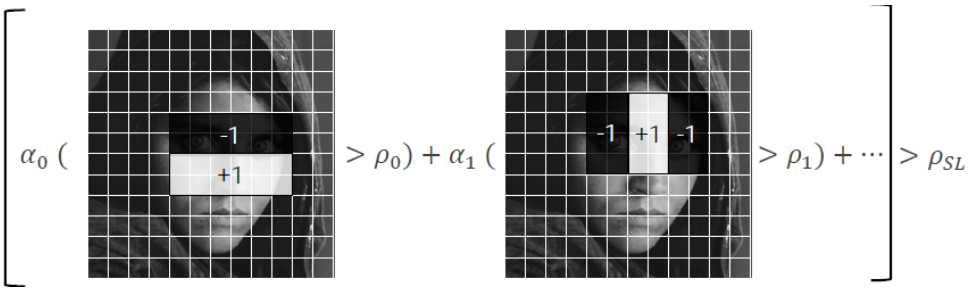
In the Viola Jones face detector, decision stumps are called **Haar-like features**. In particular, weak classifiers used to detect faces are simple rectangular filters which get applied at a fixed position within a 24×24 patch. The response of every weak learned is computed as:

$$WL_j(x) = \begin{cases} 1 & \text{if } s_j f_j \geq s_j \rho_j \\ -1 & \text{otherwise} \end{cases} \tag{44}$$

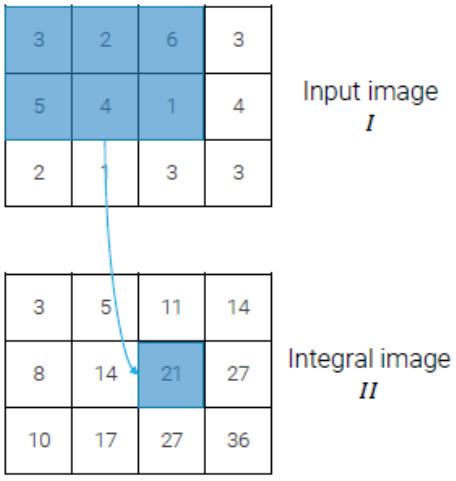
where $s_j \in \{-1, +1\}$ is a variable encoding a positive or negative sign, f_j is a feature, and ρ_j is a threshold. In this setting, one also have to choose the correct filters to apply.



Lastly, the strong learner output can be computed taking into account what have been produced by the different weak learners as:



- To speed-up the computation of rectangular features, the authors proposed the use of **integral images**:



$$II(i,j) = \sum_{i' \leq i, j' \leq j} I(i',j') \tag{45}$$

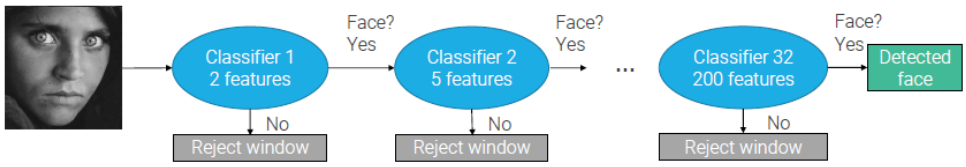
These can be computed with only one scan of the input image by using the recursive rule:

$$\begin{aligned} II(i,j) = II(i,j-1) + II(i-1,j) \\ - II(i-1,j-1) + I(i,j) \end{aligned} \tag{46}$$

Integral images allow the computation of rectangular filters in constant time.

- At test time, the strong classifier is applied to all spatial locations in the image: Viola-Jones is an example of a sliding window detector. However, faces are not necessarily 24×24 in test images: hence, multi-scale detection is necessary. Thanks to integral images, computation of larger features on larger patches requires the same constant time: hence multi-scale detection is achieved by scaling the detector instead of the input image, where the rectangular features are scaled accordingly. Even if each feature can be computed very fast thanks to integral images, there are still too many windows in an image to achieve real-time performance.

  To solve this problem, **cascade** was introduction. The key idea is to reject most of the background patches with a simpler classifier which can be run very fast and whose threshold is tuned on a validation set so that it does not discard any face:



10.1 Measuring Detector Performance

Usually, faces (or examples of a known class for a generic detector) will give rise to several, overlapping detections in test images. To check if two boxes overlap, one can measure the **intersection over union** (IoU) score:

$$IoU(BB_i, BB_j) = \frac{\text{area of intersection}}{\text{area of union}} = \frac{|BB_i \cap BB_j|}{|BB_i| + |BB_j| - |BB_i \cap BB_j|} \tag{47}$$

To obtain a single detection out of a set of overlapping boxes, one can perform **non-maxima suppression** of boxes:

1. Consider the highest scoring BB.
2. Eliminate all boxes with IoU greater than a threshold.
3. Repeat until all boxes have been tested.

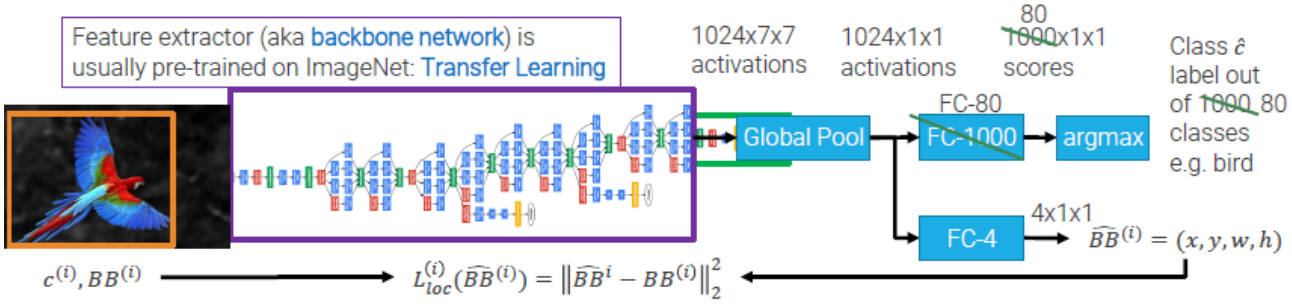
Moreover, the IoU is also used to measure the detector performance, by defining a threshold ρ_{IoU} to define a detection as correct. Indeed, a detection BB_i is considered a true positive (TP) if its overlap with a ground-truth bounding box BB_j^{GT} is greater than ρ_{IoU} (and the class is correct, when performing multi-class detection). Otherwise, BB_i will be considered a false positive (FP). Lastly, when measuring a detector performance, one would like to satisfy two conflicting requirements:

- Detect all objects of interest, i.e. achieve a high **recall**, $Re = \frac{TP}{|GT \text{ boxes}|}$.
- Do not select false objects in background regions, i.e. achieve a high **precision**, $Pr = \frac{TP}{|\text{detections}|}$.

One can influence the number of TPs and FPs, and therefore precision and recall, by varying the minimum confidence to accept a detection (ρ_{min}).

10.2 Deep Learning for Object Detection

If one can assume that only one object is present in the image, then the object detection problem simplifies to an object localization problem (i.e. predicting one class and one bounding box per image). To solve such problem, one can use any of the architectures used for image classification, adding a regression head that predicts the bounding box (i.e. four numbers) next to the standard classifier.



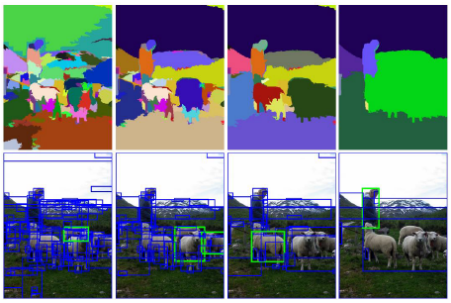
The final total loss is then computed as:

$$L^{(i)} = CE(\text{softmax}(scores^{(i)}), \mathbb{1}(c^{(i)})) + \lambda L_{loc}^{(i)}(\hat{BB}^{(i)}) \tag{48}$$

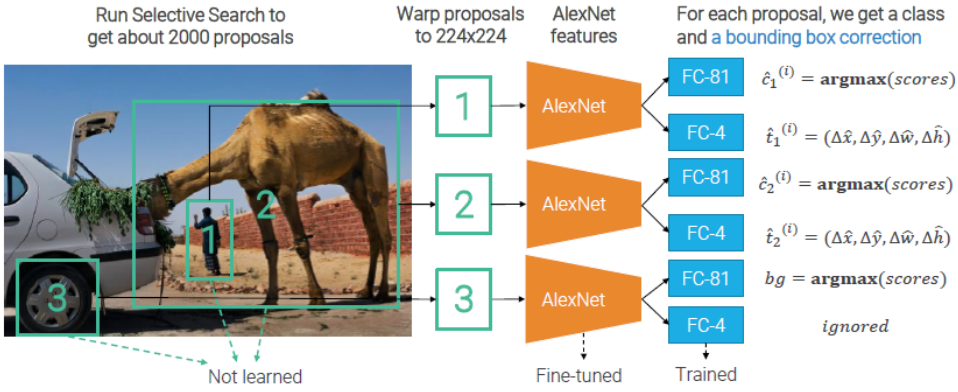
In this context, transfer learning can be used to speed-up training: one could either use pre-trained feature extractor or fine-tuning (i.e. train more the feature extractor, namely the first part of the network). This approach can also be used to detect multiple objects by applying a classification CNN as a sliding window detector. However, one has to consider too many boxes. A solution to this problem is **region proposal**.

10.2.1 Region Proposal

Regional proposal algorithms (e.g. selective search) inspect the image and attempt to find regions of an image that likely contain an object. These procedures first over-segment the image into highly uniform regions, then, based on similarity scores of colour, texture and size, iteratively aggregates them: the two most similar regions are grouped together, and new similarities are calculated between the resulting region and its neighbours, until the whole image becomes a single region.



An example of region proposal followed by deep learning techniques is given by **region-based CNNs** (**R-CNN**):



In particular:

- Given the selection search bounding box $BB_{SS} = (x_{SS}, y_{SS}, w_{SS}, h_{SS})$ the network predicts a correction $\hat{t} = (\Delta\hat{x}, \Delta\hat{y}, \Delta\hat{w}, \Delta\hat{h})$. The output box is then computed as:

$$BB = (\underbrace{x_{SS} + w_S S\Delta\hat{x}, y_{SS} + h_{SS}\Delta\hat{y}}_{\text{translate relative to box size}}, \underbrace{w_{SS}\exp(\Delta\hat{w}), h_{SS}\exp(\Delta\hat{h})}_{\text{log-space scaling}}) \tag{49}$$

- Given the selective search box $BB_{SS}^{(i)}$ corresponding to training example $x^{(i)}$, and the most overlapping ground truth bounding box $BB_{GT} = (x_{GT}, y_{GT}, w_{GT}, h_{GT})$, the targets $t = (\Delta x, \Delta y, \Delta w, \Delta h)$ for the loss of the regression head are computed as:

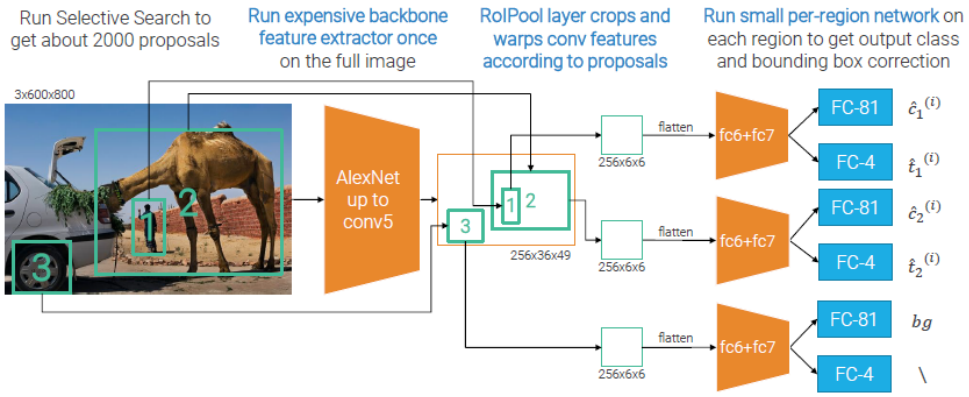
$$\Delta x = (x_{GT} - x_{SS}) \tag{50}$$
$$\Delta y = (y_{GT} - y_{SS}) \tag{51}$$
$$\Delta w = \ln(w_{GT}/w_{SS}) \tag{52}$$
$$\Delta h = \ln(h_{GT}/h_{SS}) \tag{53}$$

  The total loss is the usual multi-task loss:

$$L^{(i)}(\theta; x^{(i)}, c^{(i)}, BB_{GG}^{(i)}) = CE(\text{softmax}(scores^{(i)}), \mathbb{1}(c^{(i)})) + \lambda I[c^{(i)} \neq bg]\|\hat{t}^{(i)} - t^{(i)}\|_2^2 \tag{54}$$

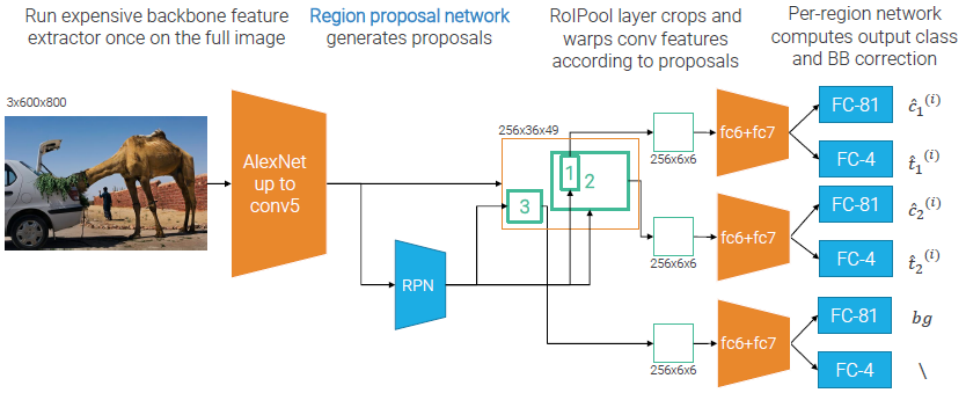
- Warping adds some context while deforming the regions: given a region selected by selective search, this region is warped to 224×224 to match AlexNet input size. Before warping, the region size is expanded to a new size that will result in 16 pixels of context in the warped frame:

- The main limitation of vanilla R-CNN is that it is slow. In particular, it consists of thousands of CNN forward passes per image. The solution to this is to apply (most of) the CNN before warping, obtaining a **fast R-CNN**:

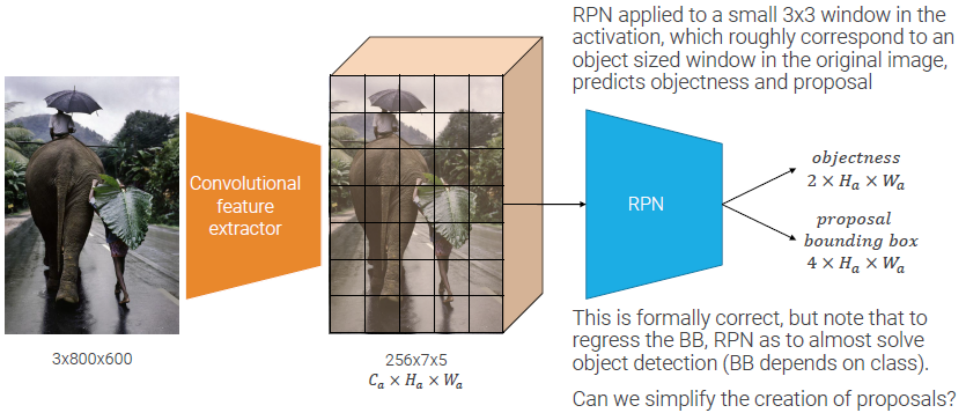


where the RolPool layer converts the activations inside the RoI, corresponding to rescaled selective search regions, into activations with fixed spatial dimensions (the ones required by the remaining layers of the network). This type of layer internally computes a max-pool operation.

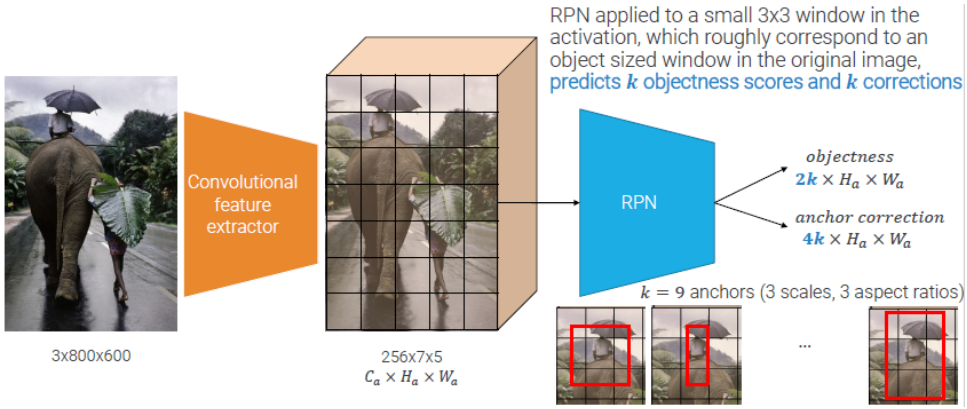
- The main bottleneck of fast R-CNN is selective search: the algorithm is slow, and not learned by the network. To solve the problem one can define a **region proposal network**, which generates proposals:



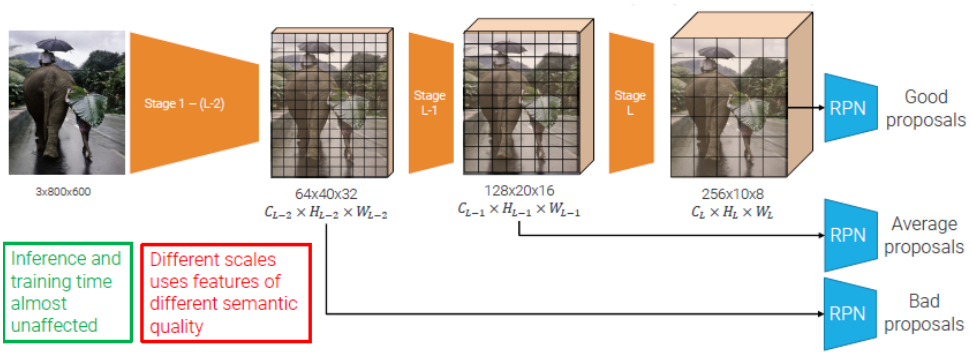
where this region proposal network will output: an objectness scalar value, indicating the degree of belief that a specific window of the activation of the feature extractor contains an object, and a proposal bounding box for the specific object.



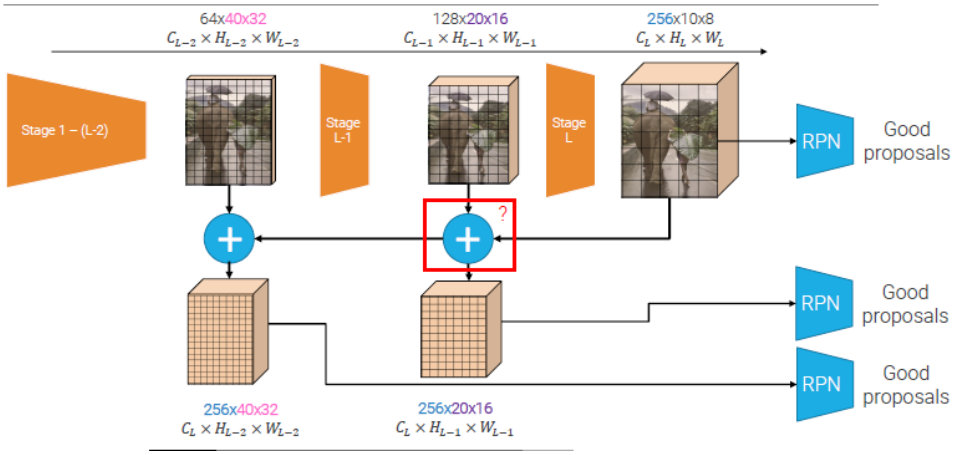
This approach is formally correct, but to regress the bounding box, the region proposal network has to almost solve the object detection task. This can be simplified by outputting a correction to an input proposal t (**anchor correction**), as done in vanilla R-CNNs. This type of region proposal network will output k objectness scores and k corrections:



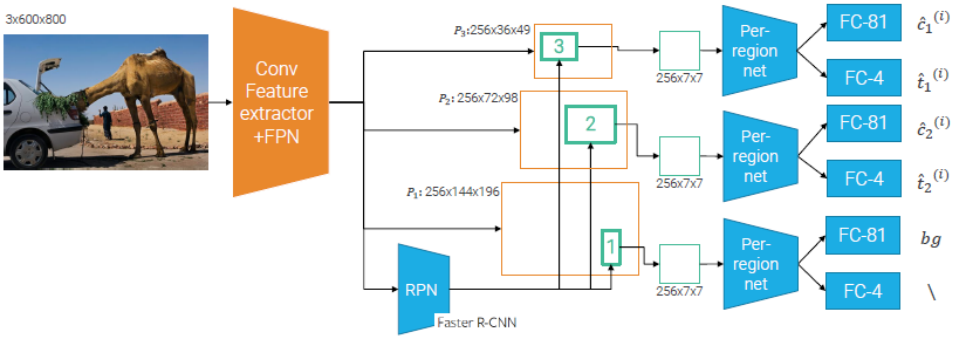
- This last type of R-CNNs can process only the last activation of the convolutional feature extractor, which usually is a semantically rich signal (i.e. it includes higher level features than previous activations), but very coarse in spatial resolution: even if small scale anchors are provided, it may miss objects smaller than the grid size. To attain scale-invariant detection and detect large and small objects alike, the main approach before CNNs was based on image pyramids (SIFT). Now, CNNs do produce a pyramid of features, but with different semantic quality at different depths:



To achieve always the same quality and to exploit the pyramid of features approach, **feature pyramid networks** (**FPNs**) can be defined. In particular, such networks hallucinate higher resolution features with high semantic value by merging coarser but higher-level features with less effective but more spatially localized features, with limited computational overhead:



This new approach gives rise to **faster R-CNNs with FPN**:



YOLOv3 is an example of this type of network. In particular, it uses a custom backbone optimized to have a good trade-off between classification accuracy and speed, and uses the idea of multi-scale detections on features with different spatial resolutions, as in FPN. Lastly, it concatenates activations from different stages instead of summing them.

When considering multi-scale networks such as faster R-CNNs with FPN, multi-scale training and fine-tuning are usually used as the main training approach. This approach forces the network to learn well across a variety of input dimensions. In particular:

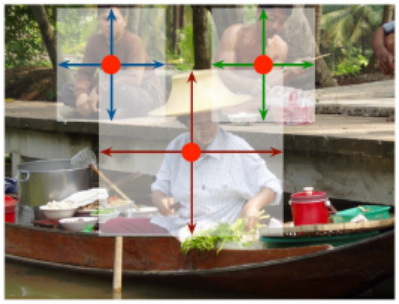
- The backbone is trained on ImageNet with a fixed input resolution (e.g. 224×224).
- Fine-tune the backbone for ten epochs on ImageNet while changing the resolution of the input images every ten epochs.

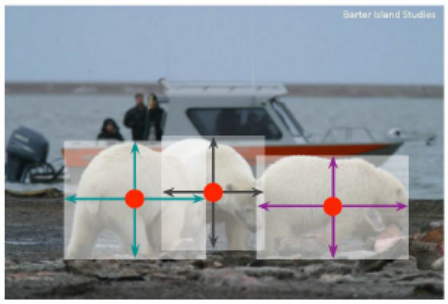
10.2.2 Limits of Anchors

Anchor-based detectors have three main limits:

- Anchors are a brute force approach to detection, in which one enumerates all possible boxes.
- Anchors produce a lot of duplicated entries for an object, which then must be post-processed with NMS.
- There is the need to assign anchors to the ground truths for training. This is usually done manually via hand-crafted rules.

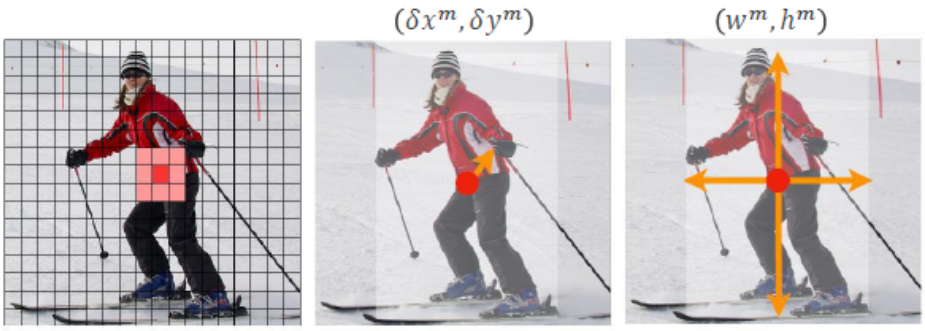
To overcome these limits, one could treat objects as points, and then regress the size of the objects or other properties.





Keypoint detection can be carried out using CenterNets. In particular, given an image of size $3 \times H \times W$, the aim of the network is to produce a heatmap $\hat{Y}$ with values in $[0, 1]$ and size $C \times \frac{H}{R} \times \frac{W}{R}$ with output stride R. To recover the discretization error caused by the output stride, the network also predicts an offset $\hat{O}$ of size $2 \times \frac{H}{R} \times \frac{W}{R}$ for each centre point. Finally, the network also predicts the bounding box size, $\hat{S}$, of size $2 \times \frac{H}{R} \times \frac{W}{R}$. At inference time, given a spatial local maxima in the channel $\hat{Y}_c$ at

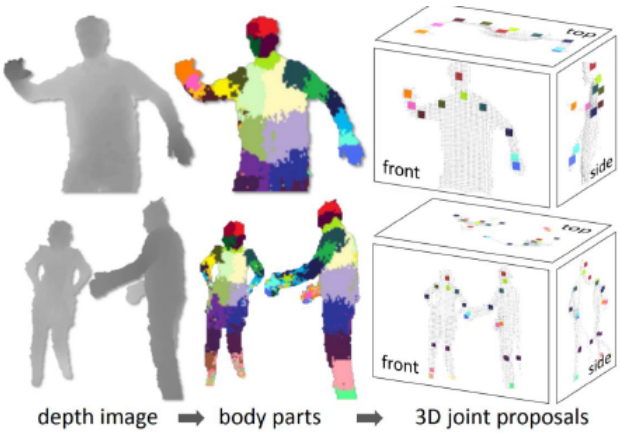
position (x^m, y^m), the box centered at $(x^m + \delta x^m, y^m + \delta y^m)$ of size (w^m, h^m) and class c is detected, without any further post-processing.



Keypoints can be seen as a special case of anchor: a single, shape-agnostic anchor. In this case, there exists one box per object without the need to perform NMS.

11 Semantic Segmentation

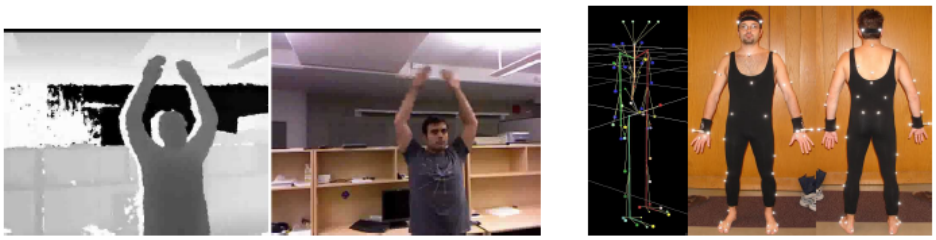
Semantic segmentation is the problem which requires to output a category c_{uv} for each pixel $p = (u, v)$, $c_{uv} \in [1, \ldots, C]$, of an RGB image. One of the most successful applications of semantic segmentation has been the human pose estimation algorithm running on data collected by the Kinect controller for the Xbox gaming console. The key design goals of human pose estimation are: computational efficiency, robustness (i.e. it must work on a frame-by-frame basis, tracking cannot be used to guide the estimation). The main idea is to solve pose estimation by learning to segment a depth image of one or more users into body parts.



The main pipeline is the following:

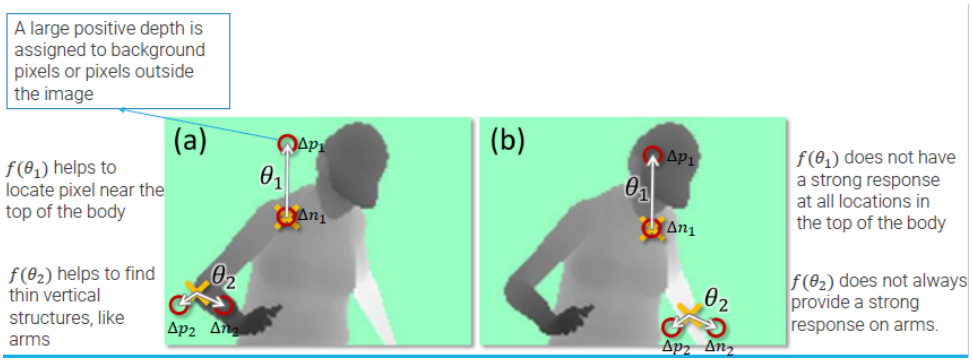
1. Capture the depth image and remove background.
2. Classify each pixel into a body part with a **random forest** classifier processing depth features.
3. Find local modes to define joint positions.

In general, there exist distinct body parts for left and right, allowing the classifier to disambiguate the left and right sides of the body. Moreover, the generation of realistic RGB images to learn poses is made difficult by the huge colour and texture variability induced by clothing, hair, and skin. The use of a depth camera significantly reduces the space of variations. To obtain a realistic and varied set of poses, a mixture of real **motion capture data** (mo-cap data), augmented in a synthetic pipeline, is used.



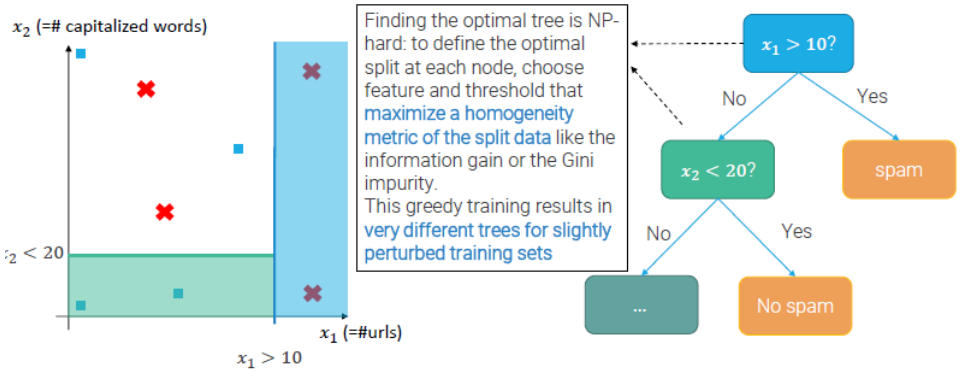
The main features that are computed are called **depth comparison features**. At each pixel location p, simple depth comparison features are computed given the depth of the image, D, and offsets θ:

$$f(p; D, \theta = (\Delta p, \Delta n)) = D\left(p + \frac{\Delta p}{D(p)}\right) - D\left(p + \frac{\Delta n}{D(p)}\right) \tag{55}$$

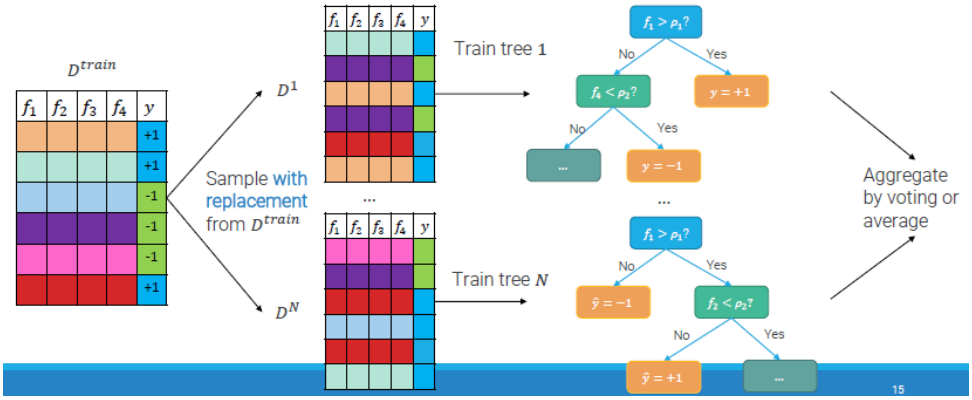


11.1 Random Forests

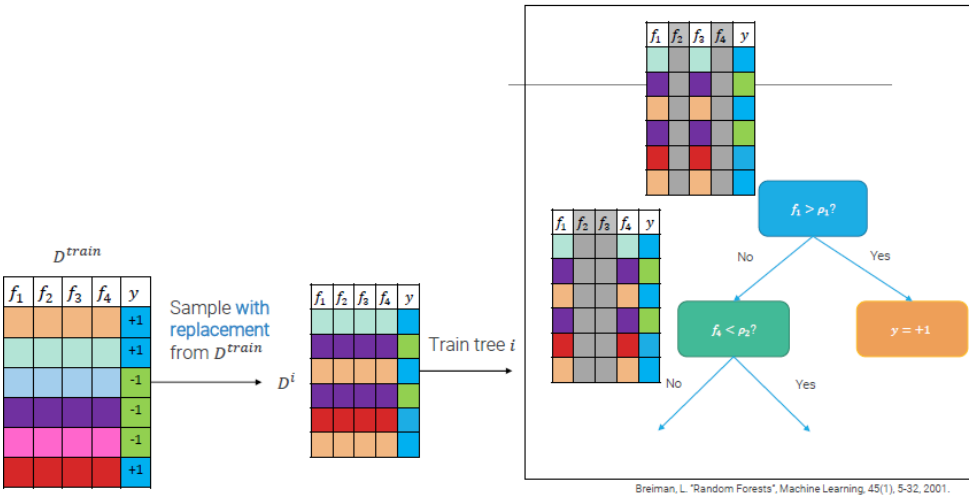
Decision trees are the building blocks of random forests:



Decision trees are known to be robust classifiers but generalize poorly: when tuned to rach low bias, they suffer from high variance. **Random forests** are another example of ensembles of simple classifiers, in this case of decision trees, which train N trees on random views of the dataset and average their precisions. If the trees are uncorrelated, different trees will make different errors: the average of those errors will approach zero as more trees are added to the ensemble, reducing variance. In particular, **bagging** is what usually is used to train random forests. The main idea behind bagging is to train different trees on bootstrapped replicas of the same dataset.



Yet, if a subset of features is particularly predictive, the resulting trees wil likely use them and will still be highly correlated. Random forests reduce the correlation by selecting a random subset of features to define the split at each node:



11.2 Performance Metrics for Segmentation Algorithms

On way to measure the performances of a model is to use the **generalized intersection over union**. Indeed, intersection over union can be generalized to segmentation tasks: for a class $c = 1, \ldots, C$:

$$IoU_c = \frac{\text{area of intersection}}{\text{area of union}} \tag{56}$$

where:

$$\text{area of intersection} = TP_c = \sum_{images} \text{number of pixels where } y_{uv} = c \text{ and } \hat{y}_{uv} = c \tag{57}$$

and:

$$\text{area of union} = \sum_{images} (\text{number of pixels where } \hat{y}_{uv} = c + \text{number of pixels where } y_{uv} = c) - TP_c \quad (58)$$

To compute the $mIoU$ score for a dataset, which is the main measure to rank semantic segmentation algorithms, one can average IoU_c over classes:

$$mIoU = \frac{1}{C} \sum_{c=1}^{C} IoU_c \quad (59)$$

Other measures are:

- **Pixel accuracy**, the fraction of pixels correctly classified:

$$\frac{\sum_c TP_c}{\text{number of pixels in the dataset}} \quad (60)$$

- **Mean accuracy**, the average of the accuracy for each class:

$$\frac{1}{C} \sum_c \frac{TP_c}{\text{number of pixels of class } c \text{ in the dataset}} \quad (61)$$

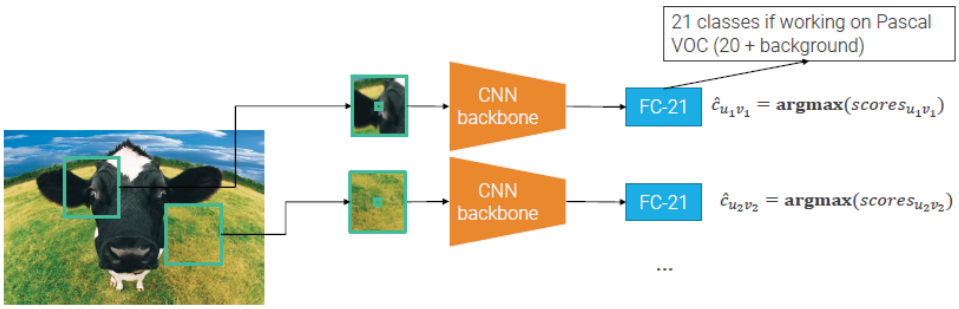
- **Frequency weighted intersection over union**, the weighted average of IoU_c for each class, with weights given by the frequency of a class in the dataset:

$$\sum_c \frac{\text{number of pixels of class } c}{\text{number of pixels in the dataset}} IoU_c \quad (62)$$

11.3 Deep Learning Architectures for Semantic Segmentation

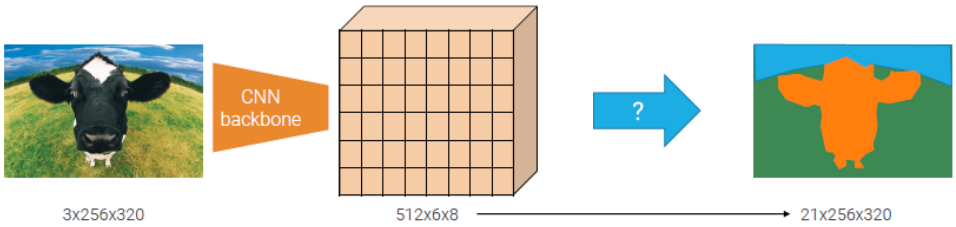
Some examples of deep learning architectures for semantic segmentation are the following:

- **Slow R-CNN**:

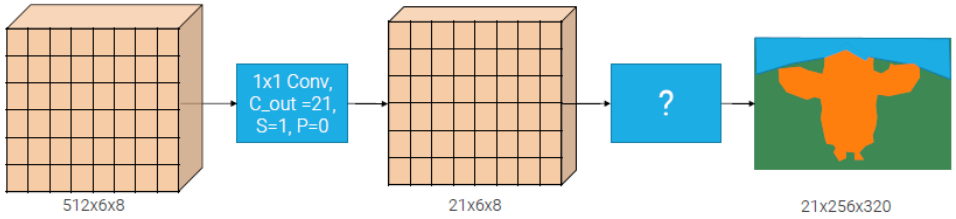


This network must slide a window at all possible positions: there is no region proposal since it must process every pixel. Thus, it is very slow.

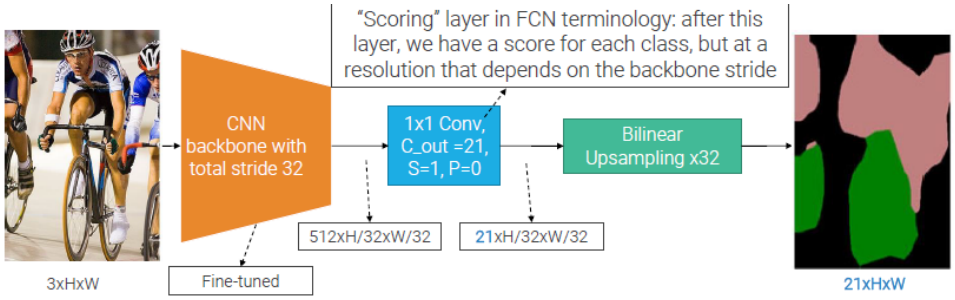
- **Fully convolutional network** (**FCNs**):



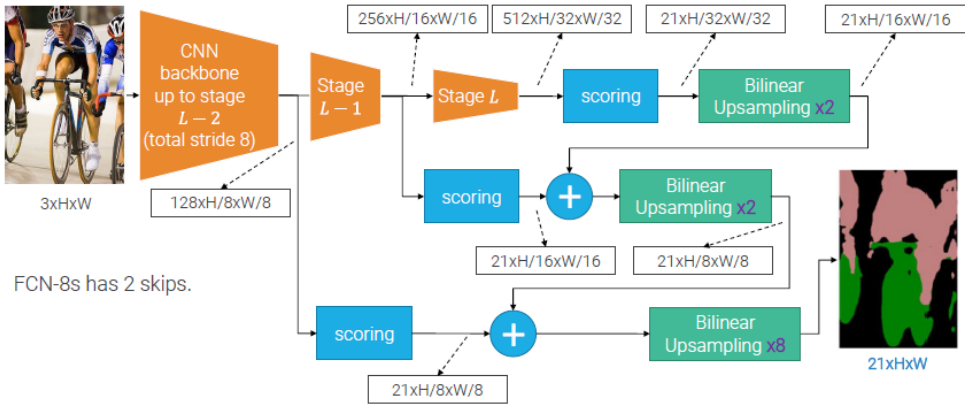
where the output can be computed with a 1×1 convolution with a number of kernel equal to the number of total classes, and then by using an upsampling operation:



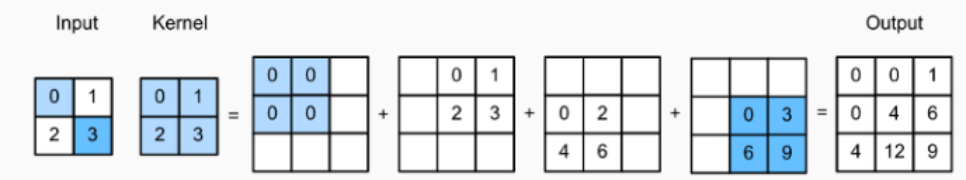
Some ways of performing upsampling is to use standard, not-learned image processing operators such as nearest neighbour or bilinear interpolation:



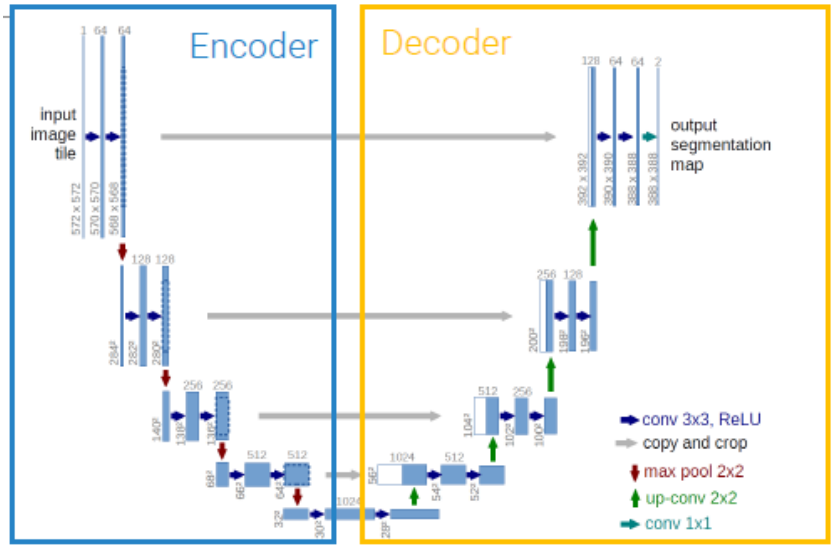
The problem with this naive approach is that without learning a non-linear upsampling transformation, one can only uniformly spread the coarse information in the final convolutional activation, obtaining very coarse masks. A solution to this is to upsample multiple activations at different resolutions (like in FPN). An example of this is **FCN-8s**:



Another way of performing upsampling is by using **transposed convolutions**. Transposed convolutions are used to upsample the input feature map to a desired output feature map using learnable parameters. For example:

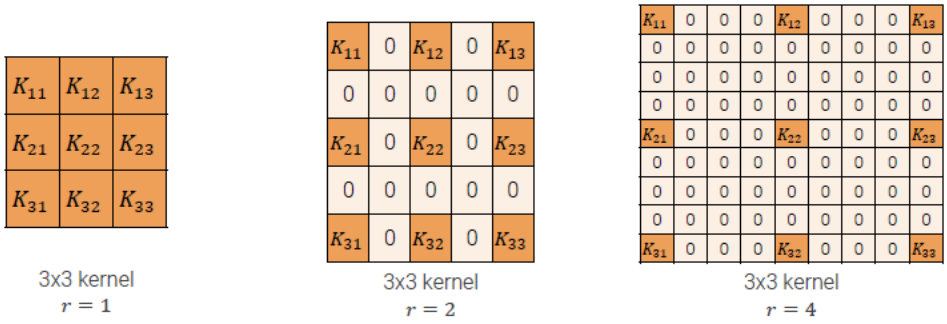


- **U-net**. This network extends the idea of skips from FCNs to create a full-fledged decoder, which has roughly a symmetric structure with respect to the encoder:

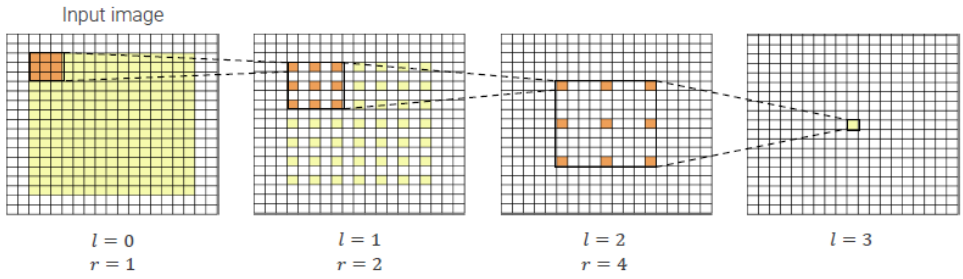


Every activation produced by a stage of the backbone (encoder) has a skip connection with the corresponding level of the decoder. Skip connections use concatenation instead of summation, and 2×2 stride-2 transposed convolutions are used to upsample the activations in the decoder, while halving the number of channels. Normal 3×3 convolutions are used in the decoder as well.

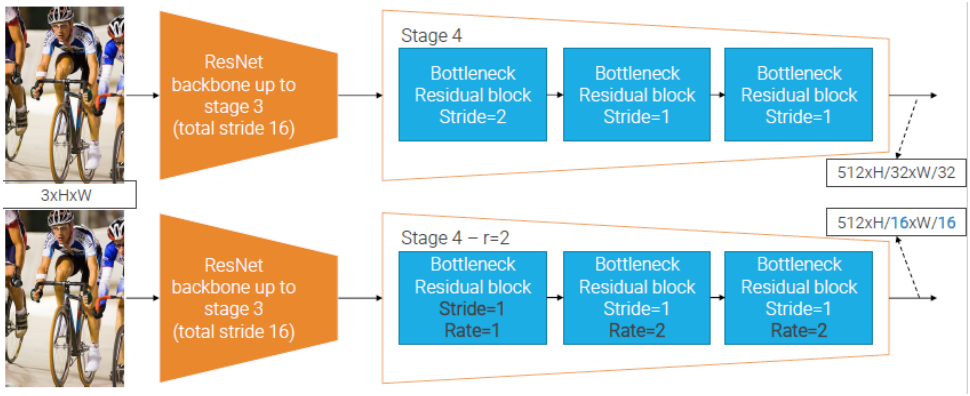
- **Dilated ResNets**. In standard ResNets, the more proceeds with adding convolutional stages to the backbone of the network, the more the receptive field is enlarged and resolution is lost. To solve this problem, and so to have rich features with large spatial resolutions and large receptive fields, one could insert dilated convolutions into ResNets. Dilated convolutions expose an additional parameter, the dilation rate r. These convolutions insert holes between filter weights.



If one stacks dilated convolutions with an exponentially increasing dilation rate $r_l = 2^l$, the effective receptive field grows exponentially with the number of layers, while the number of parameters grows linearly, and the resolution is not reduced. In general at level l the receptive field of an activation entry will be $(2^{l+1} - 1) \times (2^{l+1} - 1)$.



One can modify ResNet so to create a dilated backbone: when the stride from a convolution or pooling layer is removed, the dilation rate is doubled in all subsequent convolutions. By iterating this process, one could process images at full resolution. However, it becomes too computationally expensive due to the large resolution of the activations.

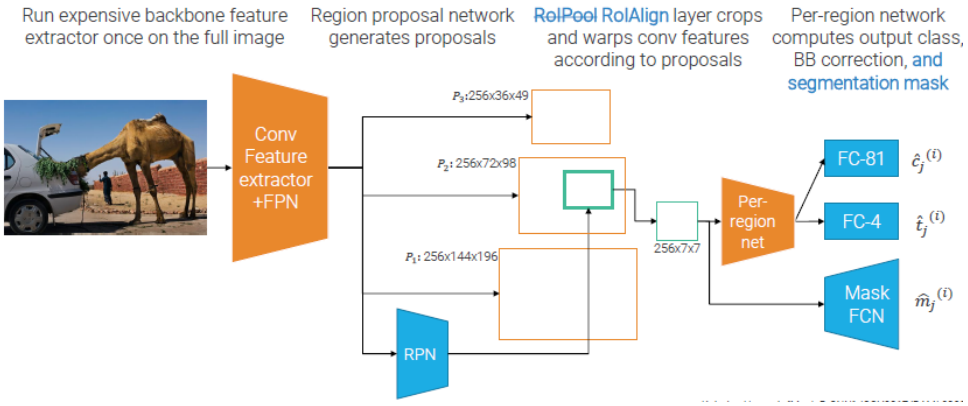


DeepLab is an example of ResNet with dilated convolutions.

11.4 Instance Segmentation

Instance segmentation is about detecting all instances of the objects of interest in an image and classifying them. Moreover, image segmentation must segment instances of the sought object from the background. To solve this problem one could use the following networks:

- **Mask R-CNN**, which is based on the faster R-CNN with FPN:



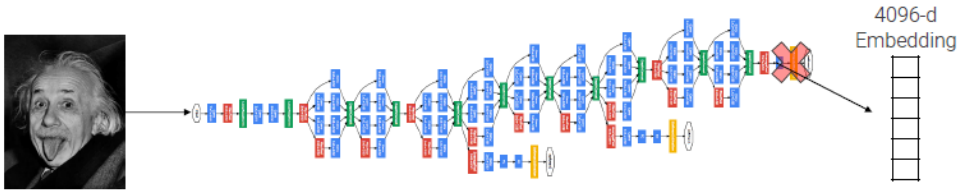
This network is a simple modification of the faster R-CNN. In particular, it adds a branch to the second stage to predict the segmentation mask on each RoI, realized as a small fully-connected convolutional network. Moreover, it improves the RoIPool layer by using a RoIAlign layer, which crops and warps convolution features according to the proposals.

11.5 Metric Learning

In general, by removing the classification head of an image classification network one can compute a low-dimensional representation (a low-dimensional embedding) or the input images. At this point, one could perform a nearest neighbour search on such embeddings to get semantically similar images as neighbours.

11.6 Face Recognition and Face Verification

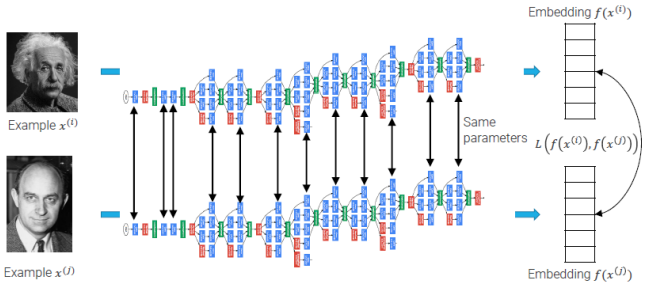
Given a query face, **face recognition** is the problem of recognizing such face. This can be seen as a one-to-many matching problem. Face recognition can be treated as a classification problem, using a classification network. However, this approach suffers from scalability and flexibility problems. For example, whenever a user is added or deleted from the training set, then the entire network must be re-trained. The idea to solve such problems is to rely on transfer learning, and to re-use the representation learned, only once, on a reasonably sized dataset as feature extractor, with a k-NN classifier on the embeddings.



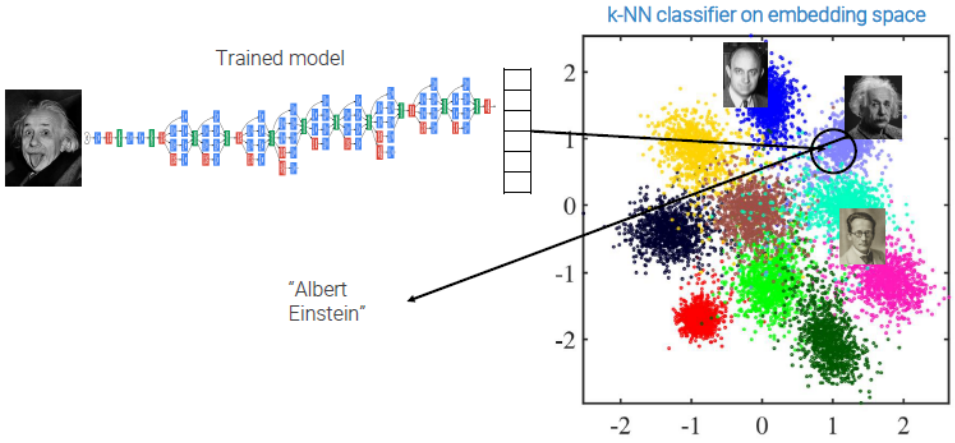
Face verification is the problem of confirming whether two images depict the same person or not. Usually, this problem is solved by learning a distance/similarity function between images, and a threshold. The idea of **metric learning**, or **similarity learning**, is to use a specific loss at training time so to guide the feature extractor to favour a clustered structure of the embedding such that:

- The distance between faces of the same person is minimized.
- The distance between faces of different persons is maximized.

In this setting, the loss function must reason upon two examples. A specific type of network which can use such function is the **siamese network**:



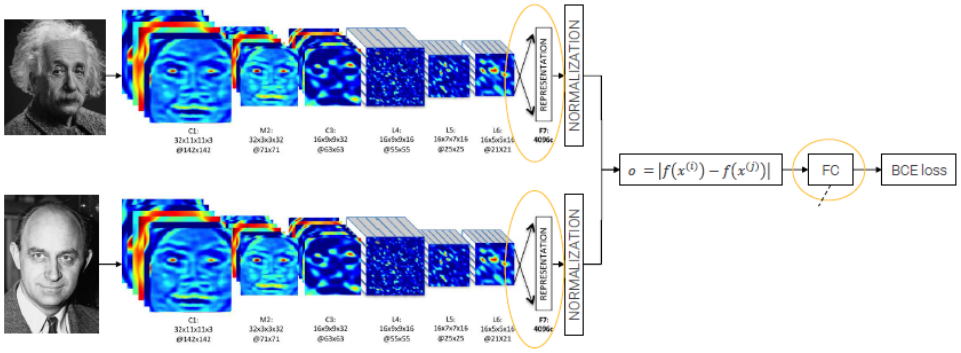
This model uses two copies of the same network with shared weights. On can then apply a k-NN classifier on the produced embeddings:



Examples of networks used is siamese models are:

- **DeepFace**, which is a network that is first trained on a classification problem. Images fed to the network are pre-processed by fitting a 3D model to extract frontal faces. The learned representations are further normalized by dividing each component by the maximum seen at training time, and $L2$-normalized:

$$f(x^{(i)}) = \frac{\bar{G}(x^{(i)})}{\|\bar{G}(x^{(i)})\|_2} \quad \text{where} \quad \bar{G}(x^{(i)})_k = \frac{G(x^{(i)})_k}{\max(G(x^{(1:N)})_k, \epsilon)} \tag{63}$$



The network can be fine-tuned by training the circled layers. The FC layer is actually a vector of 4096 weights, α_k, which induces a distance d in the embedding:

$$d(x^{(i)}, x^{(j)}) = \sum_k \alpha_k |f(x^{(i)}) - f(x^{(j)})| \tag{64}$$

However, DeepFace still casts face verification as classification, which may not generalize well in open-world problems. A loss enforcing directly a clustered embedding structure should:

- Make $d(x^{(i)}, x^{(j)})$ small, if $x^{(i)}$ and $x^{(j)}$ depict the same person.
- Make $d(x^{(i)}, x^{(j)})$ large, if $x^{(i)}$ and $x^{(j)}$ depict two different persons.

If the Euclidean norm is used, $d(x^{(i)}, x^{(j)}) = \|f(x^{(i)}) - f(x^{(j)})\|_2$, a simple loss that capture both requirements is:

$$L(f(x^{(i)}), f(x^{(j)})) = \begin{cases} \|f(x^{(i)}) - f(x^{(j)})\|_2^2 & \text{if } y^{(i,j)} = 1 \text{ (i.e. same person)} \\ -\|f(x^{(i)}) - f(x^{(j)})\|_2^2 & \text{if } y^{(i,j)} = 0 \text{ (i.e. different person)} \end{cases} \tag{65}$$

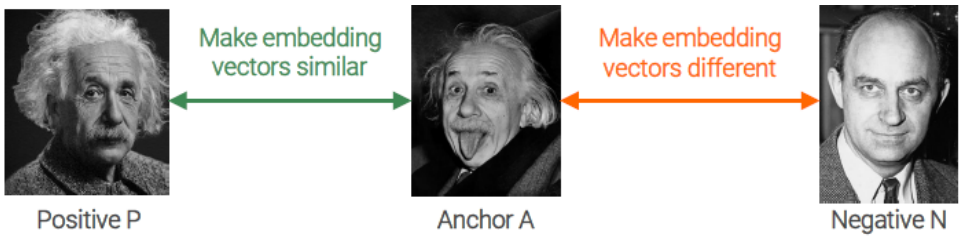
A formulation which is less prone to overfitting uses a margin m to stop pushing clusters of different classes apart from each other:

$$L(f(x^{(i)}), f(x^{(j)})) = \begin{cases} \|f(x^{(i)}) - f(x^{(j)})\|_2^2 & \text{if } y^{(i,j)} = 1 \text{ (i.e. same person)} \\ \max(0, (m - \|f(x^{(i)}) - f(x^{(j)})\|_2)^2) & \text{if } y^{(i,j)} = 0 \text{ (i.e. different person)} \end{cases} \tag{66}$$

which can be re-written as:

$$L(f(x^{(i)}), f(x^{(j)})) = y^{(i,j)} \|f(x^{(i)}) - f(x^{(j)})\|_2^2 + (1 - y^{(i,j)}) \underbrace{\max(0, (m - \|f(x^{(i)}) - f(x^{(j)})\|_2)^2)}_{\text{Hinge loss}} \tag{67}$$

To obtain a more effective embedding, one should consider three images:



The **triplet loss** optimizes the embedding to better fulfil both requirements at once:

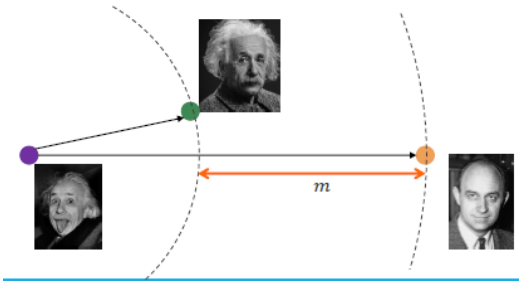
$$\|f(P) - f(A)\|_2^2 < \|f(N) - f(A)\|_2^2 \tag{68}$$

However, this simple definition if not robust (i.e. it does not guarantee inter-class distances to be large), and it also risk training collapse (namely, if the embedding is a constant value, e.g. all 0s, the criteria is satisfied). To solve this, one can use the concept of margin m:

$$\|f(P) - f(A)\|_2^2 + m < \|f(N) - f(A)\|_2^2 \quad \Rightarrow \quad \|f(P) - f(A)\|_2^2 - \|f(N) - f(A)\|_2^2 + m < 0 \tag{69}$$

which leads to:

$$L(A, P, N) = \max(0, \|f(P) - f(A)\|_2^2 - \|f(N) - f(A)\|_2^2 + m) \tag{70}$$



The most important part in training embedding with the triplet loss is to effectively form the triplets. In particular, the choice of the negative examples, N, is key. For most triplets, the constraint $\|f(P) - f(A)\|_2^2 + m < \|f(N) - f(A)\|_2^2$ will be already satisfied. One usually wants to compute the loss only on the so called active triplets, which will contribute to learning. Hence, a large mini-batch of B samples is formed by picking a fixed number of images for D identities, to ensure that a significant representation of the anchor-positive distance can be formed, and randomly sampled negatives are added to complete the batch. Then, triplets are formed on-line by:

1. Creating all possible (A, P) pairs for each identity.
2. Creating a triplet for each **semi-hard negative** N if:

$$\|f(P) - f(A)\|_2^2 < \|f(N) - f(A)\|_2^2 z \|f(P) - f(A)\|_2^2 + m \tag{71}$$

   Namely, if the negative example is within the margin.

12 Depth Estimation

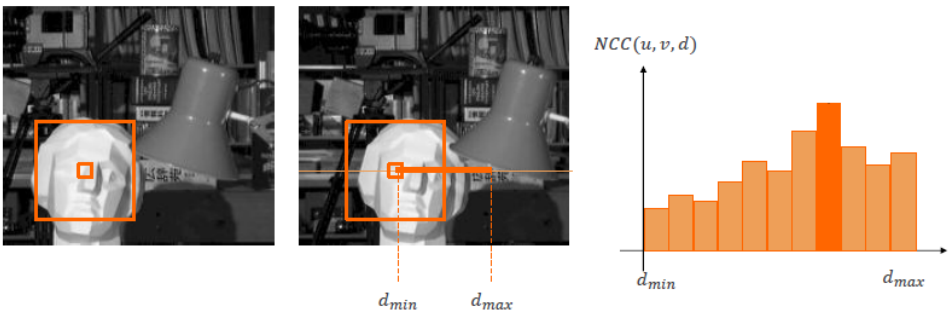
Given an ideal stereo setup, the depth of each point can be obtained by solving the **correspondence problem** along rows. If point (u_L, v_L) in the left image, and point $(u_R, v_R = v_L)$ in the right image are the projections of the same 3D point, its distance from the camera, z, can be recovered from the disparity $d = u_L - u_R$ as:

$$z = \frac{b \cdot f}{d} \tag{72}$$

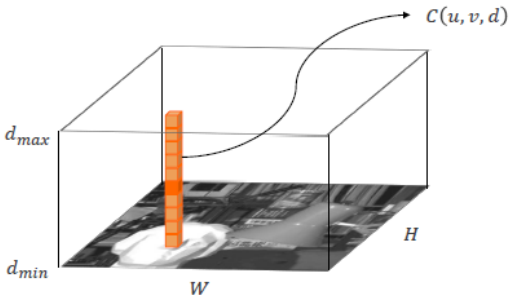
Given one patch for the left image and one patch for the right image, one can compute disparity by computing the matching cost (i.e. the normalized cross-correlation between pixels)

$$NCC(u, v, d) = \frac{\sum_{m=-W}^{+W} \sum_{n=-W}^{+W} L(u+m, v+n) R(u+m+d, v+n)}{\sqrt{\sum_{m=-W}^{+W} \sum_{n=-W}^{+W} L(u+m, v+n)^2} \sqrt{\sum_{m=-W}^{+W} \sum_{n=-W}^{+W} R(u+m+d, v+n)^2}} \tag{73}$$

and then by taking $\hat{d}(u, v) = \operatorname{argmax} NCC(u, v, d)$:



The 3D data structure containing all matching costs for all pixels at all evaluated disparities is referred to as the **cost volume** or **disparity space image** (DSI).

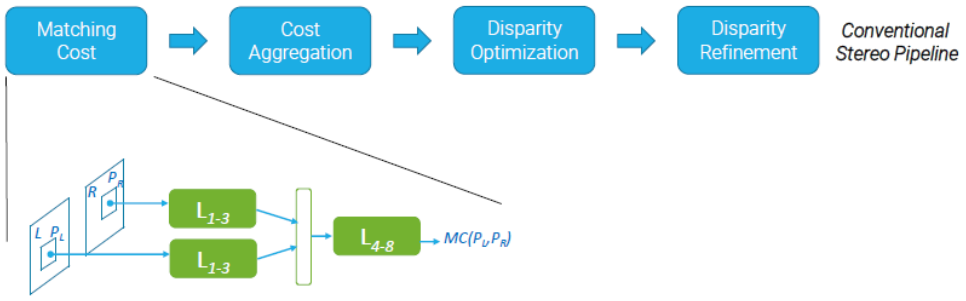


A related problem is **optical flow**: estimating the apparent motion of all pixels between two frames. The key difference is that cameras/frames are not calibrated, and motion can happen in every direction, and can be due to camera motion between frames, object motion or both.

12.1 Deep Learning for Stereo

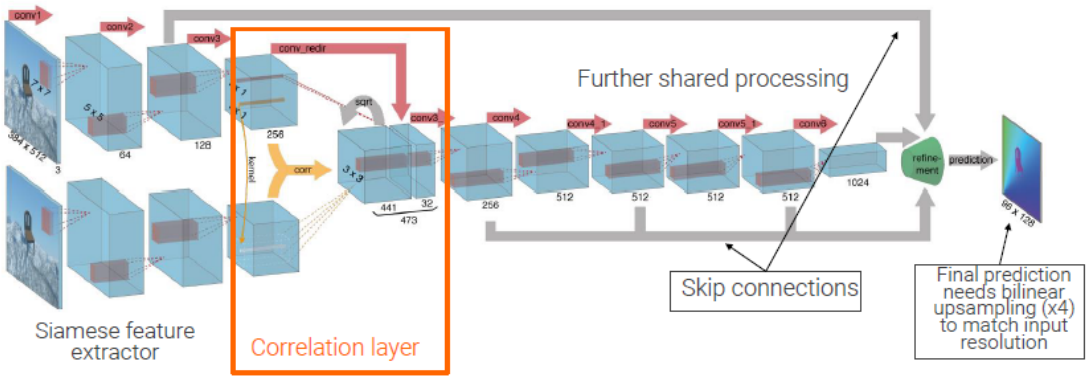
Below is the evolution of deep learning for stereo vision:

- The first part of the traditional stereo pipeline to be solved with deep networks (**MC-CNN**) has been the computation of the matching cost:



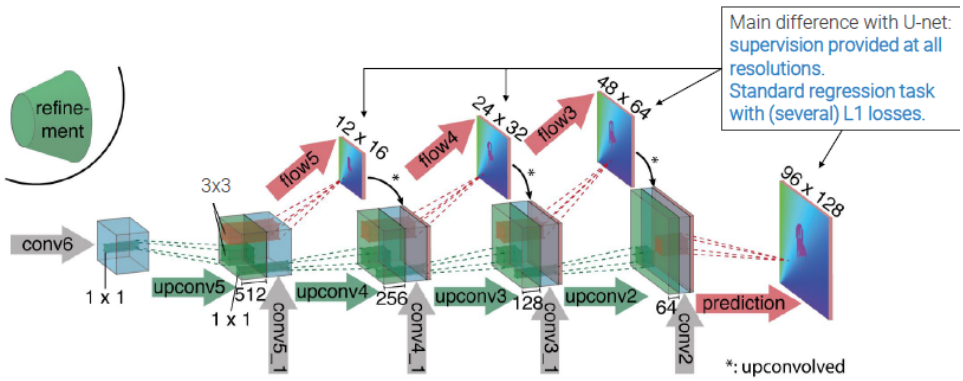
  In particular, a siamese network is trained to extract features from the two patches, then features are concatenated, and a classification head estimates the cost patches as being correspondent or not.

- End-to-end solutions to predict directly the disparity map (or the flow) from a pair of images were proposed right after MC-CNN. The first proposal was for optical flow (**FlowNet**), which was then re-casted to solve stereo (DispNet) by limiting the search space for matches. FlowNet is defined as follows:

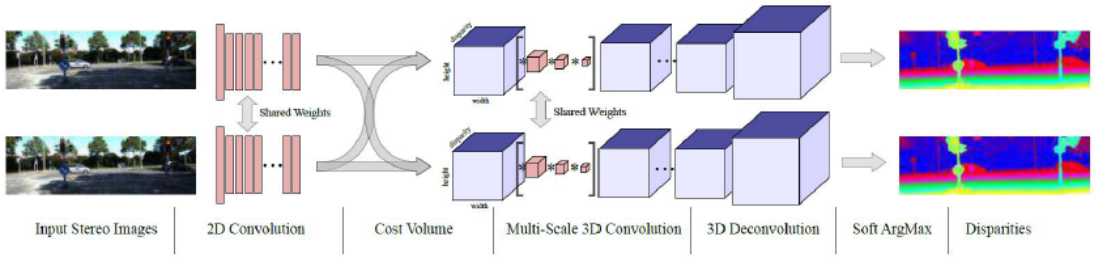


  In particular, the correlation layer aims at mimicking the computation of matching costs performed in traditional stereo/flow pipelines. This layer has no learnable parameters. Moreover,

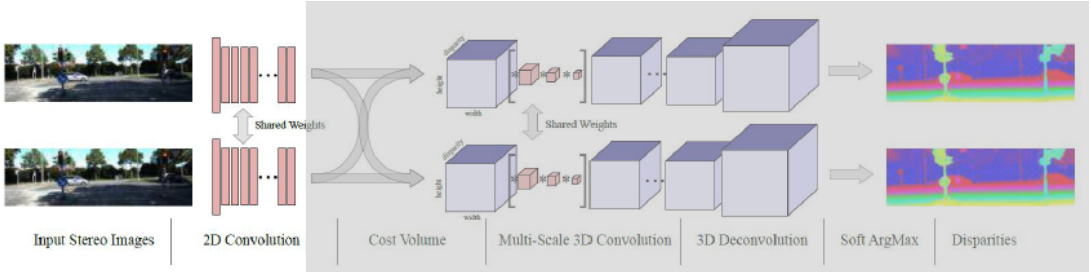
the final refinement portion of the network encapsulates a full U-net-like decoder, which is used to produce the output map.



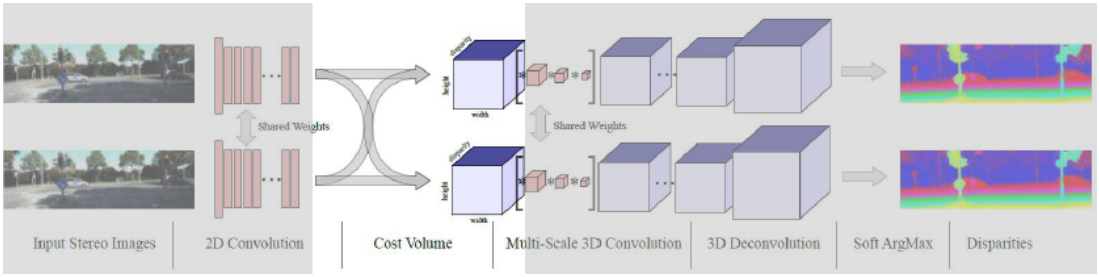
- **DispNet** follows the same organization of FlowNet, but only $H \times W \times 1$ disparity maps are produced. Notably, though, it introduces convolutions in between up-convolutions in the decoder, to obtain smoother results.
- **GCNet** makes a step further by explicitly building and processing a cost volume through 3D convolutional layers, and then by obtaining a disparity map by applying the traditional $\hat{d}(u, v) = \text{argmax}\, NCC(u, v, d)$ (winner-takes-all, WTA) selection over the volume:



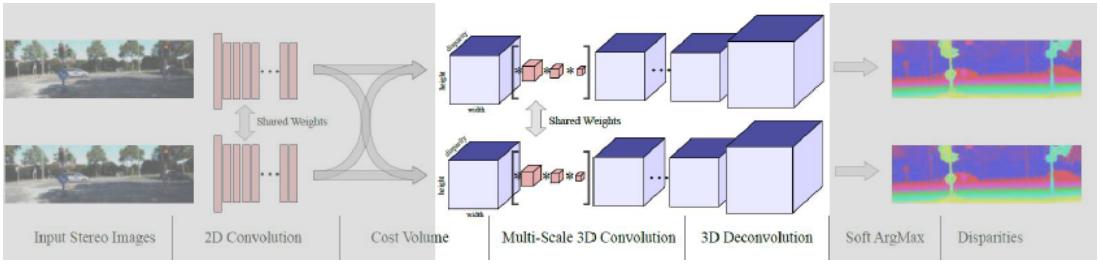
In particular, a ResNet backbone is used to extract effective features at each spatial location:



Then, given the two feature maps f^1 and f^2, a 4D structure, which is then used to compute the cost volume, is built by stacking along the fourth dimension, at position (u, v, d), the feature vectors at location $(u, v) \in f^1$ and the feature vectors at location $(u + d, v) \in f^2$. This is similar to what the correlation layer of DispNet does, but it uses concatenations instead of dot products:



Given the 4D input, a 3D U-net-like encoder-decoder further processes and optimizes the volume. This step is the key innovation of GCNet, which aims at mimicking the matching cost, cost aggregation, and disparity optimization steps of a traditional pipeline by learning the best data-driven way to perform them:



Lastly, being the WTA selection not differentiable, the authors proposed a soft version of it:

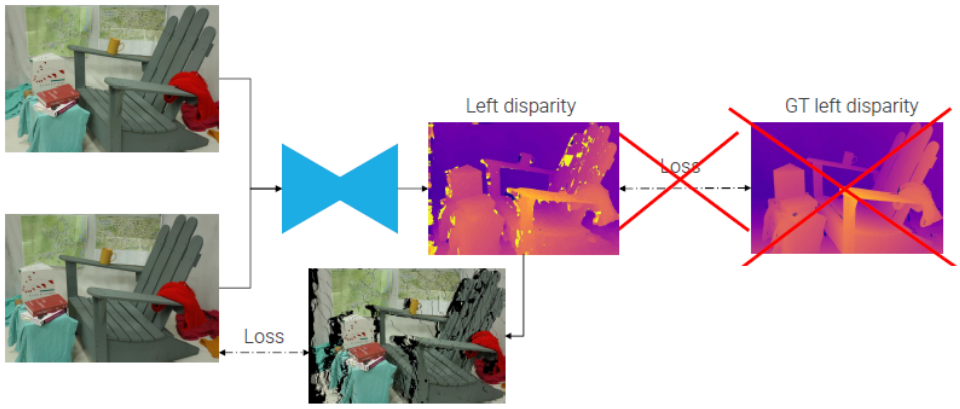
$$\hat{d} = \sum_{d=0}^{D-1} d \cdot \text{softmax}(-c_{uv}(d)) \tag{74}$$

where $c_{uv}(d)$ is the vector of costs at location (u, v).

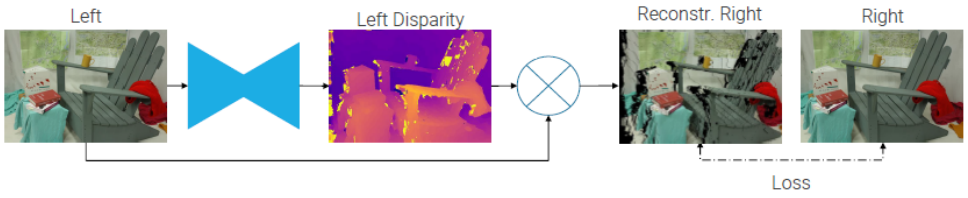
12.2 Depth from Monocular Cues

In principle, it is not possible to recover depth information from a single image: it is a mathematically ill-posed problem. Yet, humans can roughly reconstruct the 3D structure of a scene from a single image based on their experience of the world. Training a machine to mimic this ability of the human visual system is referred to as solving the single-view (or monocular) depth estimation problem. In particular:

- Preliminary attempts used the classical supervised framework. However, this approach requires a large amount of realistic synthetic data and an expensive and cumbersome hardware to provide the depth values for fine-tuning on the real scenario. Thus, it is relatively easy to obtain a pair of synchronized images (left and right images), but providing the ground-truth disparities is expensive.
- If the estimated disparity at pixel (u, v) is $\tilde{d}$ then the pixel in the right image at location $(u+\tilde{d}, v)$ should have the same appearance of pixel (u, v) in the left image. Thus, one can use the estimated disparity and the left image to hallucinate a novel right image and evaluate its quality by its similarity to the real right image:



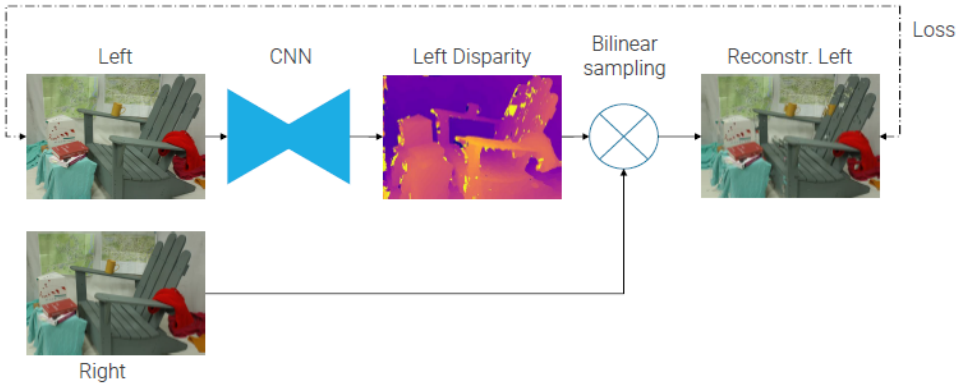
This sampling step should take pixels at locations (u, v) in the left image I^L and move them at location $(u + \tilde{d}^L(u, v), v)$ to create the reconstructed right image $\tilde{I}^R$:



However, there are four main problems with this approach:

  - The forward mapping results in holes in the reconstructed image.
  - Disparities are continuous values.
  - If the right image is reconstructed, the estimated disparity (and depth) will be aligned with the right target image, unavailable at test time. Thus, one can try to reconstruct the left image by sampling from the right image. In this way, the estimated disparity (and depth) will be correctly aligned with the left (input) image.

– When trying to reconstruct the left image, one should define a proper loss function.

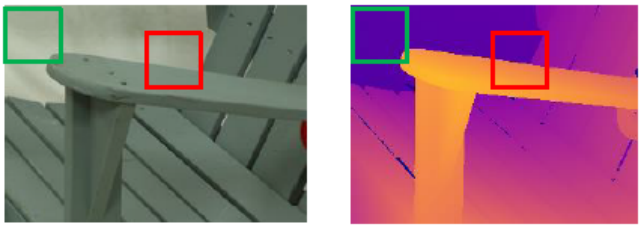


- To compare the original and the reconstructed images, a max of a perceptual distance (i.e. the structural similarity SSIM index), and L_1 norm was found to be the most effective approach:

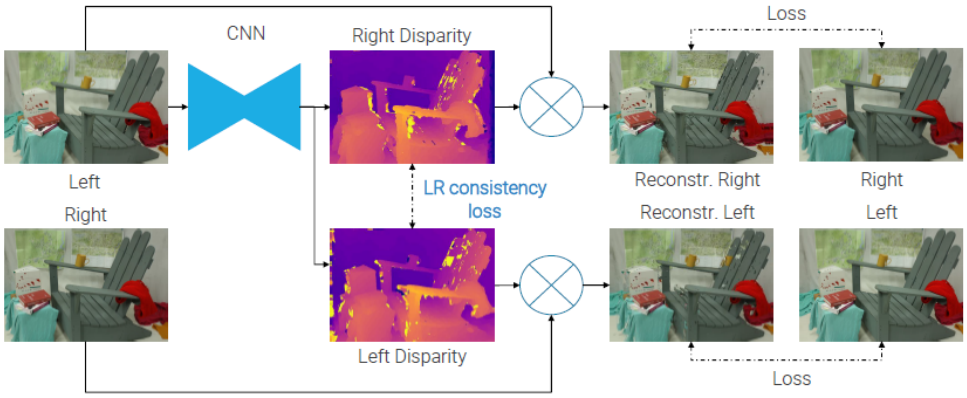
$$L^{ap}(x^{(i,L)}) = \frac{1}{N}\sum_{u,v} \alpha \frac{1 - SSIM(x_{uv}^{(i,L)}, \hat{x}_{uv}^{(i,L)})}{2} + (1-\alpha)\|x_{uv}^{(i,L)} - \hat{x}_{uv}^{(i,L)}\|_1 \tag{75}$$

Another inductive bias (i.e. prior knowledge) one can exploit to improve the results is that disparity maps ten to be locally smooth, but at object boundaries. One can thus add a term to the loss to penalize sudden variations in the disparity map (i.e. vertical and horizontal gradients of the disparity map with large norm) where there are no edges in the input image:

$$L^{ds}(x^{(i,L)}) = \frac{1}{N}\sum_{u,v} |\partial_u d_{uv}^{(i,L)}| e^{-\|\partial_u x_{uv}^{(i,L)}\|_1} + |\partial_v d_{uv}^{(i,L)}| e^{-\|\partial_v x_{uv}^{(i,L)}\|_1} \tag{76}$$



- By letting the network produce both disparity maps from a single image, one can compute the image reconstruction loss (and the smoothness term) for both images.



Moreover, one can enforce that the estimated disparities are consistent:

$$L^{lr}(x^{(i,L)}, x^{(i,R)}) = \frac{1}{N}\sum_{u,v}\left|\tilde{d}^{(i,L)}_{u,v} - \tilde{d}^{(i,R)}_{u+\tilde{d}^{(i,L)}_{u,v},v}\right| + \frac{1}{N}\sum_{u,v}\left|\tilde{d}^{(i,L)}_{u+\tilde{d}^{(i,R)}_{u,v},v} - \tilde{d}^{(i,R)}_{u,v}\right| \tag{77}$$

The total loss for a pair of stereo images then becomes:

$$\begin{aligned} L(x^{(i,L)}, x^{(i,R)}) &= \alpha^{ap}(L^{ap}(x^{(i,L)}) + L^{ap}(x^{(i,R)})) \\ &+ \alpha^{ds}(L^{ds}(x^{(i,L)}) + L^{ds}(x^{(i,R)})) \\ &+ \alpha^{lr}L^{lr}((x^{(i,L)}, x^{(i,R)}) \end{aligned} \tag{78}$$

At test time, only the predicted left disparity map is used to compute the depth for each pixel.