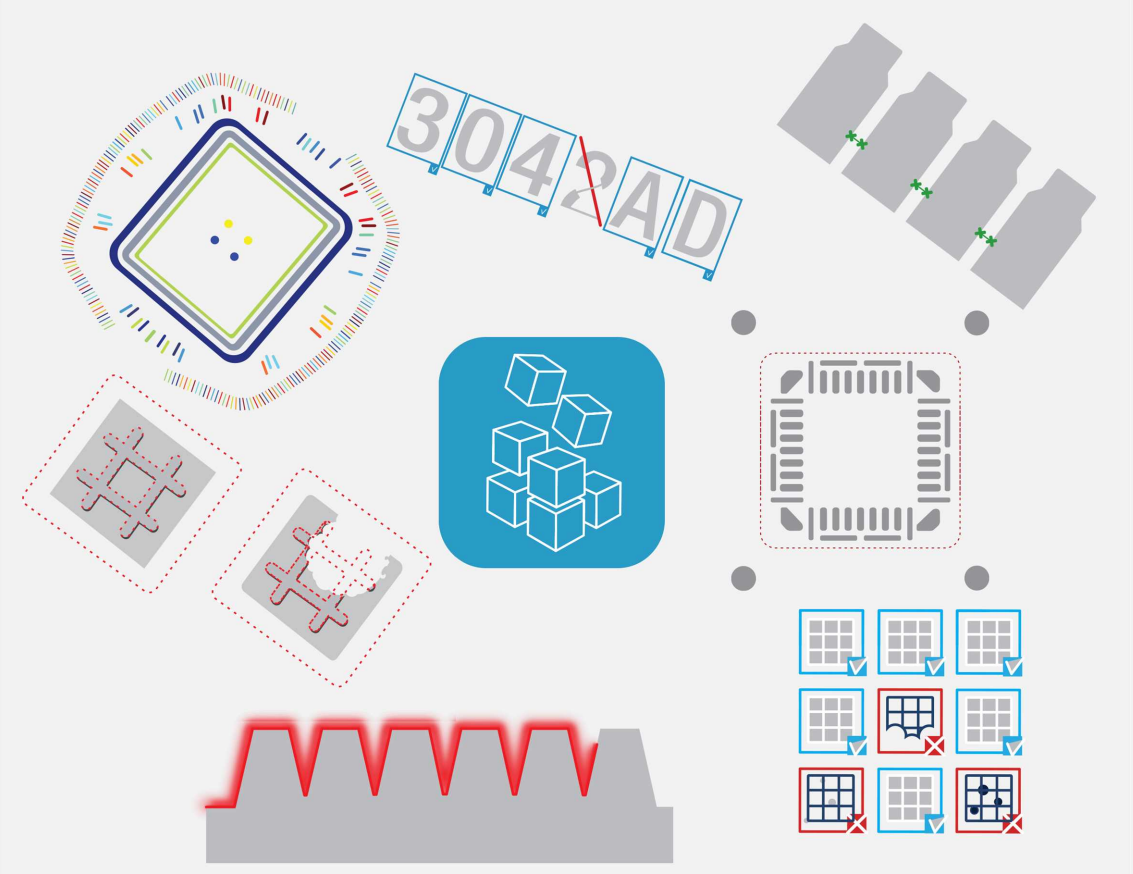
euresys
Empowering Computer Vision

USER GUIDE

Open eVision

Deep Learning Inspection Tools



Terms of Use

EURESYS s.a. shall retain all property rights, title and interest of the documentation of the hardware and the software, and of the trademarks of EURESYS s.a.

All the names of companies and products mentioned in the documentation may be the trademarks of their respective owners.

The licensing, use, leasing, loaning, translation, reproduction, copying or modification of the hardware or the software, brands or documentation of EURESYS s.a. contained in this book, is not allowed without prior notice.

EURESYS s.a. may modify the product specification or change the information given in this documentation at any time, at its discretion, and without prior notice.

EURESYS s.a. shall not be liable for any loss of or damage to revenues, profits, goodwill, data, information systems or other special, incidental, indirect, consequential or punitive damages of any kind arising in connection with the use of the hardware or the software of EURESYS s.a. or resulting of omissions or errors in this documentation.

This documentation is provided with Open eVision 2.11.1 (doc build 1125).

Contents

1. Dealing with Pixel Containers and Files 4
1.1. Pixel Container Definition 4
1.2. Pixel Container Types 6
1.3. Supported Image File Types 7
1.4. Pixel and File Types Compatibility 8
1.5. Color Types 10
2. Manipulating Pixels Containers and Files 11
2.1. Pixel Container File Save 11
2.2. Pixel Container File Load 13
2.3. Memory Allocation 14
2.4. Image and Depth Map Buffer 14
2.5. Image Drawing and Overlay 18
2.6. 3D Rendering of 2D Images 18
2.7. Vector Types and Main Properties 19
2.8. ROI Main Properties 23
2.9. Arbitrarily Shaped ROI (ERegion) 25
2.10. Flexible Masks 31
2.11. Profile 35
3. Deep Learning Tools 37
Deep Learning Tools - Inspecting Images with Deep Learning 38
Purpose and Workflow 38
Tools and Resources 40
Hardware Support (CPU/GPU) 43
Managing the Dataset 46
Images and Labels 46
ROI and Mask 49
Training and Validation Datasets 52
Using Data Augmentation 53
EasyClassify - Classifying Images 56
EasySegment - Detecting and Segmenting Defects 63
4. Code Snippets 69
4.1. Basic Types 70
Loading and Saving Images 70
Interfacing Third-Party Images 70
Retrieving Pixel Values 70
ROI Placement 71
Vector Management 71
Exception Management 72
4.2. Deep Learning Tools 72
Creating a Dataset and Training a Classifier 72
Loading a Classifier and Classifying a New Image 73
Using Multithreading for Classification 73
Loading an Unsupervised Segmenter and Segmenting an Image 74

1. Dealing with Pixel Containers and Files

1.1. Pixel Container Definition

Images

Open eVision image objects contain image data that represents rectangular images.

Each image object has a data buffer, accessible via a pointer, where pixel values are stored contiguously, row by row.

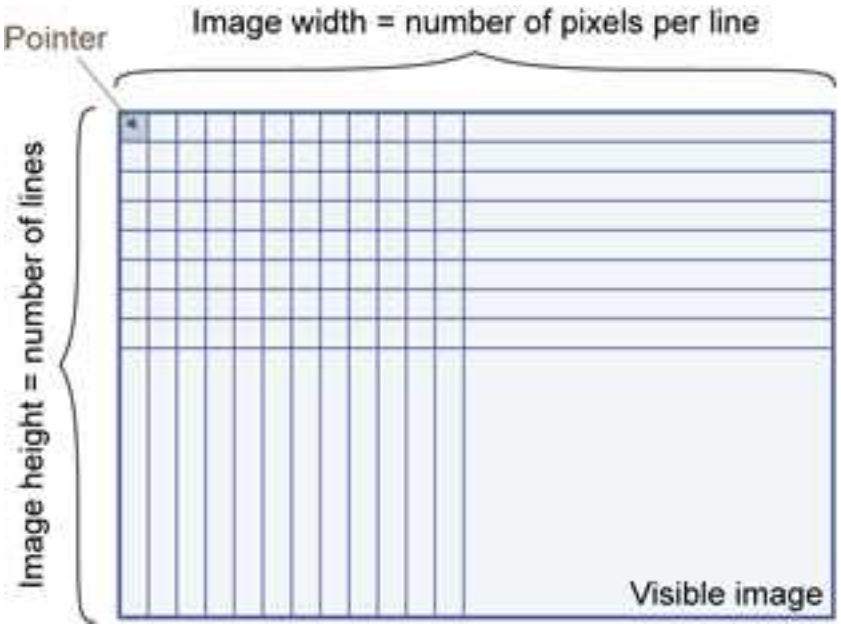


Image main parameters

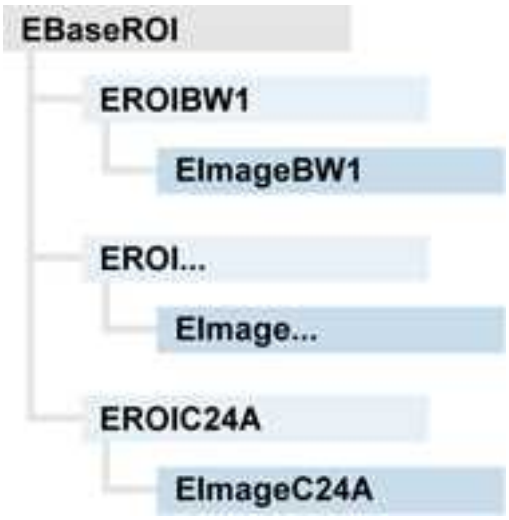
An Open eVision image object has a rectangular array of pixels characterized by `EBaseROI` parameters .

- `Width` is the number of columns (pixels) per row of the image.
- `Height` is the number of rows of the image. (Maximum width / height is 32,767 (2^{15}-1) in Open eVision 32-bit, and 2,147,483,647 (2^{31}-1) in Open eVision 64-bit.)
- `Size` is the width and height.

The `Plane` parameter contains the number of color components. Gray-level images = 1. Color images = 3.

Classes

Image and ROI classes derive from abstract class `EBaseROI` and inherit all its properties.



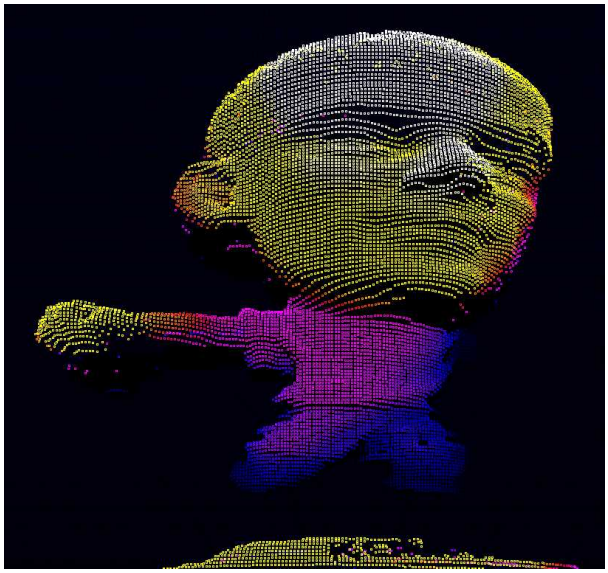
Depth maps

A depth map is way to represent a 3D object using a 2D grayscale image, each pixel in the image representing a 3D point.

The pixel coordinates are the representation of the X and Y coordinates of the point while the grayscale value of the pixel is a representation of the Z coordinate of the point.

Point clouds

A point cloud (https://en.wikipedia.org/wiki/Point_cloud) is an unstructured set of 3D points representing discrete positions on the surface of an object.



3D point clouds are produced by various 3D scanning techniques, such as Laser Triangulation, Time of Flight or Structured Lighting.

1.2. Pixel Container Types

Images

Several image types are supported according to their pixel types: black and white, gray levels, color, etc.

Easy.GetBestMatchingImageType returns the best matching image type for a given file on disk.

BW1	1-bit black and white images (8 pixels are stored in 1 byte)	EImageBW1
BW8	8-bit grayscale images (each pixel is stored in 1 byte)	EImageBW8
BW16	16-bit grayscale images (each pixel is stored in 2 bytes)	EImageBW16
BW32	32-bit grayscale images (each pixel is stored in 4 bytes)	EImageBW32
C15	15-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images and MultiCam RGB15 format.	EImageC15

C16	16-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB16 color images and MultiCam RGB16 format.	`EImageC16`
C24	C24 images store 24-bit color images (each pixel is stored in 3 bytes). Compatible with Microsoft® Windows RGB24 color images and MultiCam RGB24 format.	`EImageC24`
C24A	C24A images store 32-bit color images (each pixel is stored in 4 bytes). Compatible with Microsoft® Windows RGB32 color images and MultiCam RGB32 format.	`EImageC24A`

Depth Maps

8 and 16-bit depth map values are stored in buffers compatible with the 2D Open eVision images.

EDepth8	8-bit depth map (each pixel is stored in 1 byte as an integer)	`EDepthMap8`
EDepth16	16-bit depth map (each pixel is stored in 2 bytes as a fixed point)	`EDepthMap16`
EDepth32f	32-bit depth map (each pixel is stored in 4 bytes as a float)	`EDepthMap32f`

Point Clouds

Point Cloud	Set of points coordinates (stored as float)	`E3DPointCloud`

1.3. Supported Image File Types

Type	Description
BMP	Uncompressed image data format (Windows Bitmap Format)
JPEG	Lossy data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 10918-1. Compression irretrievably looses quality.
JFIF	JPEG File Interchange Format

Type	Description
JPEG-2000	Data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 15444-1 and ISO/IEC 15444-2. Open eVision supports only lossy compression format, file format and code stream variants. - **code stream** describes the image samples. - **file format** includes meta-information such as image resolution and color space.
PNG	Lossless data compression method (Portable Network Graphics).
Serialized	Euresys proprietary image file format obtained from the **serialization** of Open eVision image objects.
TIFF	Tag Image File Format is currently controlled by Adobe Systems and uses the LibTIFF third-party library to process images written for 5.0 or 6.0 TIFF specification. File **save** operations are lossless and use CCITT 1D compression for 1-bit binary pixel types and LZW compression for all others. File **load** operations support all TIFF variants listed in the LibTIFF specification.

1.4. Pixel and File Types Compatibility

Depth map to image conversion

For a 8- and 16-bit depth maps, the `AsImage()` method returns a compatible image object (respectively `EImageBW8` and `EImageBW16`) that can be used with Open eVision's 2D processing features.

Pixel and file types compatibility

Pixel access

The recommended method to access pixels is to use `SetImagePtr` and `GetImagePtr` to embed the image buffer access in your own code. See also Image Construction and Memory Allocation and Retrieving Pixel Values.

Use of the following methods should be limited because of the overhead incurred by each function call:

Direct access

`EROIBW8::GetPixel` and `SetPixel` methods are implemented in all image and ROI classes to read and write a pixel value at given coordinates. To scan all pixels of an image, you could run a double loop on the X and Y coordinates and use `GetPixel` or `SetPixel` each iteration, but this is not recommended.

TIP
For performance reasons, these accessors should not be used when a significant number of pixel needs to be processed. When that is the case, retrieving the internal buffer pointer using `GetBufferPtr()` and iterating on the pointer is recommended.

Quick Access to BW8 Pixels

In BW8 images, a call to `EBW8PixelAccessor::GetPixel` or `SetPixel` will be faster than a direct `EROIBW8::GetPixel` or `SetPixel`.

Supported structures

- `EBW1`, `EBW8`, `EBW32`
- `EC15` (*), `EC16` (*), `EC24` (*)
- `EC24A`
- `EDepth8`, `EDepth16`, `EDepth32f`,

(*) These formats support RGB15 (5-5-5 bit packing), RGB16 (5-6-5 bit packing) and RGB32 (RGB + alpha channel) but they must be converted to/from EC24 using `EasyImage::Convert` before any processing.

NOTE
Transition with versions prior to eVision 6.5 should be seamless: image pixel types were defined using typedef of integral types, pixel values were treated as unsigned numbers and implicit conversion to/from previous types is provided.

Pixel and File Type compatibility during Load or Save operations

Type	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
BW1	Ok	N/A	N/A	Ok	Ok	Ok
BW8	Ok	Ok	Ok	Ok	Ok	Ok
BW16	N/A	N/A	Ok	Ok	Ok (***)	Ok
BW32	N/A	N/A	N/A	N/A	Ok (***)	Ok
C15	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C16	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C24	Ok	Ok	Ok	Ok	Ok (**)	Ok
C24A	Ok	N/A	N/A	Ok	N/A	Ok
Depth8	Ok	Ok	Ok	Ok	Ok	Ok
Depth16	N/A	N/A	Ok	Ok	Ok (***)	Ok
Depth32f	N/A	N/A	N/A	N/A	N/A	Ok

N/A: Not supported. An exception occurs if you use the combination.

Ok: Image integrity is preserved with no data loss (apart from JPEG and JPEG2000, lossy compression).

(**) C15 and C16 formats are automatically converted into C24 during the save operation.

(***) BW16 and BW32 are not supported by Baseline TIFF readers.

1.5. Color Types

EISH: Intensity, Saturation, Hue color system.

ELAB: CIE Lightness, a*, b* color system.

ELCH: Lightness, Chroma, Hue color system.

ELSH: Lightness, Saturation, Hue color system.

ELUV: CIE Lightness, u*, v* color system.

ERGB: NTSC/PAL/SMPTE Red, Green, Blue color system.

EVSH: Value, Saturation, Hue color system.

EXYZ: CIE XYZ color system.

EYIQ: CCIR Luma, Inphase, Quadrature color system.

EYSH: CCIR Luma, Saturation, Hue color system.

EYUV: CCIR Luma, U Chroma, V Chroma color system.

2. Manipulating Pixels Containers and Files

2.1. Pixel Container File Save

Images and Depth Maps

The `Save` method of an image or the `SaveImage` method of a depth map or a ZMap saves the image data of an image or of a depth map or a ZMap object into a file using two arguments:

- Path: path, filename, and file name extension.
- Image File Type. If omitted, the file name extension is used.

Images bigger than 65,536 (either width or height) must be saved in Open eVision proprietary format.

Save throws an exception when:

- The requested image file format is incompatible with the image pixel types
- The Auto file type selection method and the file name extension is not supported

TIP
When saving a 16-bit depth map, the fixed point precision is lost and the pixels are considered as 16-bit integers.

image file type arguments

Argument	Image File Type
EImageFileType_Auto(*)	Automatically determined by the filename extension. See below.
EImageFileType_Euresys	Open eVision Serialization.
EImageFileType_Bmp	Windows bitmap - BMP
EImageFileType_Jpeg	JPEG File Interchange Format - JFIF
EImageFileType_Jpeg2000	JPEG 2000 File format/Code Stream -JPEG2000
EImageFileType_Png	Portable Network Graphics - PNG
EImageFileType_Tiff	Tagged Image File Format - TIFF

(*) Default value.

Assigned image file type if argument is ImageFileType_Auto or missing

File name extension(*)	Automatically assigned image file type
BMP	Windows Bitmap Format
JPEG, JPG	JPEG File Interchange Format - JFIF
JP2	JPEG 2000 file format
J2K, J2C	JPEG 2000 Code Stream
PNG	Portable Network Graphics
TIFF, TIF	Tagged Image File Format

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

`SaveJpeg` and `SaveJpeg2K` specify the compression quality when saving compressed images. They have two arguments:

- Path: a string of characters including the path, filename, and file name extension.
- Compression quality of the image file, an integer value in range [0: 100].
  `SaveJpeg` saves image data using JPEG File Interchange Format – JFIF.
  `SaveJpeg2K` saves image data using JPEG 2000 File format.

JPEG compression values

JPEG compression	Description
JPEG_DEFAULT_QUALITY (-1)	Default quality (*)
100	Superb image quality, lowest compression factor
75	Good image quality (*)
50	Normal image quality
25	Average image quality
10	Bad Image quality

(*) The default quality corresponds to the good image quality (75).

Representative JPEG 2000 compression quality values

JPEG 2000 compression	Description
-1	Default quality (*)
1	Highest image quality, lowest compression factor

JPEG 2000 compression	Description
16	Good Image Quality (*) (16:1 rate)
512	Lowest image quality, highest compression factor

(*) The default quality corresponds to the good image quality (16:1 rate).

Point Clouds

- Use the `Save` method to save the point cloud in Open eVision proprietary file format.
- Use the `SavePCD` method to save the point cloud in a ASCII or a binary file compatible with other software such as PCL (Point Cloud Library).

TIP
The PCD format is supported in ASCII and binary modes.

2.2. Pixel Container File Load

Images and Depth Maps

- Use the `Load` method to load image data into an image object:
  - It has one argument: the **path:** path, filename, and file name extension.
  - File type is determined by the file format.
  - The destination image is automatically resized according to the size of the image on disk.
- The `Load` method throws an exception when:
  - File type identification fails
  - File type is incompatible with pixel type of the image object

TIP
Serialized image files of Open eVision 1.1 and newer are incompatible with serialized image files of previous Open eVision versions.

TIP
When loading a BW16 image (with integer values) in a depth map, the fixed point precision set in the depth map (0 by default) is left unchanged and used.

Point Clouds

- Use the `Load` method to save the point cloud in Open eVision proprietary file format.

- Use the `LoadPCD` method to save the point cloud in a ASCII or a binary file compatible with other software such as PCL (Point Cloud Library).

2.3. Memory Allocation

An image can be constructed with an internal or external memory allocation.

Internal Memory Allocation

The image object dynamically allocates and unallocates a buffer. Memory management is transparent.
When the image size changes, re-allocation occurs.
When an image object is destroyed, the buffer is unallocated.
To declare an image with internal memory allocation:

1. Construct an image object, for instance `EImageBW8`, either with width and height arguments, OR using the `SetSize` function.
2. Access a given pixel. There are several functions that do this. `GetImagePtr` returns a pointer to the first byte of the pixel at given coordinates.

External Memory Allocation

The user controls buffer allocation, or links a third-party image in the memory buffer to an Open eVision image.
Image size and buffer address must be specified.
When an image object is destroyed, the buffer is unaffected.
To declare an image with external memory allocation:

1. Declare an image object, for instance `EImageBW8`.
2. Create a suitably sized and aligned buffer (see Image Buffer).
3. Set the image size with the `SetSize` function.
4. Access the buffer with `GetImagePtr`. See also Retrieving Pixel Values.

2.4. Image and Depth Map Buffer

Image and depth map pixels are stored contiguously, from top row to bottom, from left to right, in Windows bitmap format (top-down DIB[1]) into an associated buffer.

The buffer address is a pointer to the start address of the buffer, which contains the top left pixel of the image.

[1]device-independent bitmap

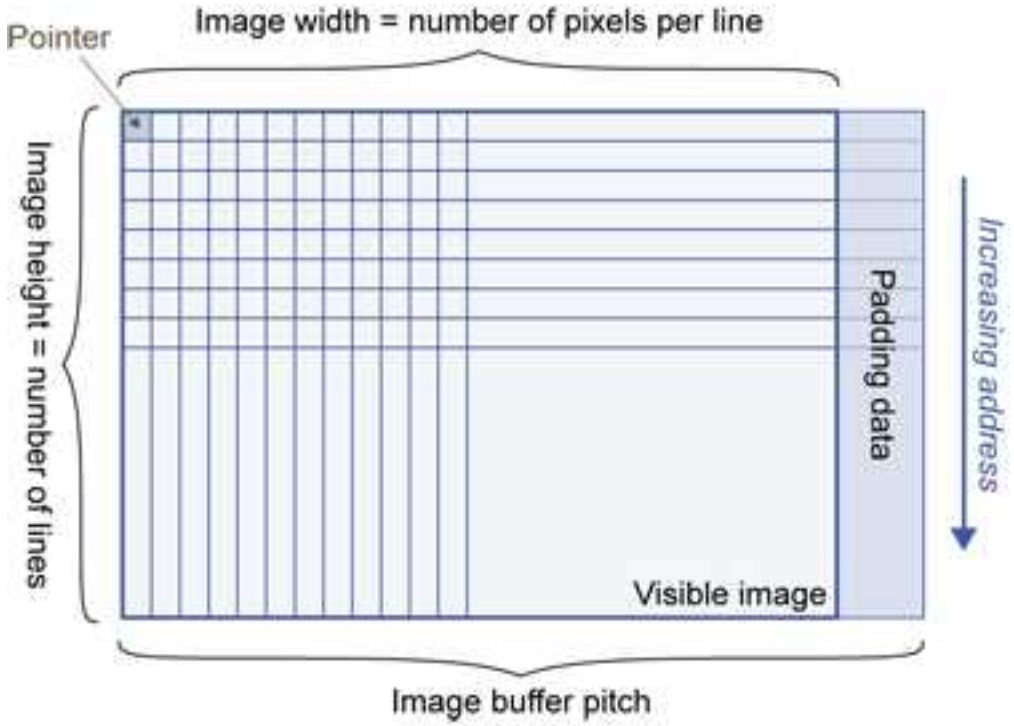


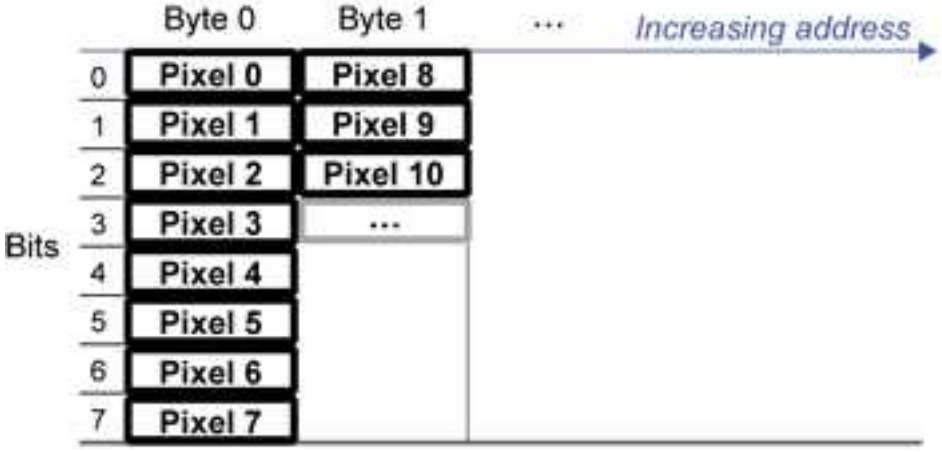
Image Buffer pitch

- Alignment must be a multiple of 4 bytes.
- Open eVision 1.2 onwards default pitch is 32 bytes for performance reasons (Open eVision 1.1.5 was 8 bytes).

Memory Layout

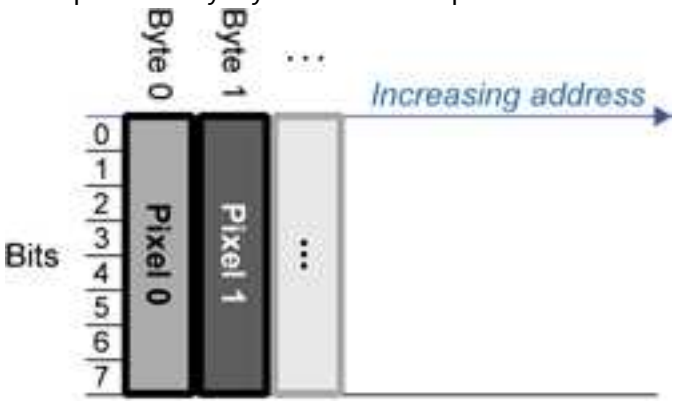
- `EImageBW1` stores 8 pixels in one byte.

  Example memory layout of the first 2 pixels of a BW1 image buffer:



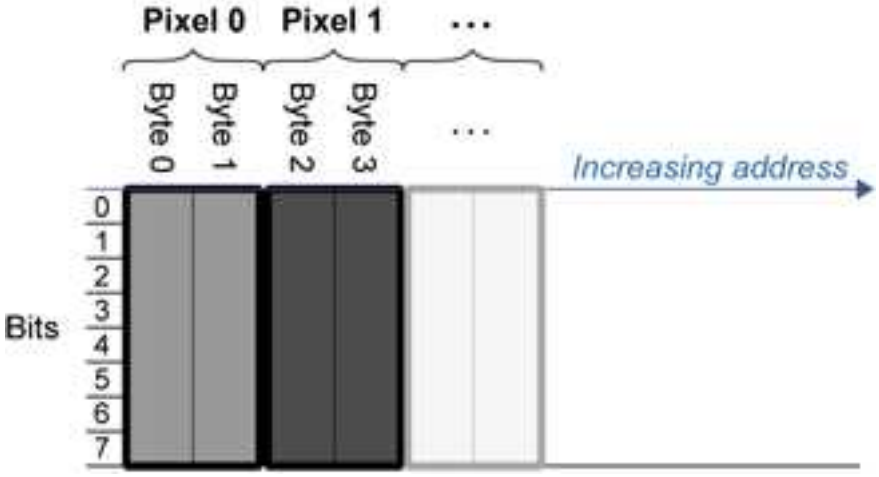
- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

Example memory layout of the first pixels of a BW8 image buffer:



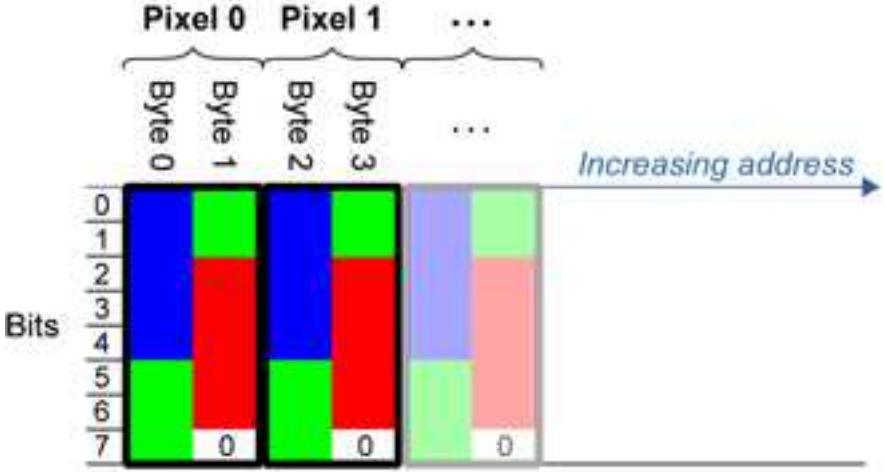
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).

  Example memory layout of the first pixels of a BW16 image buffer:



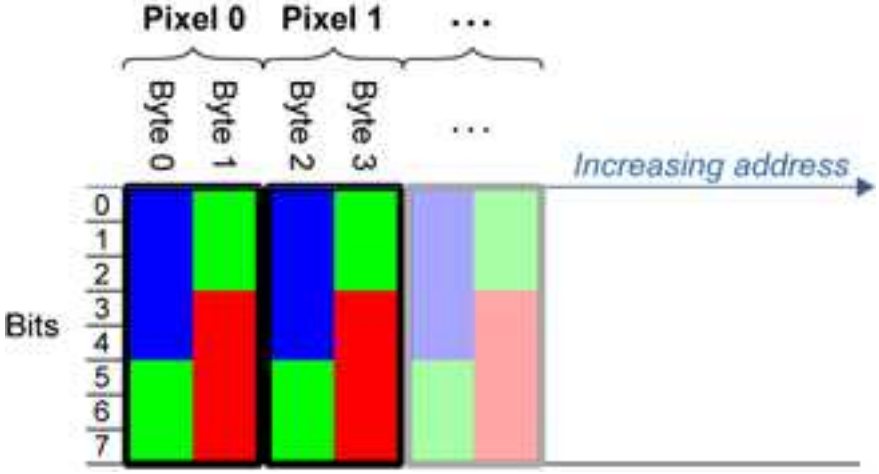
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

  Example memory layout of the first pixels of a C15 image buffer:



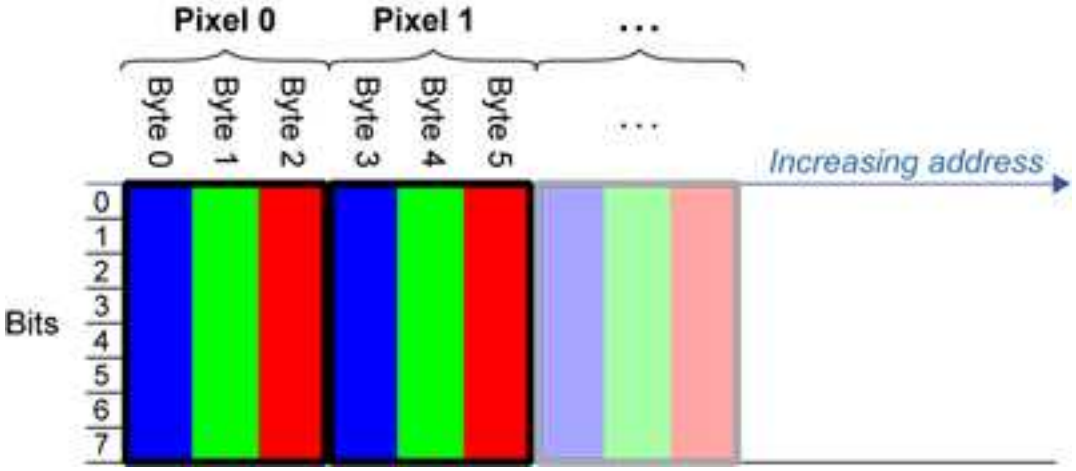
- `EImageC16` stores each pixel in 2 bytes. The first and third color components are coded with 5-bits.
  The second color component is coded with 6-bits.

Example memory layout of the first pixels of a C16 image buffer:



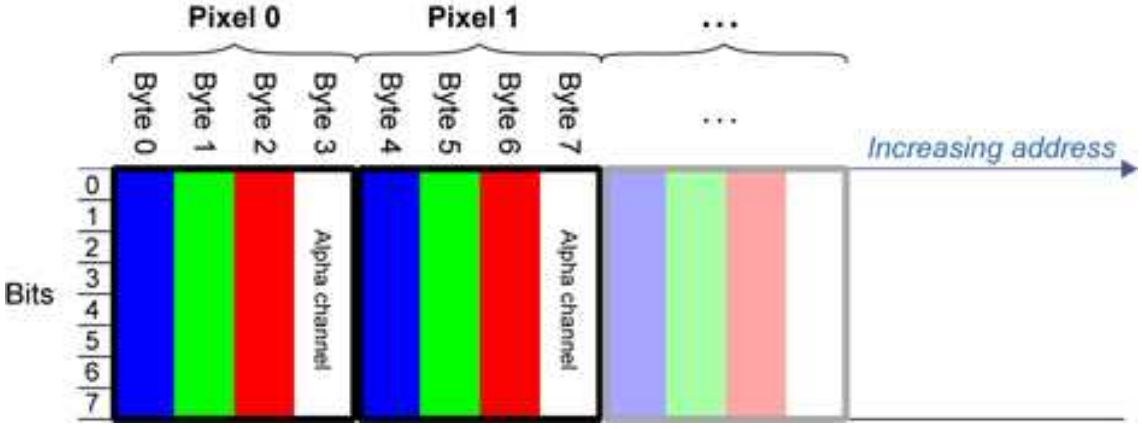
- `EDepthMap16` store each pixel in 2 bytes using a fixed point format.
- `EImageC24` stores each pixel in 3 bytes. Each color component is coded with 8-bits.

  Example memory layout of the first pixels of a C24 image buffer:



- `EImageC24A` stores each pixel in 4 bytes. Each color component is coded with 8-bits. The alpha channel is also coded with 8-bits.

  Example memory layout of the first pixels of a C24A image buffer:



- `EDepthMap32f` store each pixel in 4 bytes using a float format.

2.5. Image Drawing and Overlay

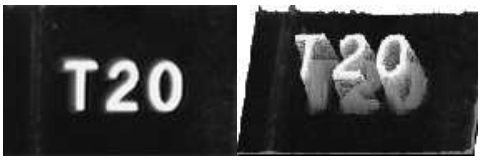
- Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.
- The color palette in 256-color display mode gives optimal rendering. Gray-level images can be improved using LUT[3]s (using histogram stretching techniques or pseudo-coloring).
- The zoom can be different horizontally and vertically.
- `DrawFrameWithCurrentPen` method draws a frame.
- **Non-destructive overlaying** drawing operations do not alter the image contents, such as `MoveTo/LineTo`.
- **Destructive overlaying** drawing operations alter the image contents by drawing inside the image such as `Easy::OpenImageGraphicContext`. Gray-level [color] images can only receive a gray-level [color] overlay.

2.6. 3D Rendering of 2D Images

These images are viewed by rotating them around the X-axis, then the Y-axis.

Gray 3D Rendering

`Easy::Render3D` prepares a 3-dimensional rendering where gray-level values are altitudes. Magnification factors in the three directions (X = width, Y = height and Z = depth) can be given. The rendered image appears as independent dots whose size can be adjusted to make the surface more or less opaque.



3D rendering

Color Histogram 3D Rendering

`Easy::RenderColorHistogram` prepares a 3-dimensional rendering of a color image histogram. The pixels are drawn in the RGB space (not XY-plane) to show clustering and dispersion of RGB

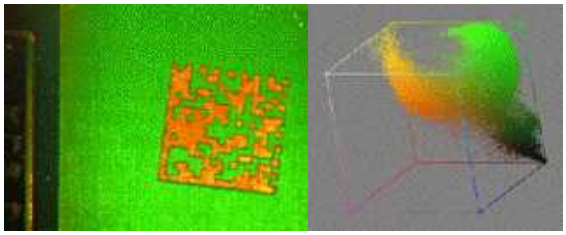
[1]Graphics Device Interface

[2]Microsoft Foundation Class

[3]LookUp Tables

values.
This function can process pixels in other color systems (using EasyColor to convert), but the raw RGB image is required to display the pixels in their usual colors.

Magnification factors in all three directions (X = red, Y = green and Z = blue) can be given.

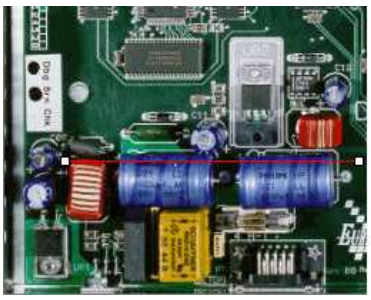


Color histogram rendering

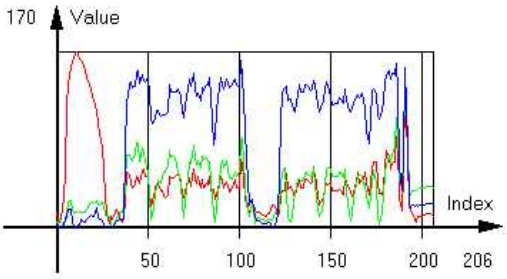
2.7. Vector Types and Main Properties

A vector is a one-dimensional array of pixels (taken from an image profile or contour).

`EVector` is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.



Profile in a C24 image



RGB values plot along profile

Index	Red	Green	Blue
0	15	5	3
1	7	4	0
2	5	8	0
3	9	5	0
4	29	1	0
5	55	6	9
6	120	15	9
7	139	24	17
8	157	26	18
9	161	17	6
10	165	13	0
11	170	14	1

RGB values array (`EC24Vector`)

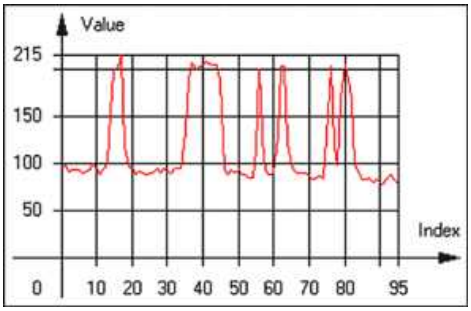
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

1. **Create a vector of the appropriate type**, using its constructor and pre-allocate elements if required.

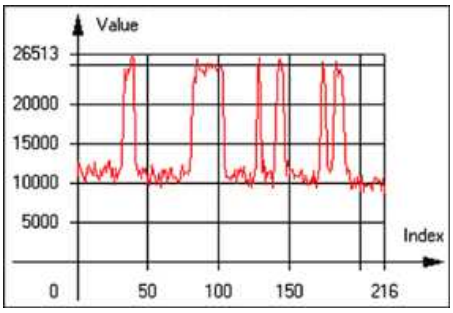
Vector types

- `EBW8Vector`: a sequence of gray-level pixel values, often extracted from an image profile (used by `EasyImage::Lut`, `EasyImage::SetupEqualize`, `EasyImage::ImageToLineSegment`, `EasyImage::LineSegmentToImage`, `EasyImage::ProfileDerivative`, ...).



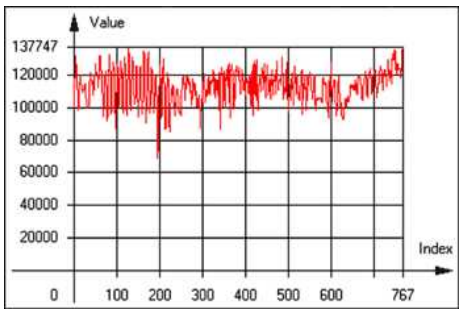
Graphical representation of an `EBW8Vector` (see `Draw` method)

- `EBW16Vector`: a sequence of gray-level pixel values, using an extended range (16 bits), mainly for intermediate computations.



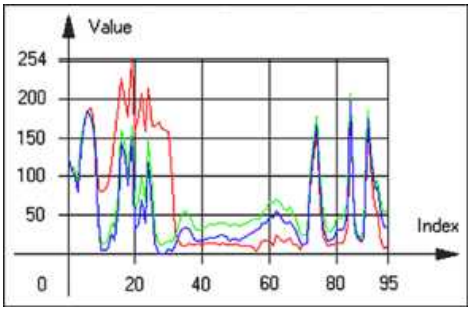
Graphical representation of an `EBW16Vector`

- `EBW32Vector`: a sequence of gray-level pixel values, using an extended range (32 bits), mainly for intermediate computations
  (used in `EasyImage::ProjectOnARow`, `EasyImage::ProjectOnAColumn`, ...).



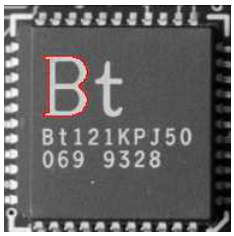
Graphical representation of an `EBW32Vector`

- `EC24Vector`: a sequence of color pixel values, often extracted from an image profile (used by `EasyImage::ImageToLineSegment`, `EasyImage::LineSegmentToImage`, `EasyImage::ProfileDerivative`, ...).



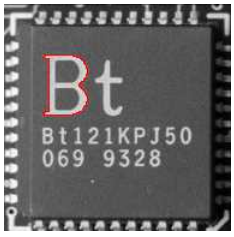
Graphical representation of an `EC24Vector`

- `EBW8PathVector`: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
  (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).



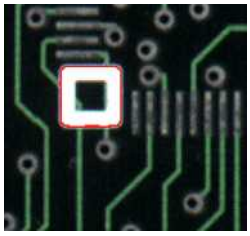
Graphical representation of an `EBW8PathVector` (see `Draw` method)

- `EBW16PathVector`: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
  (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).



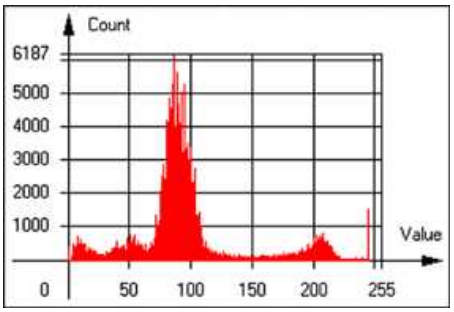
Graphical representation of an `EBW16PathVector` (see `Draw` method)

- `EC24PathVector`: a sequence of color pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
  (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).



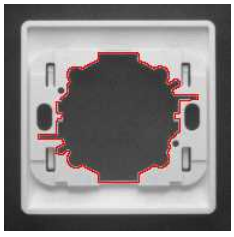
Graphical representation of an `EC24PathVector` (see `Draw` method)

- `EBWHistogramVector`: a sequence of frequency counts of pixels in a BW8 or BW16 image (used by `EasyImage::IsodataThreshold`, `EasyImage::Histogram`, `EasyImage::AnalyseHistogram`, `EasyImage::SetupEqualize`, ...).



Graphical representation of an `EBWHistogramVector` (see `Draw` method)

- `EPathVector`: a sequence of pixel coordinates. The corresponding pixels need not be contiguous
  (used by `EasyImage::PathToImage` and `EasyImage::Contour`).



Graphical representation of an `EPathVector` (see `Draw` method)

- `EPeakVector`: peaks found in an image profile
  (used by `EasyImage::GetProfilePeaks`).
- `EColorVector`: a description of colors
  (used by `EasyColor::ClassAverages` and `EasyColor::ClassVariances`).

2. **Fill a vector with values**. First empty it, using the `EVector::Empty` member, then add elements one at a time by calling the `EC24Vector::AddElement` member. You can access any element by means of indexing.

3. **Access a vector element**, either for reading or writing. Use the brackets operator, for instance, `EC24Vector::operator[]`.
4. **Determine the current number of elements**, use member `EVector::NumElements`.
5. **Draw the vector**.
   A pixel vector is a plot of the element values as a function of the element index, so its graphical appearance depends on its type. You can draw a vector in a window. For legibility, the drawing should appear on a neutral background.
   Drawing is done in the device context associated to the desired window. By default, curves are drawn in blue, annotations are drawn in black. The following parameters can be defined: graphicContext, width, height, origin, origin, color0, color1, color2.
   The `EC24Vector` has three curves drawn instead of one, each corresponding to a color component. By default, red, blue and green pens are used.

2.8. ROI Main Properties

ROIs are defined by a width, a height, and **origin** x **and** y **coordinates**.
The origins are specified with respect to the top left corner in the parent image or ROI.
The ROI must be wholly contained in its parent image.
The processing/analysis time of a BW1 ROI is faster if `OrgX` and `Width` are multiples of 8.

Save and load

You can save or load an ROI as a separate image, to be used as if it was a full image. The ROIs perform **no memory allocation** at all and never duplicate parts of their parent image, the parent image provides them with access to its image data.

The image size of the new file must match the size of the ROI being loaded into it. The image around the ROI remains unchanged.

ROI Classes

An Open eVision ROI inherits parameters from the abstract class `EBaseROI`.

There are several ROI types, according to their pixel type. They have the same characteristics as the corresponding image types.

- `EROIBW1`
- `EROIBW8`
- `EROIBW16`
- `EROIBW32`
- `EROIC15`
- `EROIC16`
- `EROIC24`
- `EROIC24A`

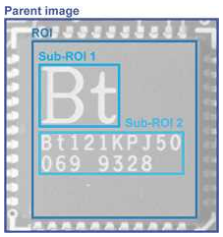
Attachment

An ROI must be `attached` to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.
A normal image cannot be attached to another image or ROI.

Nesting

Set and Get functions change or query the width, height and position of the origin of an ROI, with respect to its immediate or topmost parent image.

An image may accommodate an arbitrary number of ROIs, which can be nested in a hierarchical way. Moving the ROI also moves the embedded ROIs accordingly. The image/ROI classes provide several methods to traverse the hierarchy of ROIs associated with an image.



Nested ROIs: Two sub-ROIs attached to an ROI, itself attached to the parent image

Cropping

`CropToImage` crops an ROI which is partially out of its image. The resized ROI never grows.
An exception is thrown if a function attempts to use an ROI that has limits that extend outside of the parents.

NOTE
(In Open eVision 1.0.1 and earlier, an ROI was silently resized or repositioned when placed out of its image and sometimes grew. If ROI limits extended outside parents, they were silently resized to remain within parent limits.)

Resizing and moving

- ROIs can easily be resized and positioned by two functions and dragging handles:
  - `EBaseROI::Drag` adjusts the ROI coordinates while the cursor moves.
  - `EBaseROI::HitTest` informs if the cursor is placed over a dragging handle. Once the handle is known, the cursor shape can be changed by an `OnSetCursor` MFC event handler. HitTest is unpredictable if called while dragging is in progress.
    HitTest can be used in an OnSetCursor MFC event handler to change the cursor shape, or before a dragging operation like OnLButtonDown,
    (or EvSetCursor and EvLButtonDown in Borland/CodeGear's OWL)
    (or FormMouseMove and FormMouseDown in Borland/CodeGear's VCL).
    In VB6, MouseDown, MouseMove,MouseUp events return the current cursor position in twips rather than pixels, so conversion is mandatory.

2.9. Arbitrarily Shaped ROI (ERegion)

See also:example: Inspecting Pads Using Regions / code snippets: ERegion

Regions or arbitrarily shaped ROI

You define and use regions of interest (ROI) to restrict the area processed with your vision tool and to reduce and optimize the processing time.

In Open eVision:

- An **ROI** (`EROIxxx` class) designates a rectangular region of interest.
- A **region** (`ERegion` class) designates an arbitrarily shaped ROI. With regions, you can determine precisely which part of the image, down to a single pixel, is used for your processing.

Currently, only the following Open eVision methods support `ERegions`:

Library	Method
EasyImage	`EasyImage::Threshold`
	`EasyImage::DoubleThreshold`
	`EasyImage::Histogram`
	`EasyImage::Area`
	`EasyImage::AreaDoubleThreshold`
	`EasyImage::BinaryMoments`
	`EasyImage::WeightedMoments`
	`EasyImage::GravityCenter`
	`EasyImage::PixelCount`
	`EasyImage::PixelMax`
	`EasyImage::PixelMin`
	`EasyImage::PixelAverage`
	`EasyImage::PixelStat`
	`EasyImage::PixelVariance`
	`EasyImage::PixelStdDev`
	`EasyImage::PixelCompare`
Easy3D	`EDepthMapToMeshConverter::Convert`
	`EDepthMapToPointCloudConverter::Convert`
	`EStatistics::ComputePixelStatistics`
	`EStatistics::ComputeStatistics`
	`E3DObjectExtractor::Extract`
	`EZMapToPointCloudConverter::Convert`
EasyObject	`EImageEncoder::Encode`
EasyFind	`EPatternFinder::Find`
	`EPatternFinder::Learn`
EasyOCR2	`EOCR2::Read`
	`EOCR2::Detect`

Library	Method
EasyGauge	`EPointGauge::Measure`
	`ELineGauge::Measure`
	`ERectangleGauge::Measure`
	`ECircleGauge::Measure`
	`EWedgeGauge::Measure`

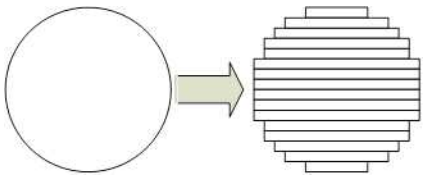
TIP
In the future Open eVision releases, the support of `ERegions` will be gradually extended to all operators.

Creating regions

Open eVision offers multiple ways to create regions, depending on the shape you need:

The `ERegion` is the base class for all regions and the most versatile. It encodes a region using a Run-Length Encoded (RLE) representation.

- The RLE representation of a region is made of runs (horizontal, 1-pixel high slices).
- The runs are stored in the form of their ordinate, starting abscissa and length.



Run-Length Encoding of a circle-shaped region

To create a region, either:

- Use one of the geometry-based region classes.
- Use the result of another tool, such as EasyFind, EasyMatch or EasyObject.
- Combine or modify other regions.
- Use a mask image.
- Directly provide the list of runs.

Geometry-based regions

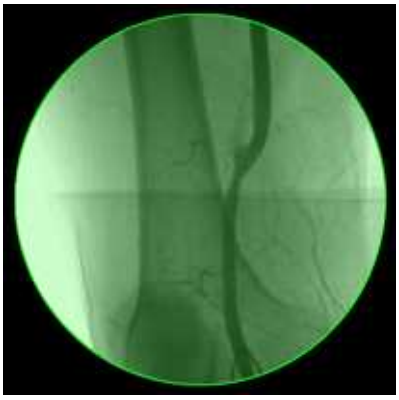
Geometry based regions are specialized classes of regions that are encompassed in simple geometries. Open eVision currently provides classes based on a rectangle, a circle, an ellipse or a polygon.

Use these classes to setup geometric regions and modify them with translation, rotation and scaling. The transformation operators return new regions, leaving the source object unchanged.

- `ERectangleRegion`
  - The contour of an `ERectangleRegion` class is a rectangle.
  - Define it using its center, width, height and angle.
  - Alternatively, use an `ERectangle` instance, such as one returned by an `ERectangleGauge` instance.

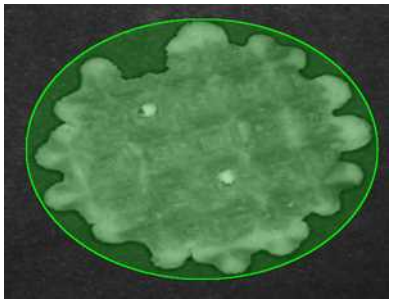
Rectangle region separating a bar code from the background

- `ECircleRegion`
  - The contour of an `ECircleRegion` class is a circle.
  - Define it using its center and radius or 3 non-aligned points.
  - Alternatively, use an `ECircle` instance, such as one returned by an `ECircleGauge` instance.



Circle region encompassing the useful part of an X-Ray image

- `EEllipseRegion`
  - The contour of an `EEllipseRegion` class is an ellipse.
  - Define it using its center, long and short radius and angle.



Ellipse region encompassing a waffle

- `EPolygonRegion`
  - The contour of an `EPolygonRegion` class is a polygon.
  - It is constructed using the list of its vertices.

Polygon region encompassing a key

Using the result of other tools

The `ERegion` class provides a set of specialized constructors to create regions from the results of another tool.

In a tool chain, these constructors restrict the processing of a tool to the area issued from the previous tool.

Open eVision provides constructors for the following tools:

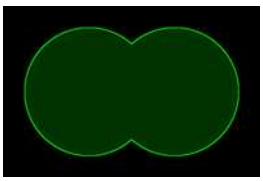
- EasyFind: `EFoundPattern`
- EasyMatch: `EMatchPosition`
- EasyGauge: `ECircle` and `ERectangle`
- EasyObject: `ECodedElement`

> **TIP**
> When compatible, Open eVision also provides specialized constructors for the geometry-based regions. For instance, `ECircleRegion` provides a constructor using an `ECircle`.

Combining regions

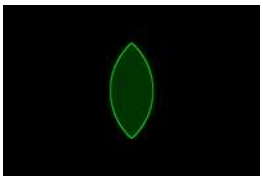
Use the following operations to create a new region by combining existing regions:

- Union
  - The `ERegion::Union(const ERegion&, const ERegion&)` method returns the region that is the addition of the two regions passed as arguments.



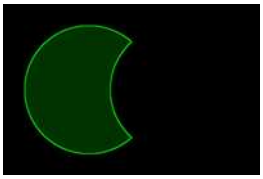
Union of 2 circles

- Intersection
  - The `ERegion::Intersection(const ERegion&, const ERegion&)` method returns the region that is the intersection of the two regions passed as argument.



Intersection of 2 circles

- Subtraction
  - The `ERegion::Substraction(const ERegion&, const ERegion&)` method returns the first region passed as argument after removing the second one.



Subtraction of 2 circles

Using regions

The tools supporting regions provide methods that follow one of these conventions:

- `Method(const EImage& source, const ERegion& region)`
- `Method(const EImage& source, const ERegion& region, EImage& destination)`

> **NOTE**
> The source, the region and the destination must be compatible. It means that the region must at least partly fit in the source, and that source and destination must have the same size.

Preparing the region

- Open eVision automatically prepares the regions when it applies them to an image, but this preparation can take some time.
- If you do not want that your first call to a method takes longer than the next ones, you can prepare the region in advance by using the appropriate `Prepare()` method.
- To manually prepare the regions, adapt the internal RLE description to your images.

Drawing regions

The `ERegion` classes provide several ways to display the regions:

- `ERegion::Draw()` draws the region area, in a semi-transparent way, in the provided device context.
- `ERegion::DrawContour()` draws the region contour in the provided device context.

- `ERegion::ToImage()` renders the region as a mask into the provided destination image.
  - You can configure the foreground and the background colors.
  - If you initialized your image with a width and a height, Open eVision renders the region inside those bounds.
  - If not, Open eVision resizes the image to contain the whole region.
  - Use `ToImage()` to create masks for the Open eVision functions that support them.

ERegions and EROIs

- The older `EROI` classes of Open eVision are compatible with the new regions.
- Some tools allow the usage of regions with source and/or destinations that are `ERoi` instead of `EImage` follow one of these conventions:
  - `Method(const ERoi& source, const ERegion& region)`
  - `Method(const ERoi& source, const ERegion& region, ERoi& destination)`

> **TIP**
> In that case, the coordinates used for the region are relative to the reduced ROI space instead of the whole image space .

ERegion and 3D

- The new regions are compatible with the 2.5D representations of Easy3D (`EDepthMap` and `EZMap`).
- You can also reduce the domain of processing when using these classes.

2.10. Flexible Masks

ROIs vs flexible masks

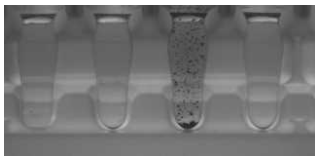
ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 23 apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
- Flexible Masks are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some EasyObject and EasyImage library functions.

Flexible Masks

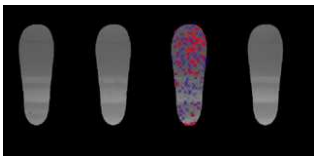
A flexible mask is a BW8 image with the same height and width as the source image. It contains shapes of areas that must be processed and ignored areas (that will not be considered during processing):

- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image

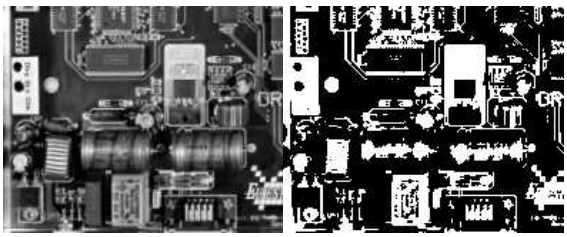
Associated mask



Processed masked image

A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.

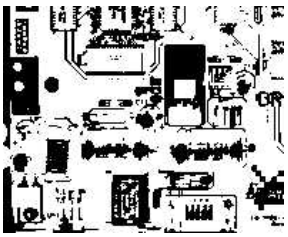
Flexible Masks in EasyImage



Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

1. **Call the functions from EasyImage that take an input mask as an argument**. For instance, one can evaluate the average value of the pixels in the white layer and after in the black layer.
2. **Display the results.**



Resulting image

EasyImage Functions that support flexible masks

- `EImageEncoder::Encode` has a flexible mask argument for BW1, BW8, BW16, and C24 source images.
- AutoThreshold.
- Histogram (function `HistogramThreshold` has no overload with mask argument).
- RmsNoise, SignalNoiseRatio.
- Overlay (no overload with mask argument for BW8 source images).
- ProjectOnAColumn, ProjectOnARow (Vector projection).

- ImageToLineSegment, ImageToPath (Vector profile).

Flexible Masks in EasyObject

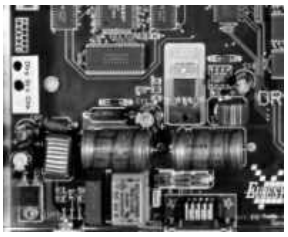
A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

If an object of interest has the same gray level as other regions of the image, you can define "keep" and "ignore" areas using flexible masks and `Encode` functions.

A flexible mask is a BW8 image with the same height and width as the source image.

- A pixel value of 0 in the flexible mask masks the corresponding source image pixel so it doesn't appear in the encoded image.
- Any other pixel value in the flexible mask causes the pixel to be encoded.

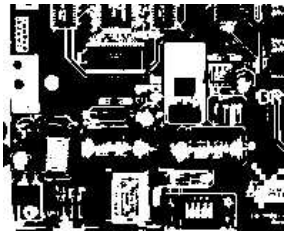
EasyObject functions that create flexible masks



Source image

1) ECodedImage2::RenderMask: from a layer of an encoded image

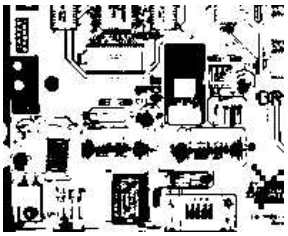
1. To encode and extract a flexible mask, first construct a coded image from the source image.
2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).
3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers using minimum residue thresholding).
4. Make the mask image the desired size using `mask.SetSize(sourceImage.GetWidth(), sourceImage.GetHeight())`.
5. Exploit the flexible mask as an argument to `ECodedImage2::RenderMask`.



BW8 resulting image that can be used as a flexible mask

2) ECodedElement::RenderMask: from a blob or hole

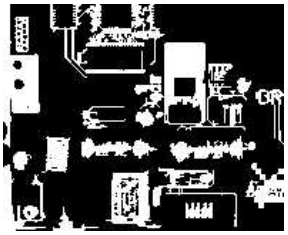
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

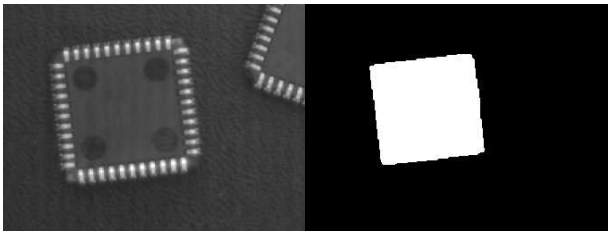
3) EObjectSelection::RenderMask: from a selection of blobs

`EObjectSelection::RenderMask` can, for example, discard small objects resulting from noise.



BW8 resulting image that can be used as a flexible mask

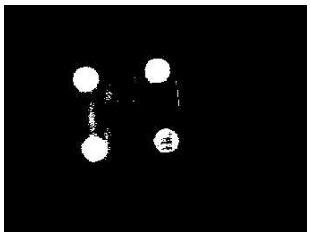
Example: Restrict the areas encoded by EasyObject



Find four circles (left) Flexible mask can isolate the central chip (right)

1. Declare a new `ECodedImage2` object.
2. Setup variables: first declare source image and flexible mask, then load them.
3. Declare an `EImageEncoder` object and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.

4. Encode the source image. Encoding a layer with just the area in the flexible mask is then pretty straightforward.
   We see that the circles are correctly segmented in the black layer with the grayscale single threshold segmenter:



5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

2.11. Profile

Profile Sampling

A **profile** is a series of pixel values sampled along a line/path/contour in an image.

- `EasyImage::ImageToLineSegment` copies the pixel values along a given line segment (arbitrarily oriented and wholly contained within the image) to a vector. The vector length is adjusted automatically. This function supports flexible mask.
- A path is a series of pixel coordinates stored in a vector.
  `EasyImage::ImageToPath` copies the corresponding pixel values to the vector. This function supports flexible mask.
- A **contour** is a closed or not (connected) path, forming the boundary of an object.
  `EasyImage::Contour` follows the contour of an object, and stores its constituent pixels values inside a profile vector.

Profile Analysis

The profile can be processed to find peaks or transitions:

- A transition corresponds to an object edge (black to white or white to black). It can be detected by taking the first **derivative** of the signal (which transforms transitions (edges) into peaks) and looking for peaks in it.
  `EasyImage::ProfileDerivative` computes the first derivative of a profile extracted from a gray-level image.
  The `EBW8` data type only handles unsigned values, so the derivative is shifted up by 128. Values under [above] 128 correspond to negative [positive] derivative (decreasing [increasing] slope).

- A peak is the portion of the signal that is above [or below] a given threshold - the maximum or minimum of the signal. This may correspond to the crossing of a white or black line or thin feature. It is defined by its:
  - Amplitude: difference between the threshold value and the max [or min] signal value.
  - Area: surface between the signal curve and the horizontal line at the given threshold.

  `EasyImage::GetProfilePeaks` detects max and min peaks in a gray-level profile. To eliminate false peaks due to noise, two selection criteria are used. The result is stored in a peaks vector.

Profile Insertion Into an Image

`EasyImage::LineSegmentToImage` copies the pixel values from a vector or constant to the pixels of a given line segment (arbitrarily oriented and wholly contained within the image).

`EasyImage::PathToImage` copies the pixel values from a vector or a constant to the pixels of a given path.

3. Deep Learning Tools

Deep Learning Tools - Inspecting Images with Deep Learning

Purpose and Workflow

Tools

The deep learning tools are based on deep convolutional neural networks (CNNs):

- **EasyClassify** classifies images into a predefined set of classes. Use this tool to identify a product in an image or to detect if the product is good or defective.
- **EasySegment** detects and segments defects in images. This tool works in an unsupervised way. This means that it is trained on good products only.

  As you build only a model of what a good product is and not a model of what a defective product is:

  - The advantage is that the tool can detect and segment defects that are not in your dataset or that are unexpected.
  - The drawback is that the type of defects that the tool can detect and segment is more limited than when you build an explicit model of the defects.

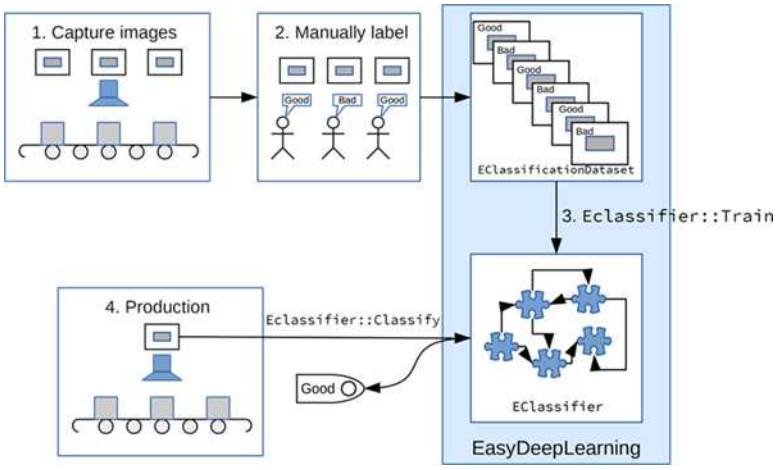
By opposition to traditional machine vision techniques, the deep learning tools do not require an explicit model of what to recognize and/or segment inside an image. Instead, they learn this model from a set of example images. Thus the deep learning tools can solve machine vision problems where an explicit model is too complex to build.

Specifications

	EasyClassify	EasySegment (unsupervised mode)
Minimum image size	128 × 128	64 × 64
Maximum image size	1024 × 1024	10 000 × 10 000
Best image size	256 × 256 - 600 × 600	n.a.
Number of channels	1 or 3 (grayscale and color images)	
Bit depth	8 bits, 16 bits	
Number of labels	2 - 1000	2 (good and defective)
Minimum number of images per label	2	1 for the good label 0 for the defective label
Supported image format	bmp, png, jpeg, j2k, tiff	

TIP
To accelerate computations, we strongly recommend running the deep learning tools on a recent NVIDIA GPU. Refer to the section "Hardware Support (CPU/GPU)" on page 43 for installing the required NVIDIA CUDA and deep learning library.

Workflow



To create an application based on the deep learning tools:

1. Capture a dataset of images representative of the problem you want to solve.
   - The capture conditions must be as close as possible of the production conditions.
   - Preferably, all images should have the same resolution.
   - The number of images needed to obtain a good performance depends on the complexity of the task and the tool used.
     Please refer to the specifications of each tool for the constraints on the resolution and the number of images.
2. Manually label the images in the dataset with the different categories you want to recognize.

   These categories depend on the tool:
   - **EasyClassify**:
     - Each image must correspond to one and only one category.
     - There must be at least 2 categories.
   - **EasySegment**:
     - A single category for images of good samples.
     - As many categories as you want (including none) for images of defective samples.

   Use the `EClassificationDataset` class to compile your labeled images.
3. Choose the deep learning tool that suits your need.

   All deep learning tools are child classes of the `EDeepLearningTool` class:
   - **EasyClassify**: `EClassifier` class

- **EasySegment**: `EUnsupervisedSegmenter` class

4. Train the deep learning tool on the dataset.
5. Apply the trained tool in production.

Each tool returns a specific object.

Tools and Resources

Deep Learning Studio

Deep Learning Studio is a graphical user interface that:

- Creates datasets and labeling images,
- Configures and visualizes the data augmentation transformations,
- Trains a deep learning tool,
- Analyzes the performance of the tool,
- Applies the tool to new images.

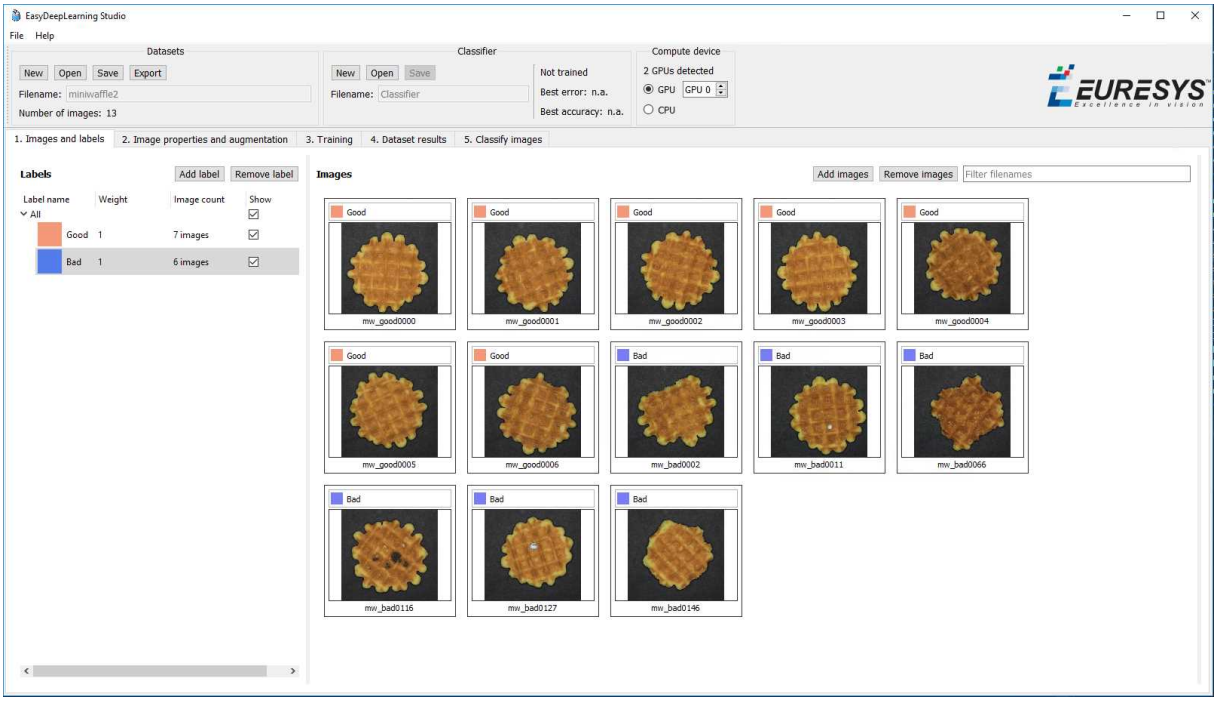
TIP
The **Deep Learning Studio** is available in the installation folder of Open eVision.

Resources and code snippets

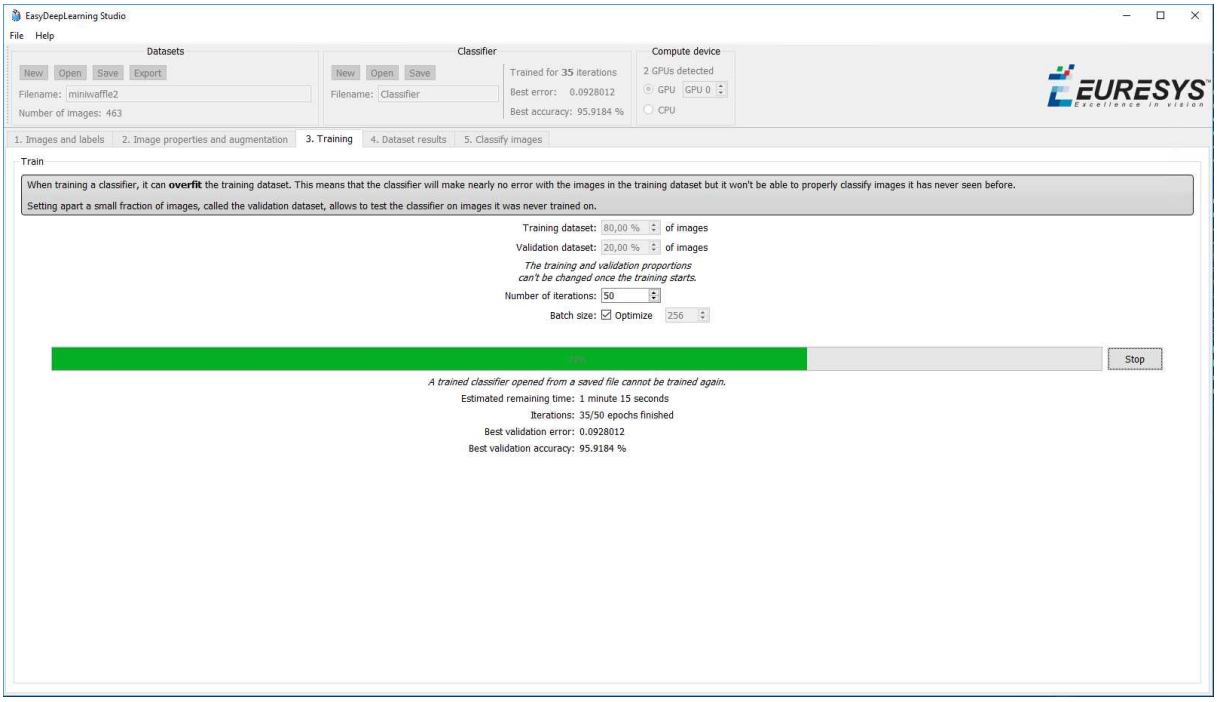
- The sample dataset called “MiniWaffle” illustrated below is available in the folder `Sample Images/Deep Learning/EasyClassify/MiniWaffle` of the installation folder. This dataset contains images of good and bad mini waffles.
- Some sample programs in the folder `Sample Programs` show how to train and use a deep learning tool.
- Some code snippets are also provided for illustration and reference.

Workflow illustration with Deep Learning Studio

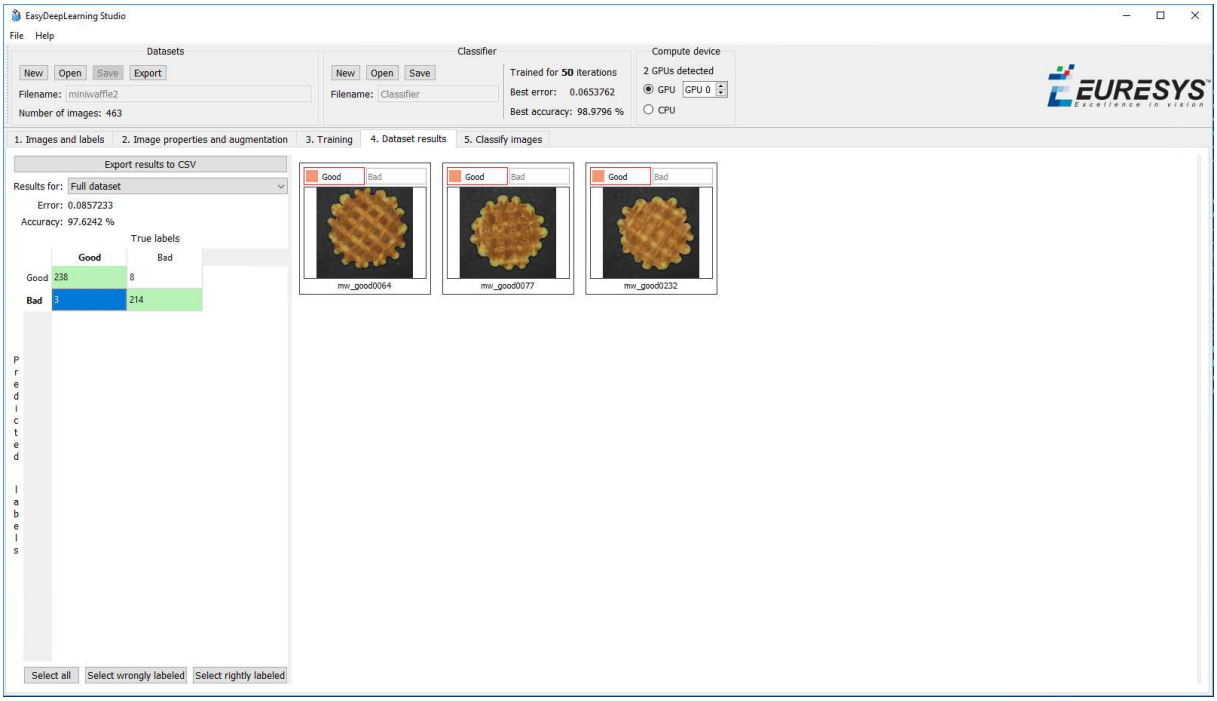
You can use **Deep Learning Studio** to perform steps 2 and 3 of the process described in section "Purpose and Workflow" on page 38.



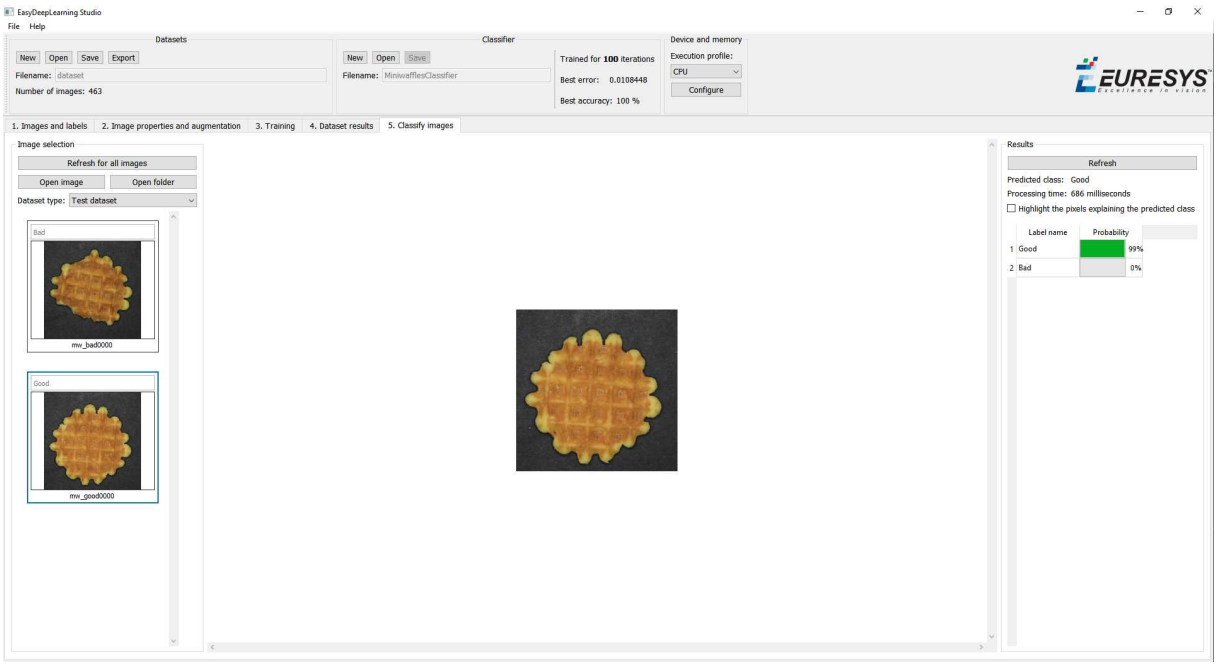
Manual labeling of images (step 2) and creating the dataset (steps 3a and 3b)



Splitting the dataset (step 3c) and starting the training (steps 3d and 3e)



Analyzing the performance (step 3f)



Classifying images (step 4)

Hardware Support (CPU/GPU)

Using a CPU

- Deep learning algorithms perform a lot of computations and can be very slow to train on a CPU.

  For example, for **EasyClassify**, on a high-end Intel Core i9-7900X CPU with a single thread, with no data augmentation:

  - The training can process up to 0.5 MegaPixels/second.
  - The validation and classification can process up to 1.5 MegaPixels/second.

- Use the `EDeepLearningTool::SetEnableGPU(false)` method to use the CPU with the deep learning tools.

> **TIP**
> The deep learning tools support CPU processing for both 32-bit and 64-bit applications. However, the memory of a 32-bit application is limited to 2 GB and this can slow the training or the classification of large images.

Using an NVIDIA CUDA® GPU

- Using a recent NVIDIA GPU greatly accelerates the processing speeds.

  For **EasyClassify**, on a NVIDIA GeForce 1080Ti, with no data augmentation:

  - The training can process up to 50 MegaPixels/second.
  - The validation can process up to 160 MegaPixels/second.
  - The classification of a single image can process up to 55 MegaPixels/second (equivalent to more than 800 256 x 256 grayscale images/second).

> **TIP**
> Please be aware that the actual speed varies with the input image format, the data augmentation, the batch size and the GPU model.

1. To use an NVIDIA GPU with the deep learning tools, install the following NVIDIA libraries on your computer:
   - NVIDIA CUDA® Toolkit version v10.0 (https://developer.nvidia.com/cuda-toolkit)
   - NVIDIA CUDA® Deep Neural Network library (cuDNN) v7 (https://developer.nvidia.com/cudnn)

2. According to the installation location:
   - If you install the NVIDIA CUDA® Toolkit in its default location (`C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0`), a deep learning tool automatically finds what it needs.
   - Otherwise, copy the DLLs `cusolver64_100.dll`, `curand64_100.dll`, `cufft64_100.dll` and `cublas64_100.dll` in the Open eVision DLL folder (its default location is `C:\Program Files (x86)\Euresys\Open eVision X.X\Bin64\`).
3. Install the NVIDIA CUDA ® Deep Neural Network library (cuDNN) that comes as a zip archive:
   a. Unzip the files.
   b. Copy the unzipped files to the NVIDIA CUDA® Toolkit installation directory as indicated in https://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html#installwindows.
   c. If the NVIDIA CUDA® Toolkit is not installed in its default location, copy the DLL file `cudnn64_7.dll` in the Open eVision DLL folder (its default location is `C:\Program Files (x86)\Euresys\Open eVision X.X\Bin64\`).
4. Use the method `EDeepLearningTool::SetEnableGPU(true)` to use the GPU with the deep learning tools.

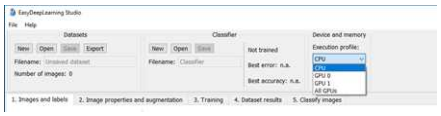
Using multiple GPUs

You can use multiple GPUs for the training and the batch classification.

- In the API, to set the list of GPUs, use the `EDeepLearningTool::SetGPUIndexes` method.

> **NOTE**
> Using multiple GPUs increases the training and batch classification speed only if these GPUs are Quadro or Tesla models with the TCC driver model (see https://docs.nvidia.com/gameworks/content/developertools/desktop/nsight/tesla_compute_cluster.htm).
> Using multiple GeForce GPUs is slower than using a single one. If there are more than one GPU installed on your computer, set the index of the GPU to use with the `EDeepLearningTool::SetGPUIndexes` method.

- In **Deep Learning Studio**, to choose the processing devices, select an execution profile.



- You can configure these execution profiles to match your needs.
- GPU processing is not possible with 32-bit applications.

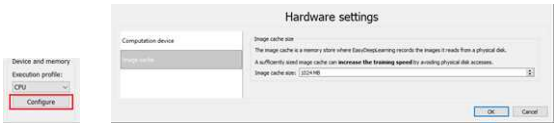
Image cache

The image cache is the part of the memory reserved for storing images during training.

- The default size is 1 GB.
- With large dataset, increasing the image cache size may improve the training speed.

To specify the cache size in bytes:

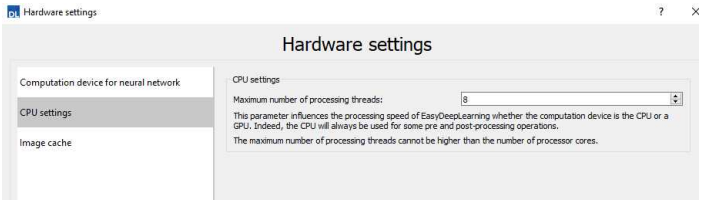
- In the API, use the `EDeepLearningTool::SetImageCacheSize` method.
- In **Deep Learning Studio**, click on the Configure button below the Execution profile control and select Image cache in the menu.



Multicore processing

The deep learning tools support multicore processing (see "Multicore Processing" on page 1):

- In the API, use the multicore processing helper function from Open eVision (that is `Easy::SetMaxNumberOfProcessingThreads()` with a value greater than 1).
- In **Deep Learning Studio**, click on the Configure button below the Execution profile control and select CPU Settings in the menu.



Managing the Dataset

Images and Labels

- In the API, a dataset is represented by an object of the `EClassificationDataset` type.
- The supported image file types are:
  - PNG
  - TIFF
  - JPEG
  - BMP
  - J2K
- The supported Open eVision image object types are:
  - `EImageType_BW8`
  - `EImageType_BW16`
  - `EImageType_C24`

Adding images

- In **Deep Learning Studio**, add image files (PNG, TIFF, JPEG, BMP and J2K types) to your datasets in one of the following ways:
  - Right-click on a label and click Add images to label.

  - Drag and drop your files directly on a label.

  - Select a label and click on the Add Images button.

- Add a single image and its label to a `EClassificationDataset`, in one of the following ways:
  - `EClassificationDataset::AddImage(path, label)` for an image file,
  - `EClassificationDataset::AddImage(img, label)` for an **Open eVision** image object.
- Add several image files that share the same label, with the method `EClassificationDataset::AddImages(filter, label)`.

  `filter` is a glob pattern with the wildcard characters:
  - `*` means "zero or more characters"
  - `?` means "a single character"

  For example, `EClassificationDataset::AddImages("*good*.png", "good")` adds all PNG image files that contain "`good`" in their filename.

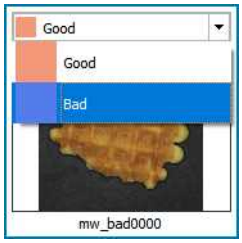
TIP
EasyClassify automatically generates the set of labels from the labels of the images that you add to the dataset.

NOTE
The labels are case sensitive.

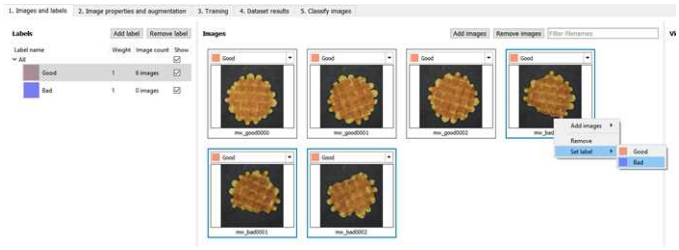
Changing the labels

In **Deep Learning Studio**:

- To change the label of a single image, select or type the label directly on the image thumbnail.



- To change the label of a group of images:
  a. Select the images (keep the CTRL key pressed and click on the images).
  b. Right click on one of the selected images.
  c. Select the new label.



Setting the label weights

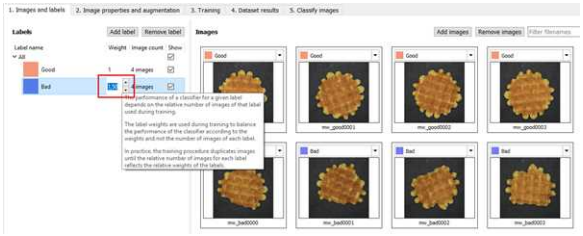
The label weights represent the relative importance that the deep learning tool gives to each class during the training.

> **TIP**
> The **EasySegment** tool does not use the label weights as it is trained on the images with one and only one specific label.

> **TIP**
> Increase the weight of a label to improve the accuracy of this label. Keep in mind that this also means a lowering of the accuracy of the other labels.

By default, all labels are given the same weight of 1.

- In **Deep Learning Studio**, set the label weights directly in the label list of the first tab:



- In the API, set the weight of a label with `EClassificationDataset::SetLabelWeight (labelId, weight)`.

ROI and Mask

Setting a ROI

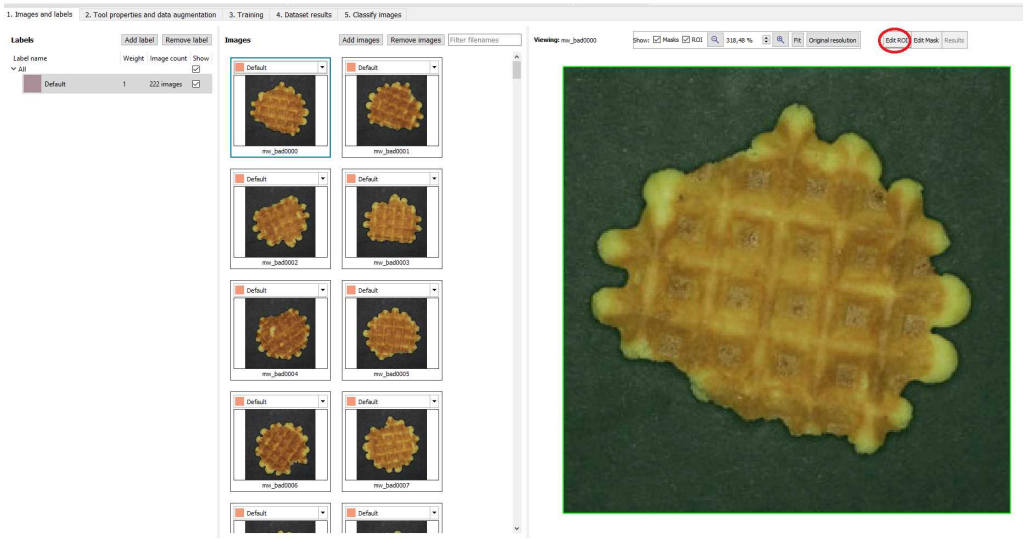
Use an ROI (region of interest) to crop an image or a whole dataset to a rectangular area aligned with the axis.

In the API:

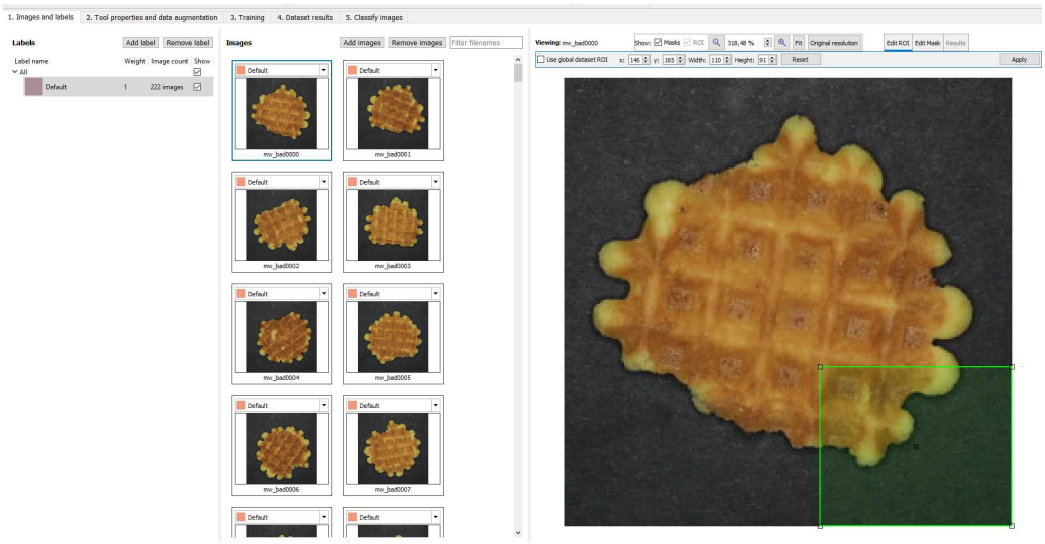
- To define a global ROI, use the `SetGlobalRegionOfInterest` method (set the ROI origin, width and height).
- To define a different ROI for each image, specify the ROI when you add an image to the dataset.

In Deep Learning Studio:

- To change the ROI:
  a. Select an image from the dataset.
  b. Click on the Edit ROI button.



c. Drag the ROI green box, or directly set the ROI origin, width and height.



- To set the same ROI for all the images of the dataset:

  a. Check the Use global dataset ROI box.

  b. Click on the Apply button.

Setting a mask

Set a mask on an image in a dataset to remove the pixels in the mask area from any computation. The mask works as a “don’t care area”.

In the API:

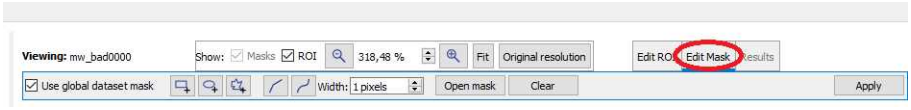
- To define a global mask, use the `SetGlobalMask` method (use a a `ERegion` to define the mask).
- To define a different mask for each image, specify the mask when you add an image to the dataset.

In Deep Learning Studio:

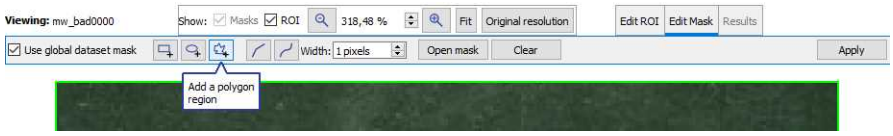
- To change the mask:

  a. Select an image from the dataset.

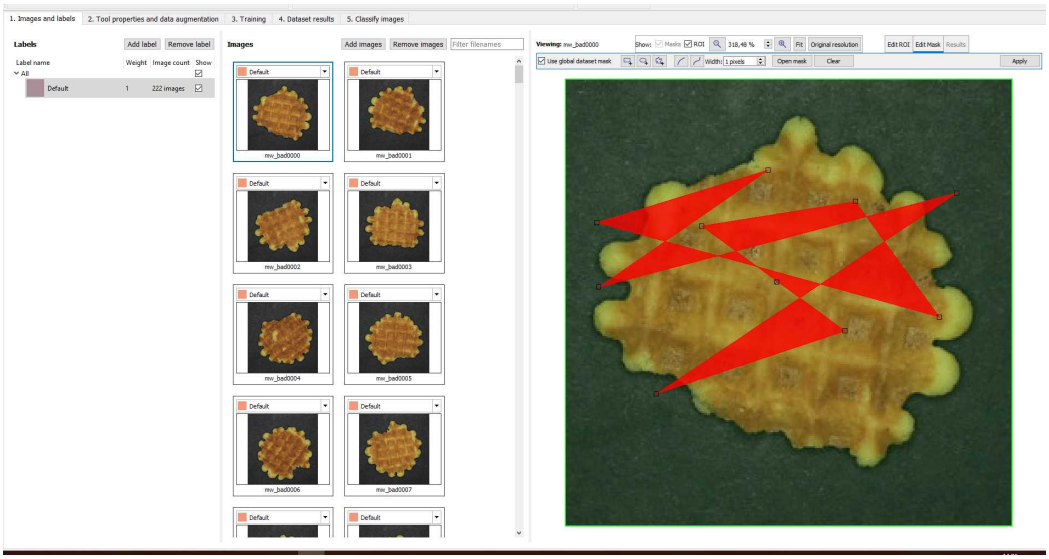
  b. Click on the Edit Mask button.



c. Select a kind of `ERegion` to draw the mask.



d. Draw the mask.



TIP
Click on the Open mask button to use an image to specify a mask. All the pixels of the image (such as an `EROIBW8`) that are over 127 are considered as part of the mask.

- To set the same mask for all the images of the dataset:

a. Check the Use global dataset mask box.

b. Click on the Apply button.

Training and Validation Datasets

It is important to use at least 2 separate datasets of images:

- A training dataset to train the classifier.
- A validation dataset to automatically select the best classifier during the training.
- An optional test dataset to evaluate the final performance of your classifier.

WARNING
These datasets MAY NOT contain:
- Images of the other datasets.
- Images of an object of interest extracted from images of the other datasets.

Deep Learning Studio automatically and randomly splits the dataset into a training and a validation dataset. Add images to the test dataset in the tab Classify images.

Why is it important?

Deep learning techniques can suffer from overfitting; this means that the trained classifier is too focused on the specific images present in the training dataset and it is not able to learn a general model of your data. Such tools perform poorly in production.

The validation dataset is used during training to prevent and know when overfitting occurs. This keeps the tool in a state that gives the best performance on the validation dataset. Without the validation dataset, it is impossible to know if a tool that performs well on its training dataset can perform well in production too.

Thus, a tool that gives high performance on the training dataset but much lower performance on the validation dataset has overfitted.

To fix overfitting:

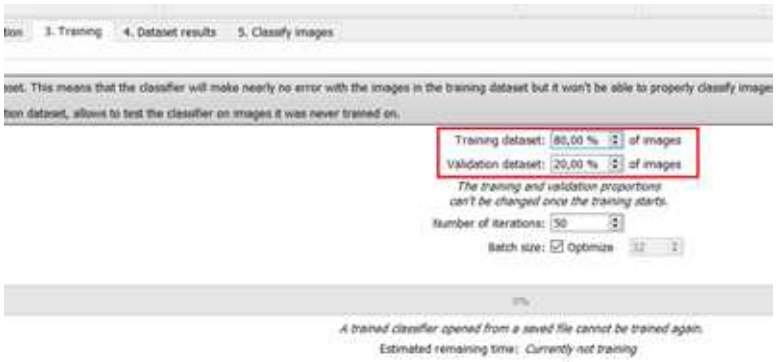
- You can add more images in your dataset.
- Or, in some cases, you can use data augmentation.

TIP
Data augmentation generates random transformations of the images in the training dataset to make the tool robust to geometric, luminosity or noise differences that are not present in the original training dataset.

Splitting the dataset

To create your training and validation datasets:

- In **Deep Learning Studio**:
  - Create a single dataset in the Images and labels tab.
  - Set the splitting percentages in the Training tab.
  - During the training, the dataset automatically splits into a training and a validation dataset according to this splitting percentage.



- In the API:
  - Create directly 2 `EClassificationDataset` objects containing 2 different sets of images.
  - Or randomly split an `EClassificationDataset` dataset into 2 parts with the method `EClassificationDataset::SplitDataset(trainingDataset, validationDataset, trainingProportion)`.

Using Data Augmentation

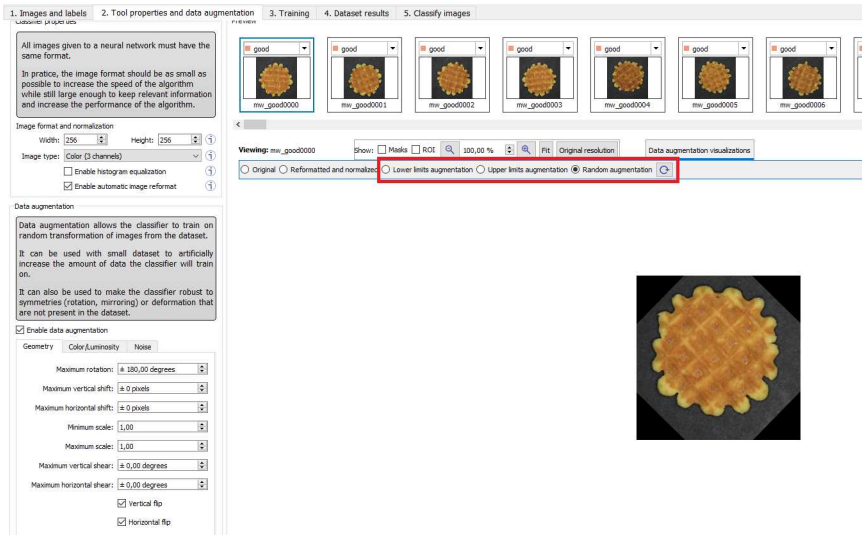
Data augmentation performs random transformations on images given to a deep learning tool (`EClassifier` or `EUnsupervisedSegmenter` object) during the training.

- Experiment different settings to choose the best parameters for your data augmentation.
- Check that the transformations do not change the label of an image (for example a defect that disappears because of a rotation or a contrast change).

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations.

In Deep Learning Studio:

- Configure the data augmentation in the second tab (Image properties and augmentation).
- Display and review the data augmented images with the minimum settings (Lower limits augmentation), the maximum settings (Upper limits augmentation) or the random settings (Random augmentation).

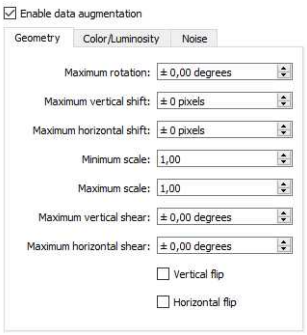


In the API:

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations.

The possible transformations are:

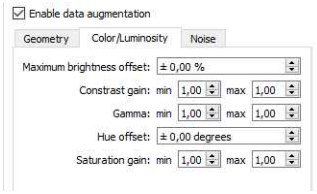
Geometric transformations



- Horizontal and vertical flips (enabled with `EClassificationDataset::SetEnableHorizontalFlip` and `EClassificationDataset::SetEnableVerticalFlip`)
- Scaling (between a minimum and maximum value defined with `EClassificationDataset::SetMinScale` and `EClassificationDataset::SetMaxScale`)

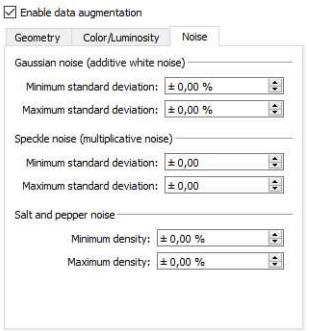
- Horizontal and vertical shifts (between `-maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxHorizontalShift(maxValue)` and `EClassificationDataset::SetMaxVerticalShift(maxValue)`)
- Rotations (between 0 and a maximum value defined with `EClassificationDataset::SetMaxRotationAngle`)
- Horizontal and vertical shear (between `-maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxHorizontalShear` and `EClassificationDataset::SetMaxVerticalShear`)

Color and luminosity transformations



- Brightness offset (between `-maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxBrightnessOffset`)
- Contrast gain (between a minimum and maximum value defined with `EClassificationDataset::SetMinContrastGain` and `EClassificationDataset::SetMaxContrastGain`)
- Gamma corrections (between a minimum and maximum value defined with `EClassificationDataset::SetMinGamma` and `EClassificationDataset::SetMaxGamma`)
- Hue offset (between `-maxValue` and `maxValue` defined with `EClassificationDataset::SetMaxHueOffset`)
- Saturation gain (between a minimum and maximum value defined with `EClassificationDataset::SetMinSaturationGain` and `EClassificationDataset::SetMaxSaturationGain`)

Noise transformations



> **TIP**
> The standard deviation is expressed as a percentage of the maximum pixel value.

- Gaussian noise, also called additive white noise, generated with a standard deviation (between a minimum and maximum value defined with `EClassificationDataset::SetGaussianNoiseMinimumStandardDeviation` and `EClassificationDataset::SetGaussianNoiseMaximumStandardDeviation`)
- Speckle noise, a multiplicative noise, generated from a Gamma distribution with a mean of 1 and a standard deviation (between a minimum and a maximum value defined with `EClassificationDataset::SetSpeckleNoiseMinimumStandardDeviation` and `EClassificationDataset::GetSpeckleNoiseMinimumStandardDeviation`).
- Salt and pepper noise generated from a pixel density (between a minimum and a maximum value defined with `EClassificationDataset::SetSaltAndPepperNoiseMinimumDensity` and `EClassificationDataset::SetSaltAndPepperNoiseMaximumDensity`).

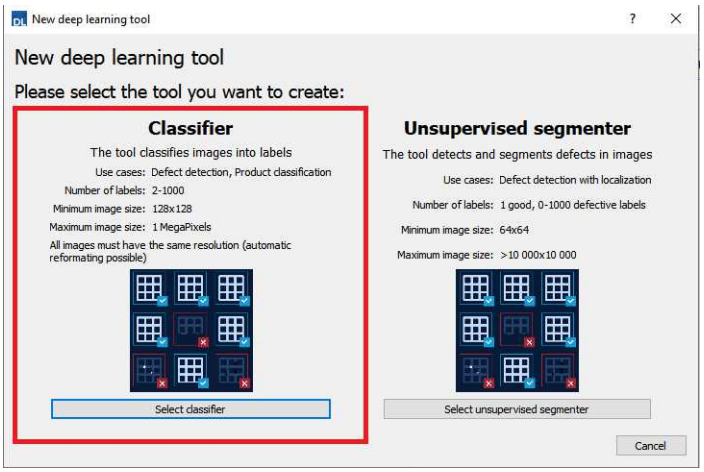
EasyClassify - Classifying Images

EasyClassify is the deep learning classification library of **Open eVision** (`EClassifier` class).

Deep Learning Studio

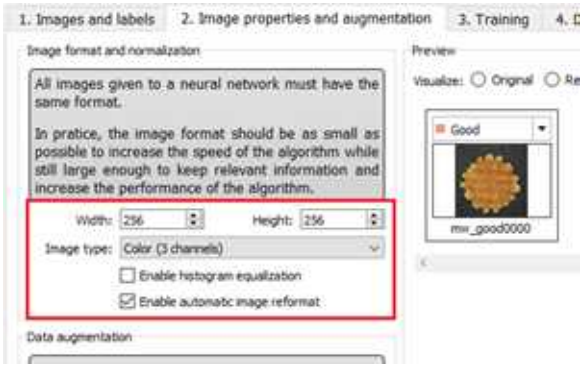
To create a classification tool in Deep Learning Studio:

1. Start Deep Learning Studio.
2. Select Classifier in the New deep learning tool dialog.

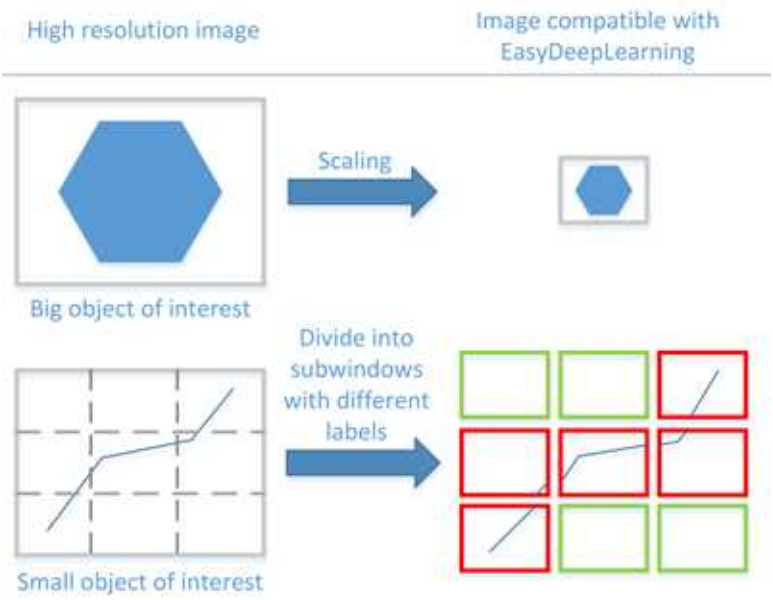


Input image format and normalization

- The input image format must have the width, height and number of channels corresponding to the input of the neural network.
- By default, a classifier uses the image format of the first image inserted in the training dataset:
  - All other images are automatically reformatted (anisotropic rescaling and conversion between color and grayscale).
  - If `EClassifier::SetEnableAutomaticImageReformat(false)` is called, the classifier throws an exception when attempting to train or classify an image that does not have the correct image format.
- In **Deep Learning Studio**, you can set the input image format in the Image properties and augmentation tab.



- In the API, you can also set manually the input image format with the methods `SetWidth`, `SetHeight` and `SetChannels` (1 channel for grayscale images and 3 channels for color images).
- The input image format must have a resolution between 128 x 128 and 1024 x 1024.
  For the best processing speed, use the lowest resolution at which your "objects of interest" are still recognizable.
  - If your original images are smaller than the minimum resolution, upscale them to a resolution higher or equal to 128 x 128.
  - If your original images are larger than the maximum resolution, lower the resolution:
    - If the "objects of interest" are still recognizable, explicitly set the input image format of the classifier to this lower resolution.
    - If the "objects of interest" are not recognizable, divide your original images into sub-windows and use these sub-windows to train the classifier and make prediction. This presents the additional advantage of localizing the "object of interest" inside the original image.



Histogram equalization

The classifier can also apply an histogram equalization to every input image:

- In **Deep Learning Studio**, activate it in the image format controls in the Image properties and augmentation tab.
- In the API, use `EClassifier::SetEnableHistogramEqualization(true)` to activate it.

Training the classifier

In the API, to train a classifier, call the method `EClassifier::Train(trainingDataset, validationDataset, numberOfIterations)`.

- Iteration:
  - An iteration corresponds to going through all the images in the training dataset once.
  - The training process requires a large number of iterations to obtain good results.
  - The default number of iterations is 50.
  - The larger the number of iterations, the longer the training is and the better the results you obtain.

TIP
Calling the `EClassifier::Train()` method several times with the same training and validation dataset is equivalent to calling `EClassifier::Train()` once but with a larger number of iterations. The total number of iterations used to train the classifier is accessible through `EClassifier::GetNumTrainedIterations()`.

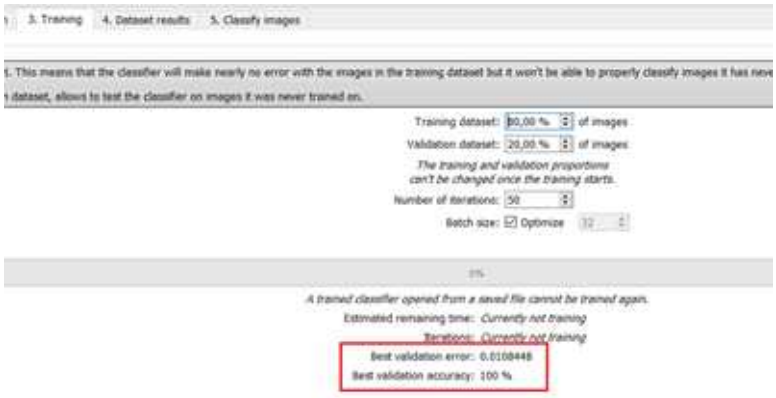
- The training process is asynchronous:
  - A call to `EDeepLearningTool::Train` launches a new thread that does the training in background.
  - The method `EDeepLearningTool::WaitForTrainingCompletion` suspends the program until the whole training is completed.
  - The method `EDeepLearningTool::WaitForIterationCompletion` suspends the program until the current iteration is completed.
  - During the training, use `EDeepLearningTool::GetCurrentTrainingProgression()` to follow the progression of the training.
- Batch size:

  The batch size corresponds to the number of images that are processed together.
  - The training is influenced by the batch size.
  - A large batch size increases the processing speed of a single iteration on a GPU but requires more memory.
  - The training process is not able to learn a good model with too small batch sizes.
  - By default, the batch size is determined automatically during training to optimize the training speed with respect to the available memory.
    - Use `EDeepLearningTool::SetOptimizeBatchSize(false)` to disable this behavior.
    - Use `EDeepLearningTool::SetBatchSize` to change the size of your batch.
  - Use `EDeepLearningTool::GetBatchSizeForMaximumInferenceSpeed()` to get the batch size that maximizes the (batch) classification speed on a GPU with respect to the available memory.
  - It is common to choose powers of 2 as batch size for performance reasons.

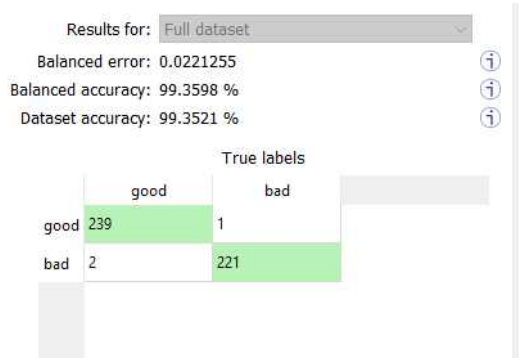
Validating the results

In Deep Learning Studio:

- The metrics are always computed without applying data augmentation on the images.
- In the Training tab, the metrics Best validation error and Best validation accuracy are computed during the training using the label weights.



- In the Dataset results tab, there are 3 metrics displayed:
  - The weighted error and the weighted accuracy (normalized with respect to the label weights instead of being dependent of the number of images for each label).
  - The dataset accuracy (it does not use the label weights).



TIP
If your dataset has a very different number of images for each of the labels, it is called *unbalanced*. In this case, the dataset accuracy is biased towards the labels containing the most images (the dataset accuracy mainly reflects the accuracy of these labels).

- In the Dataset results tab, the confusion matrix shows the number of images according to their true labels and their label predicted by the classifier.
  - The diagonal elements of the matrix shown in green are the correctly classified images.
  - All the other elements of the matrix are badly classified images.
  - Select one or more elements of the matrix to show the corresponding images.

In the API:

- After the completion of each iteration, **EasyClassify** automatically computes several performance metrics about the training and validation dataset:
  - Call the methods `EClassifier::GetTrainingMetrics(iteration)` and `EClassifier::GetValidationMetrics(iteration)` to read these metrics.
  - The iterations are indexed between 0 and `EClassifier::GetNumTrainedIterations()`-1.
  - Call `EClassifier::GetBestIteration()` to retrieve the iteration that produced the best performance.
  - After the training, the classifier is back in the state corresponding to this best iteration.
- The metrics are represented by an `EClassificationMetrics` object that contains the following performance metrics:
  - The classification error (`EClassificationMetrics::GetError()`), also called the cross-entropy loss: the quantity that is minimized during the training. It is computed from the probabilities computed by the classifier.
    - The error for a single image is the negative of the logarithm of the probability corresponding to the true label of the image. So, if this probability is low, the error for the image is high.
    - The error of the dataset is the average of the errors of each image in the dataset.
  - The classification accuracy (`EClassificationMetrics::GetAccuracy()`): the number of images correctly classified divided by the total number of images in the dataset.
  - The confusion matrix (`EClassificationMetrics::GetConfusion(groundtruthLabel, predictedLabel)`): the number of images labeled as `groundtruthLabel` that are classified as `predictedLabel`.

TIP
Call `EClassifier::Evaluate` to evaluate a dataset independently of the training.

Classifying new images

- In **Deep Learning Studio**, open Classify images tab to:
  - Classify new images.
  - Display detailed results for each image of the main dataset.

- Once the classifier is trained, call `EClassifier::Classify` to classify an Open eVision image.

  This method returns a `EClassificationResult` object:

  - `EClassificationResult::GetBestLabel()` returns the most probable label for the image.
  - `EClassificationResult::GetBestProbability()` returns the probability associated with the most probable label.
  - `EClassificationResult::GetProbability(label)` returns the probability associated with the given label.
  - `EClassificationResult::GetRanking(label)` returns the ranking of the given label. The ranking goes from 1 (most probable) to `EClassifier::GetNumLabels()` (least probable).

- You can also do batch classification or directly classify a vector of Open eVision images:
  - Images are processed together in groups determined by the batch size.
  - On a GPU, it is usually much faster to classify a group of images than a single image.
  - On a CPU, implement a multithread approach to accelerate the classification. In that case, each thread must have its own instance of `EClassifier` (see code snippets).

TIP
The batch classification has a tradeoff between the throughput (the number of images classified per second) and the latency (the time needed to obtain the result of an image): on a GPU, the higher the batch size, the higher the throughput and the latency. So, use batch classification to improve the classification speed at the cost of a longer time before obtaining the classification result of an image.

- Use `EClassifier::GetHeatmap(img, label)` to obtain an heat map highlighting the pixels that contribute the most to a label. In some cases, this heat map can provide a rough localization of the object corresponding to the label.

Memory requirements

- In addition to the properties of the classifier object and the weights of the neural network, an `EClassifier` object dynamically allocates memory for intermediate results during the training and the classification of new images.
- The size of the intermediate results depends on the width (W), height (H), batch size (B), and whether the operations are performed on a GPU or a CPU.
- For training, these intermediate results need about the following amount of memory:

$$\mathrm{TrainingMemoryCPU} = 0.000453 \times W \times H \times B - 292\ (\mathrm{MB})$$
$$\mathrm{TrainingMemoryGPU} = 0.000440 \times W \times H \times B + 25\ (\mathrm{MB})$$

- For classification, these intermediate results need up to the following memory:

$$ClassificationMemoryCPU = 0.000232 \times W \times H \times B - 97\ (MB)$$
$$ClassificationMemoryGPU = 0.000226 \times W \times H \times B + 13\ (MB)$$

- For example, training a classifier or making classifications with 256 x 256 images and a batch size of 32 on a GPU will take around respectively 950 MB or 500 MB.

TIP
Since large memory allocations take a lot of time, a classification does not released this memory and the next classifications can reuse it as long as the width, height, batch size and computation device remain the same. As such, the first classification is always be slower due to the memory allocations.

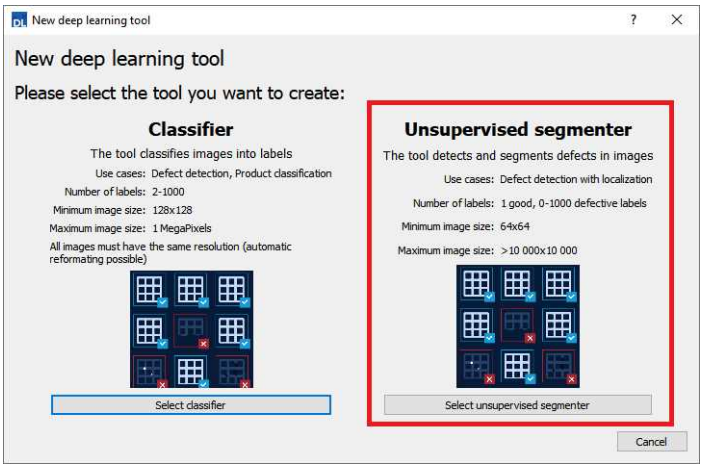
EasySegment - Detecting and Segmenting Defects

EasySegment is the deep learning segmentation library of Open eVision. It contains the unsupervised segmentation tool (`EUnsupervisedSegmenter` class).

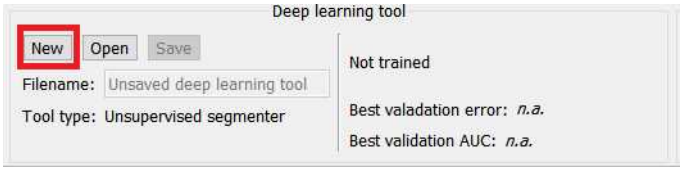
Deep Learning Studio

To create create an unsupervised segmentation tool in Deep Learning Studio:

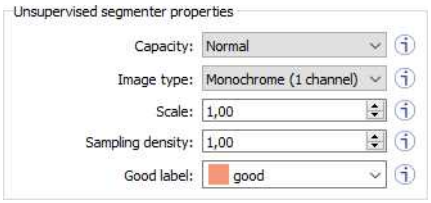
1. Start Deep Learning Studio.
2. Select Unsupervised segmenter in the New deep learning tool dialog.



The following dialog is displayed at the start of Deep Learning Studio or when you create a new deep learning tool from the toolbar.



Configuration



The unsupervised segmenter tool has 5 parameters:

1. The Capacity of the neural network (default: Normal) represents the quantity of information it is capable of learning. A larger capacity makes the tool slower.

   In the API:

   - The capacity is represented by the enumerate type `EUnsupervisedSegmenterCapacity`.
   - `EUnsupevisedSegmenter::Capacity` sets the capacity of the tool.

2. The Image type (default: Monochrome (1 channel)):

   In the API:

   - To use monochrome (grayscale, 1 channel) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `true`.
   - To use color (3 channels) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `false`.

3. Use the Scale (`EUnsupervisedSegmenter::Scale`) to automatically resize your images to a lower resolution and accelerate the processing.

4. The Sampling density (`EUnsupervisedSegmenter::SamplingDensity`) is the parameter of the sliding window algorithm used to process whole images using patches of size (`EUnsupervisedSegmenter::PatchSize`).

   - It indicates how much overlap there is between the image patches (100-100/`SamplingDensity` %).
   - In practice, the stride between 2 consecutive patches is `PatchSize`/`SampleDensity` pixels.

5. The Good label is the name of the class that contains the good images.

Training the tool

In the API, to train an unsupervised segmenter, call the `EDeepLearningTool::Train (trainingDataset, validationDataset, numberOfIterations)` method.

- An *iteration* corresponds to training on 10 000 image patches and computing the results for each training and validation image.
  - The training process requires a large number of iterations to obtain good results.
  - The default number of iterations is 50.
  - The larger the number of iterations, the longer the training is and the better the results you may obtain.

> **TIP**
> Calling the `EDeepLearningTool::Train` method several times with the same training and validation dataset is equivalent to calling it once but with a larger number of iterations.
> Call `EDeepLearningTool::GetNumTrainedIterations` to get the total number of iterations used to train the classifier.

- The training process is *asynchronous*:
  - `EDeepLearningTool::Train` launches a new thread that does the training in background.
  - `EDeepLearningTool::WaitForTrainingCompletion` suspends the program until the whole training is completed.
  - `EDeepLearningTool::WaitForIterationCompletion` suspends the program until the current iteration is completed.
  - During the training, `EDeepLearningTool::GetCurrentTrainingProgression` shows the progression of the training.
- The *batch size* corresponds to the number of image patches that are processed together.
  - The training is influenced by the batch size.
  - A large batch size increases the processing speed of a single iteration on a GPU but requires more memory.
  - The training process is not able to learn a good model with too small batch sizes.
  - By default, the batch size is determined automatically during the training to optimize the training speed with respect to the available memory.
    - Use `EDeepLearningTool::SetOptimizeBatchSize(false)` to disable this behavior.
    - Use `EDeepLearningTool::SetBatchSize` to change the size of your batch.
  - `EDeepLearningTool::GetBatchSizeForMaximumInferenceSpeed` gets the batch size that maximizes the batch classification speed on a GPU according to the available memory.
  - It is common to choose powers of 2 as the batch size for performance reasons.

Validating the results

There are 2 types of metric for the unsupervised segmentation tool:

- *Unsupervised* metric only uses the results of the tool on good images. There is only one unsupervised metric: the error.
- *Supervised* metrics requires both good and defective images. The supervised metrics are the AUC (Area Under ROC Curve), the ROC curve, the accuracy, the good detection rate (also called the true negative rate), the defect detection rate (also called the true positive rate).

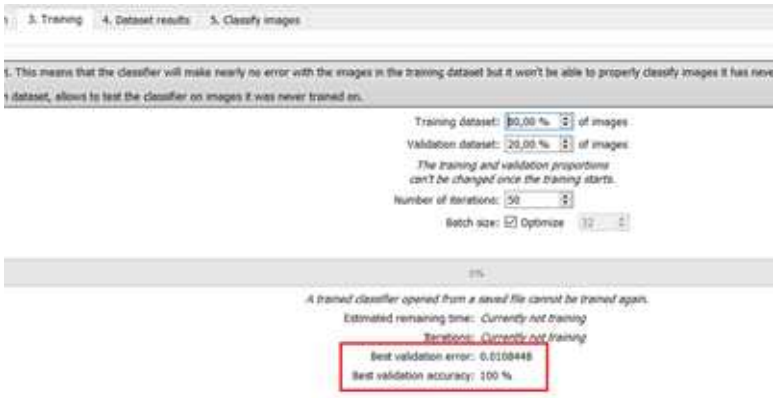
The unsupervised segmentation tool computes a score for each image (see `EUnsupervisedSegmenterResult::ClassificationScore`). The label of a result is obtained by thresholding this score with the segmenter classification threshold (`EUnsupervisedSegmenter::ClassificationThreshold`). So, the supervised metrics also depends on the value of this classification threshold.

The ROC curve (Receiver Operating Characteristic) is the plot of the defect detection rate (the true positve rate) against the rate of good images classified as defective (also called the false positive rate). It is obtained by varying the classification threshold. The ROC curve shows the possible tradeoffs between the good detection rate and the defect detection rate.

The area under the ROC curve (AUC) is independent of the chosen classification threshold and represents the overall performance of the tool. Its value is between 0 (bad performance) and 1 (perfect performance).

In Deep Learning Studio:

- In the Training tab, the metrics Best validation error and Best validation AUC are computed during the training on the validation dataset without using data augmentation. The validation error, the training error and the validation AUC are plotted for each iteration.



- In the Dataset results tab, various metrics, the confusion matrix, a cumulative score histogram, and the ROC curve are displayed. You can also change the classification threshold directly in this tab.
  - The cumulative score histogram shows the cumulative proportion of good (in green) and defective (in red) images with respect to the scores of the image.
  - You can change the classification threshold in 3 ways : direct input, dragging the threshold line in the score histogram and selecting a point on the ROC curve.

In the API:

- The metrics are represented by an `EUnsupervisedSegmenterMetrics` object that contains the following performance metrics:
  - The error on good image (`EUnsupervisedSegmenterMetrics::GetError`)
  - The confusion matrix (`EUnsupervisedSegmenterMetrics::GetConfusion`)
  - If the results for bad images are included in the metrics, `EUnsupervisedSegmenterMetrics::IsTotallyUnsupervised` is `false` and the following metrics are also be accessible:
    - The accuracy (`EUnsupervisedSegmenterMetrics::GetAccuracy`)
    - The Area under ROC curve (`EUnsupervisedSegmenterMetrics::GetAreaUnderROCCurve`)
    - The ROC point corresponding to the classification threshold (`EUnsupervisedSegmenterMetrics::GetROCPoint`)

Applying the tool to new images

In Deep Learning Studio:

- Open the Classify images tab to:
  - Apply the segmenter to new images.
  - Display detailed results for each image of the main dataset.
- Once the unsupervised segmenter is trained, call `EUnsupervisedSegmenter::Apply` to detect and segment defects in an **Open eVision** image.

  This method returns a `EUnsupervisedSegmenterResult` object:
  - `EUnsupervisedSegmenterResult::IsGood` and `EUnsupervisedSegmenterResult::IsDefective` returns whether the tool has decided that the image is good or defective according to the `EUnsupervisedSegmenterResult::ClassificationScore` and the `EUnsupervisedSegmenter::ClassificationThreshold`.

- `EUnsupervisedSegmenterResult::GetSegmentationMap` returns an `EImageBW8` image where all pixels with a value different than 0 are *defective* pixels.
  The value of a defective pixel is proportional to the importance of the defect at that position.
- `EUnsupervisedSegmenterResult::GetRegion` returns an `ERegion` object corresponding to the segmented region of the image (all the pixels of `EUnsupervisedSegmenterResult::GetSegmentationMap` that have a value strictly higher than 0).

4. Code Snippets

4.1. Basic Types

Loading and Saving Images

```
////////////////////////////////////////////////////////
// This code snippet shows how to load and save an image. //
////////////////////////////////////////////////////////
```

```
// Images constructor
EImageBW8 srcImage;
EImageBW8 dstImage;
```

```
// Load an image file
srcImage.Load("mySourceImage.bmp");
```

```
// ...
```

```
// Save the destination image into a file
dstImage.Save("myDestImage.bmp");
```

```
// Save the destination image into a jpeg file
// The default compression quality is 75
dstImage.Save("myDestImage.jpg");
```

```
// Save the destination image into a jpeg file
// set the compression quality to 50
dstImage.SaveJpeg("myDestImage50.jpg", 50);
```

Interfacing Third-Party Images

```
////////////////////////////////////////////////////////////
// This code snippet shows how to link an Open eVision image //
// to an externally allocated buffer.                        //
////////////////////////////////////////////////////////////
```

```
// Images constructor
EImageBW8 srcImage;
```

```
// Size of the third-party image
int sizeX;
int sizeY;
```

```
//Pointer to the third-party image buffer
EBW8* imgPtr;
```

```
// ...
```

```
// Link the Open eVision image to the third-party image
// Assuming the corresponding  buffer is aligned on 4 bytes
srcImage.SetImagePtr(sizeX, sizeY, imgPtr);
```

Retrieving Pixel Values

```
////////////////////////////////////////////////////////////
// This code snippet shows the recommended method (fastest)  //
// to access the pixel values in a BW8 image                 //
////////////////////////////////////////////////////////////
```

```
EImageBW8 img;
```

```
OEV_UINT8* pixelPtr;
OEV_UINT8* rowPtr;
OEV_UINT8  pixelValue;
OEV_UINT32 rowPitch;
OEV_UINT32 x, y;
```

```
rowPtr = reinterpret_cast <OEV_UINT8*>(img.GetImagePtr());
rowPitch = img.GetRowPitch();
```

```
for (y = 0; y < height; y++)
{
    pixelPtr = rowPtr;

    for (x = 0; x < width; x++)
    {
        pixelValue = *pixelPtr;

        // Add your pixel computation code here

        *pixelPtr = pixelValue;
        pixelPtr++;
    }

    rowPtr += rowPitch;
}
```

ROI Placement

```
////////////////////////////////////////////////////////////////
// This code snippet shows how to attach an ROI to an image  //
// and set its placement.                                     //
////////////////////////////////////////////////////////////////
```

```
// Image constructor
EImageBW8 parentImage;
```

```
// ROI constructor
EROIBW8 myROI;
```

```
// ...
```

```
// Attach the ROI to the image
myROI.Attach(&parentImage);
```

```
//Set the ROI position
myROI.SetPlacement(50, 50, 200, 100);
```

Vector Management

```
////////////////////////////////////////////////////////////////
// This code snippet shows how to create a vector, fill it   //
// and retrieve the value of a given element.                 //
////////////////////////////////////////////////////////////////
```

```
// EBW8Vector constructor
EBW8Vector ramp;
```

```
// Clear the vector
ramp.Empty();
```

```
// Fill the vector with increasing values
for(int i= 0; i < 128; i++)
{
        ramp.AddElement((EBW8)i);
}
```

```
// Retrieve the 10th element value
EBW8 value= ramp[9];
```

Exception Management

```
//////////////////////////////////////////
// This code snippet shows how to manage  //
// Open eVision exceptions.               //
//////////////////////////////////////////
```

```
try
{
        // Image constructor
        EImageC24 srcImage;
```

```
        // ...
```

```
        // Retrieve the pixel value at coordinates (56, 73)
        EC24 value= srcImage.GetPixel(56, 730);
}
```

```
catch(Euresys::Open_eVision_1_1::EException exc)
{
        // Retrieve the exception description
        std::string error = exc.What();
}
```

4.2. Deep Learning Tools

Creating a Dataset and Training a Classifier

```
//////////////////////////////////////////////////////////////
// This code snippet shows how to create a dataset, train a  //
// classifier and get the best performance metrics obtained  //
// during the training.                                      //
//////////////////////////////////////////////////////////////

// Creating dataset and classifier objects
EClassificationDataset dataset;
EClassificationDataset trainingDataset;
EClassificationDataset validationDataset;
EClassifier classifier;

// Adding images using a glob pattern
dataset.AddImages("*good*.png", "good");
dataset.AddImages("*defective*.png", "defective");

// Enabling data augmentation on the dataset
dataset.SetEnableDataAugmentation(true);

// Rotation of up to 90°
```

```
dataset.SetMaxRotationAngle(90);

// Enabling horizontal flips
dataset.SetEnableHorizontalFlip(true);

// Splitting the dataset with 80% of images for the training dataset
// and 20% for the validation dataset
dataset.Split(trainingDataset, validationDataset, 0.8);

// Training the classifier for 50 epochs
classifier.Train(trainingDataset, validationDataset, 50);
classifier.WaitForTrainingCompletion();

// Get the best metrics obtained on the validation dataset
EClassificationMetrics bestMetrics = classifier.GetValidationMetrics
(classifier.GetBestEpoch());
```

Loading a Classifier and Classifying a New Image

```
////////////////////////////////////////////////////////////
// This code snippet shows how load a trained classifier and //
// classify a new image.                                    //
////////////////////////////////////////////////////////////

// Image and classifier constructor
EClassifier classifier;
EImageBW8 srcImage;

// String and probability for the most probable result
std::string label;
float probability;

// Load classifier and image
classifier.Load(...);
srcImage.Load(...);

// Classify image
EClassificationResult result = classifier.Classify(srcImage);

// Get the most probable label
label = result.GetBestLabel();
probability = result.GetBestProbability();
```

Using Multithreading for Classification

```
////////////////////////////////////////////////////////////
// This code snippet shows how to parallelize the           //
// classification of new images on the CPU.                 //
// This code snippet is in C++ 11 and requires a recent     //
// compiler.                                                //
////////////////////////////////////////////////////////////

#include <thread>
#define NUM_THREADS 4

void task(EasyDeepLearning::EClassifier& classifier, EImageC24 img)
{
  // Classification of the image
  EasyDeepLearning::EClassificationResult result = classifier.Classify(img);
```

```
  std::string label = result.GetBestLabel();
  float proba = result.GetBestProbability();

  // Perform other actions based on the result
  ...
}
...
// Vector of classifier: one per thread
std::vector<EasyDeepLearning::EClassifier> classifiers;
classifiers.resize(NUM_THREADS);
for (int i = 0; i < NUM_THREADS; i++)
{
  classifiers[i].Load("classifier.ecl");

// Our thread pool
std::vector<std::thread> threads;
threads.resize(NUM_THREADS);

// The next thread to use
int threadToUse = 0;
bool hasImage = true;
while (hasImage)
{
  EImageC24 image;

  // Load or set the data pointer of the image
  ...

  // Check that the threads has done its previous work
  if (threads[threadToUse].joinable())
  {
       threads[threadToUse].join();
  }

  // Launch a new thread
  threads[threadToUse] = std::thread(task, classifiers[threadToUse], image);
  threadToUse = (threadToUse + 1) % NUM_THREADS;

  // Check that we still have an image to process and change the status
  // of "hasImage" if necessary.
  ...
}

// Make sure that all threads are finished
for (int i = 0; i < NUM_THREADS; i++)
{
  if (threads[i].joinable())
       threads[i].join();
}
```

Loading an Unsupervised Segmenter and Segmenting an Image

```
////////////////////////////////////////////////////////////
// This code snippet shows how to load a trained          //
// unsupervised segmenter and how to segment a new image. //
////////////////////////////////////////////////////////////

// Image
EImageBW8 image;
```

```
Image.Load(...) ;

// Segmenter
EUnsupervisedSegmenter segmenter;
segmenter.Load(...);

// Apply the segmenter on the image
EUnsupervisedSegmenterResult r = segmenter.Apply(image);

// Retrieve the segmentation map
EImageBW8 segmentationMap = r.GetSegmentationMap();
```