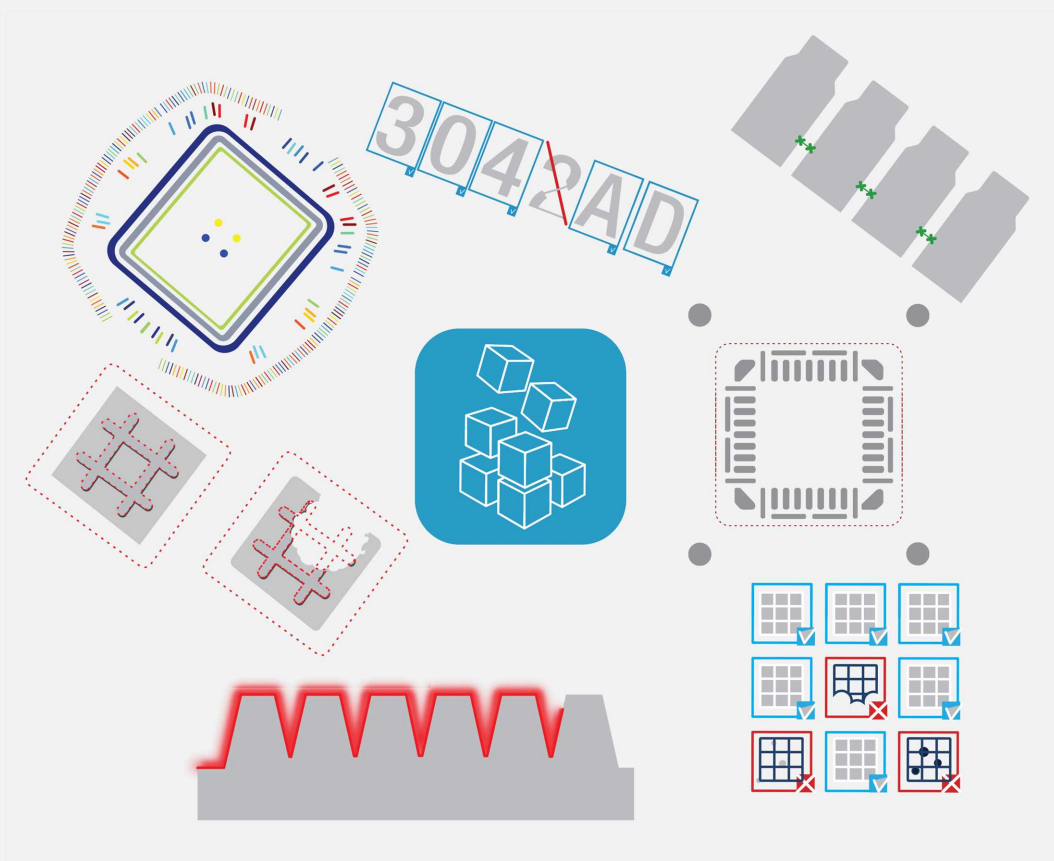


Open eVision

Deep Learning Inspection Tools



This documentation is provided with Open eVision 2.16.1 (doc build 1156).
www.euresys.com

Contents

1. Dealing with Pixel Containers and Files	5
1.1. Pixel Container Definition	5
1.2. Pixel Container Types	7
1.3. Supported Image File Types	8
1.4. Pixel and File Types Compatibility	9
1.5. Color Types	11
2. Manipulating Pixels Containers and Files	12
2.1. Pixel Container File Save	12
2.2. Pixel Container File Load	14
2.3. Memory Allocation	15
2.4. Image and Depth Map Buffer	16
2.5. Image Coordinate Systems	19
2.6. Image Drawing and Overlay	20
2.7. 3D Rendering of 2D Images	20
2.8. Vector Types and Main Properties	22
2.9. ROI Main Properties	26
2.10. Arbitrarily Shaped ROI (ERegion)	28
2.11. Flexible Masks	35
2.12. Profile	39
3. Deep Learning Tools	41
Deep Learning Tools - Inspecting Images with Deep Learning	41
Purpose and Workflow	41
Tools and Resources	44
Hardware Support (CPU/GPU)	47
Managing the Images	49
Images and Labels	49
Adding Images	51
Editing the Segmentation of an Image	52
Editing the Objects of an Image	54
ROI and Mask	56
Managing the Dataset	60
Training and Validation Datasets	60
Using Data Augmentation	62
Training a Deep Learning Tool	65
EasyClassify - Classifying Images	66
Tool and Images	66
Validating the Results	68
Classifying New Images	70
Benchmarks for EasyClassify	72
EasySegment - Detecting and Segmenting Defects	73
Unsupervised vs Supervised Modes	73
EasySegment Unsupervised	74
Tool and Configuration	74
Validating the Results	76
Applying the Tool to New Images	77
Benchmarks for EasySegment Unsupervised	78
EasySegment Supervised	80
Tool and Configuration	80
Using the Supervised Segmenter	82
Evaluating the Results	84
Benchmarks for EasySegment Supervised	89
EasyLocate - Locating Objects and Defects	91
Tool and Configuration	91
Locating Objects	95

Validating the Results	97
Benchmarks for EasyLocate	100
4. Code Snippets	102
4.1. Basic Types	103
Loading and Saving Images	103
Interfacing Third-Party Images	103
Retrieving Pixel Values	104
ROI Placement	104
Vector Management	105
Exception Management	105
4.2. Deep Learning Tools	106
Creating a Dataset and Training a Classifier	106
Loading a Classifier and Classifying a New Image	106
Using Multithreading for Classification	107
Loading an Unsupervised Segmenter and Segmenting an Image	108

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.

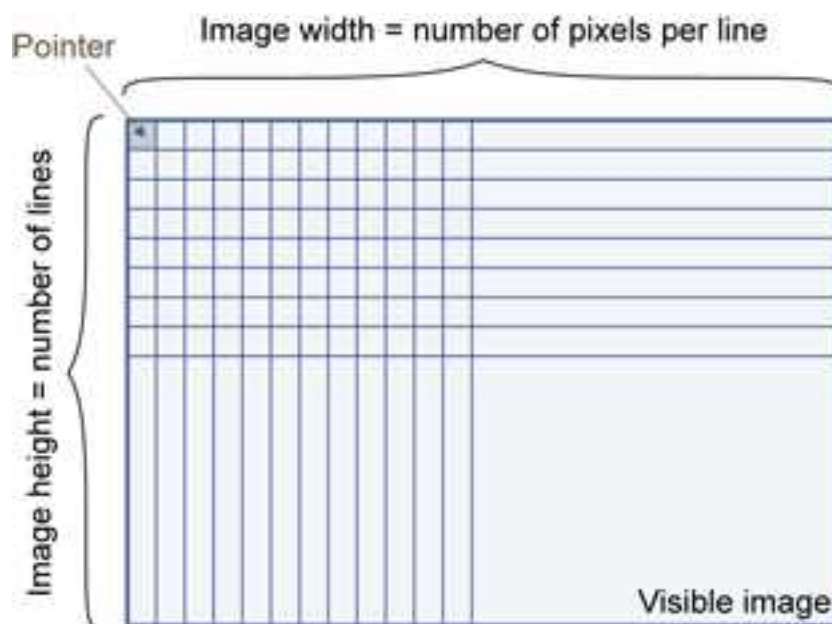


Image main parameters

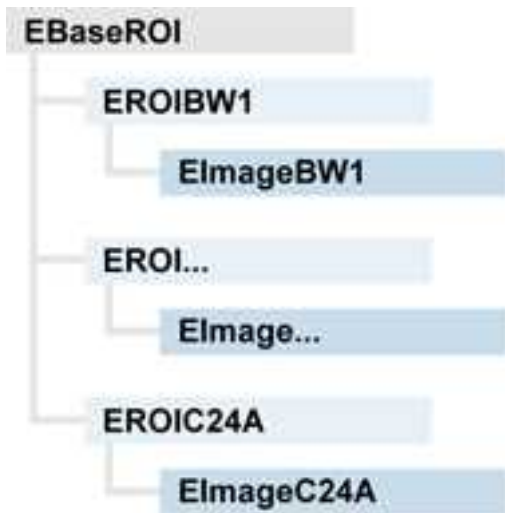
An Open eVision image object has a rectangular array of pixels characterized by [EBaseROI](#) parameters .

- **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

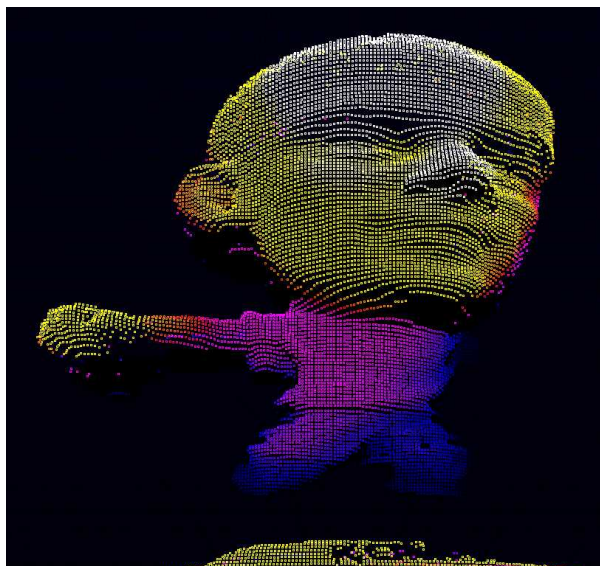
A depth map is a way to represent a 3D object using a 2D grayscale image where each pixel in the image represents a 3D point.



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

Reference

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)	EImageBW
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)	EImageBW
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.	EImageC

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)	EPointCloud
-------------	---	-----------------------------

1.3. Supported Image File Types

Reference

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 loses quality.
JFIF	JPEG File Interchange Format
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.

Type	Description
	<ul style="list-style-type: none"> - 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	<p>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.</p> <p>File save operations are lossless and use CCITT 1D compression for 1-bit binary pixel types and LZW compression for all others.</p> <p>File load operations support all TIFF variants listed in the LibTIFF specification.</p>

1.4. Pixel and File Types Compatibility

Depth map to image conversion

For 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 images 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 pixels 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

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

Saving point clouds

Use the following methods to save a point cloud in a specific format:

- `EPointCloud::Save`: Open eVision proprietary file format.
- `EPointCloud::SaveCSV`: CSV file.
- `EPointCloud::SaveOBJ`: OBJ file.
- `EPointCloud::SavePCD`: PCD file.
- `EPointCloud::SavePLY`: PLY file.
- `EPointCloud::SaveXYZ`: XYZ file.



TIP

The PCD format is supported in ASCII and binary modes.

2.2. Pixel Container File Load

Loading 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.

Loading point clouds

Use the following methods to load a point cloud saved in a specific format:

- `EPointCloud::Load`: Open eVision proprietary file format.
- `EPointCloud::LoadCSV`: CSV file.
- `EPointCloud::LoadOBJ`: OBJ file.
- `EPointCloud::LoadPCD`: PCD file.
- `EPointCloud::LoadPLY`: PLY file.
- `EPointCloud::LoadXYZ`: XYZ file.



TIP

- The PCD format is supported in ASCII and binary modes.
- The PLY is supported only in ASCII mode.

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 deallocates a buffer.

- Memory management is transparent.
- When the image size changes, reallocation occurs.
- When an image object is destroyed, the buffer is deallocated.

To declare an image with internal memory allocation:

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

- a. Declare an image object, for instance `EImageBW8`.
- b. Create a suitably sized and aligned buffer (see [Image Buffer](#)).
- c. Assign the buffer to the image with `SetImagePtr`.



NOTE

If your buffer rows are not aligned on 4 bytes, you cannot use `SetImagePtr`. In that case, use `InitializeFromUnalignedBuffer` instead.

Please note, however, that this allocates the memory internally and copies the external buffer into the internal one instead of using the external one directly.

2.4. Image and Depth Map Buffer

Image and depth map pixels are stored contiguously, from left to right and from top row to bottom row, in Windows bitmap format (top-down DIB -device-independent bitmap-) 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.

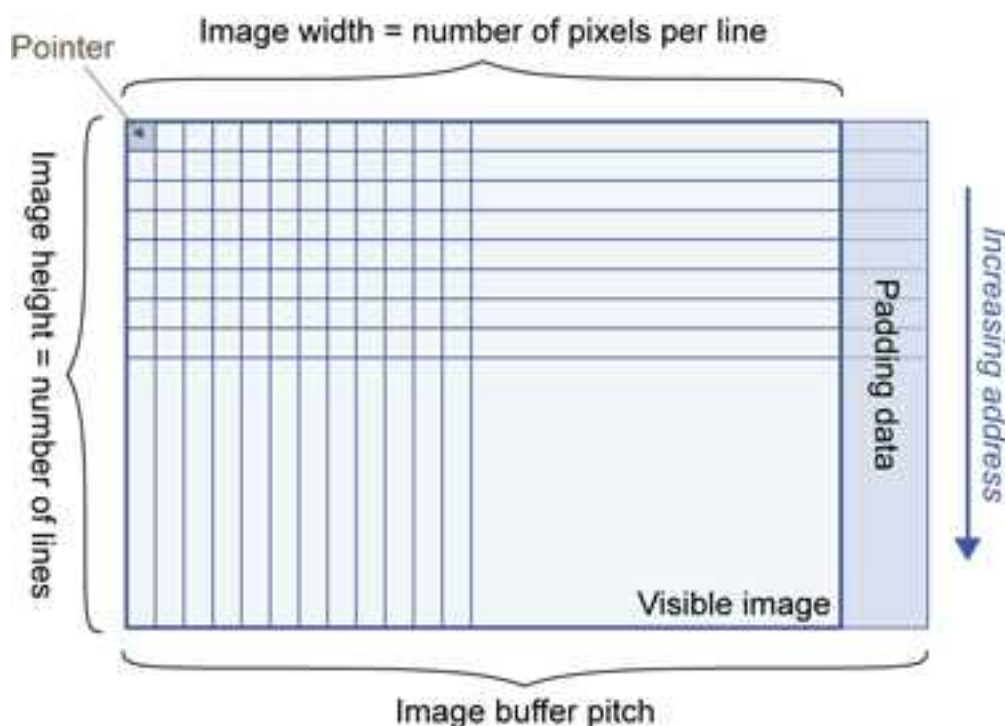


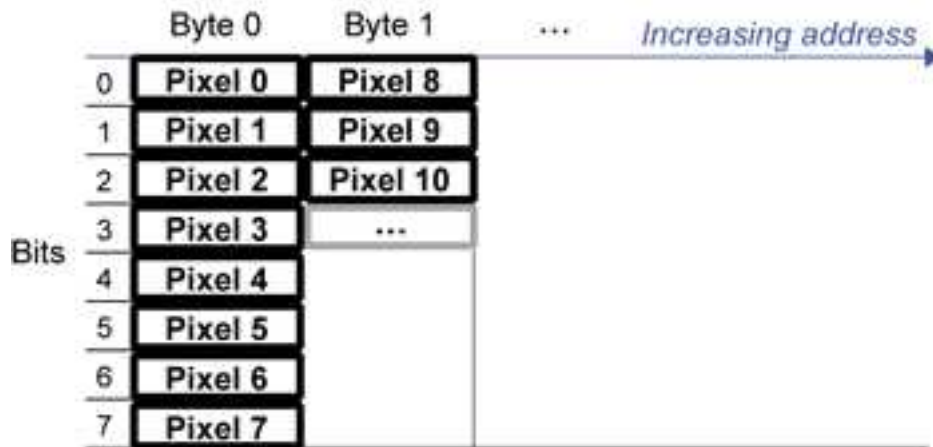
Image buffer pitch

- 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

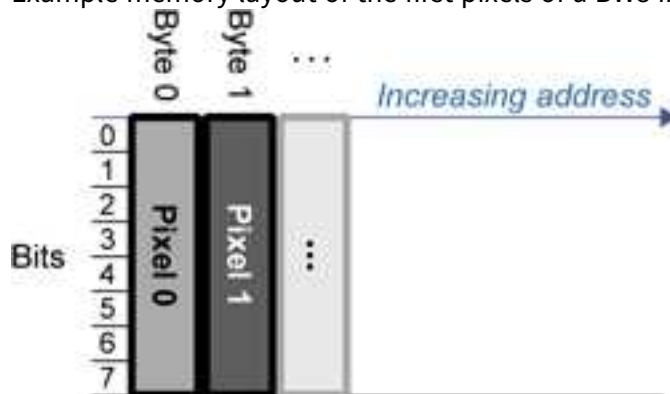
- `EImageBW_8` stores 8 pixels in one byte.

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



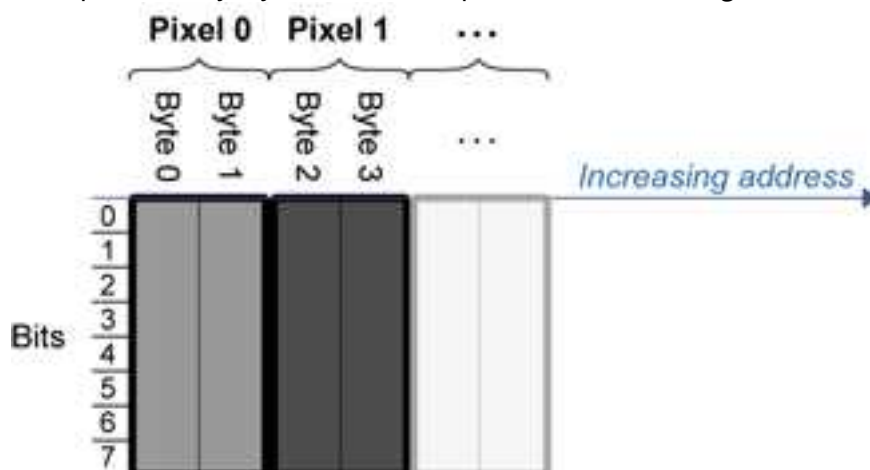
- `EImageBW_8` and `EDepthMap8` store each pixel in one byte.

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



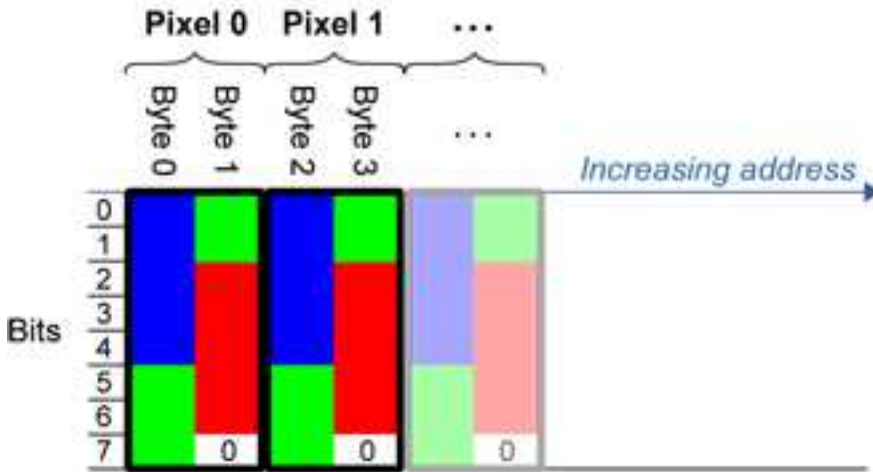
- `EImageBW_16` stores each pixel in a 16-bit word (two bytes).

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



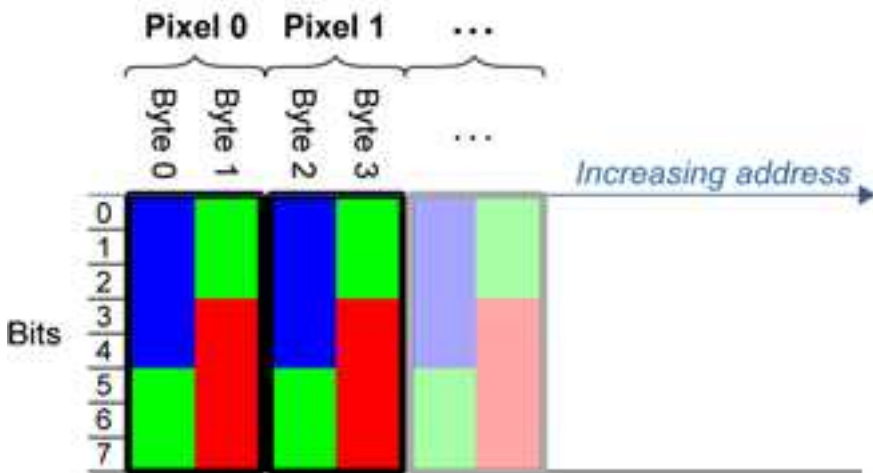
- `EImageC_5` stores each pixel in 2 bytes. Each color component is coded with 5-bits. The 16th bit is left unused.

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



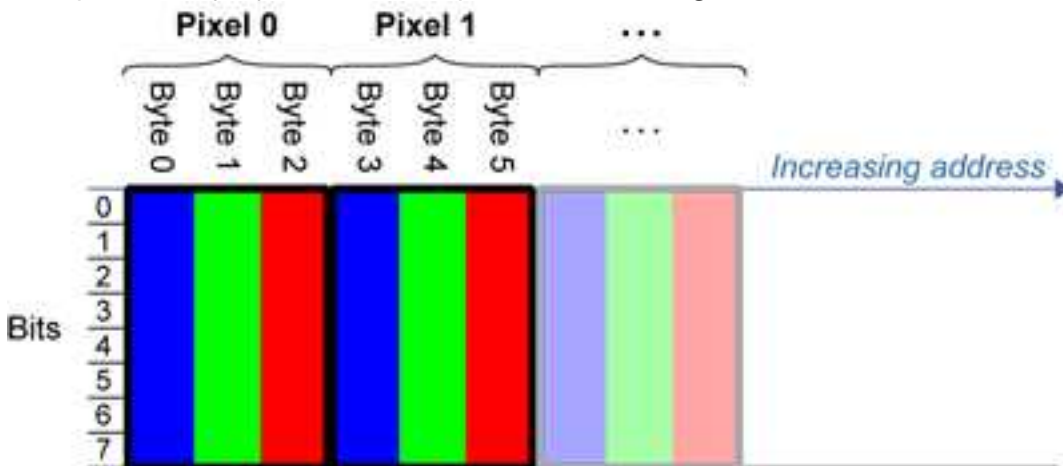
- [EImageC16](#) stores each pixel in 2 bytes. The first and third color components are coded with 5-bits. The second color component is coded with 6-bits.

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



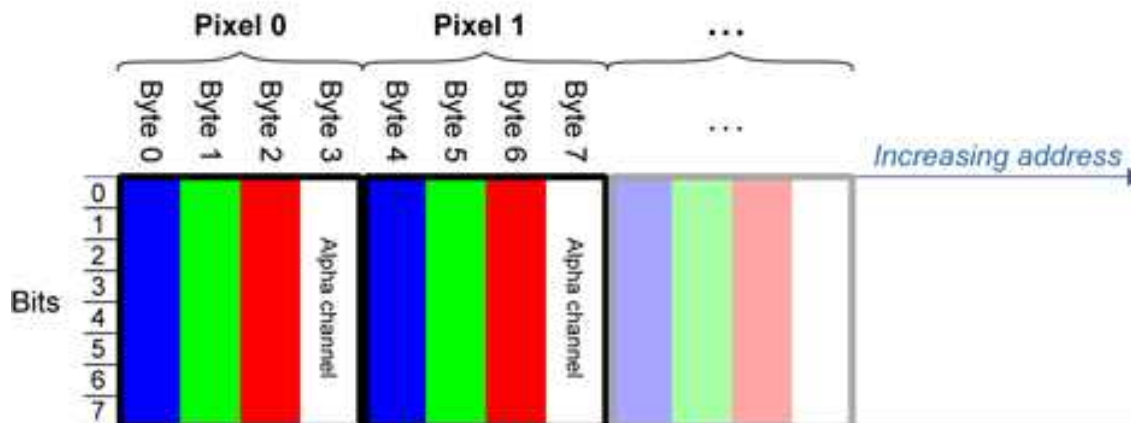
- [EDepthMap16](#) store each pixel in 2 bytes using a fixed point format.
- [EImageC24](#) stores each pixel in 3 bytes. Each color component is coded with 8-bits.

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.

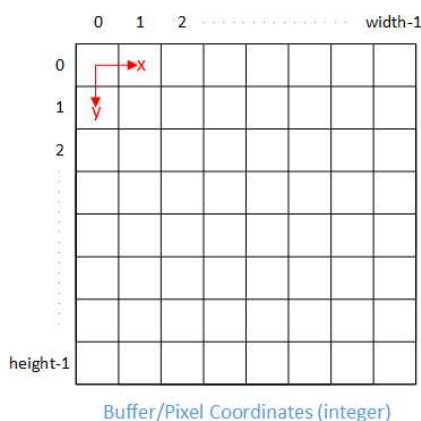
2.5. Image Coordinate Systems

The conventions below apply to all Open eVision functions and results.

- Pixel coordinates are usually given as integer numbers.
- Some results can use subpixel precision with real (floating point) numbers.
- Some exceptions apply and are documented per librarie.

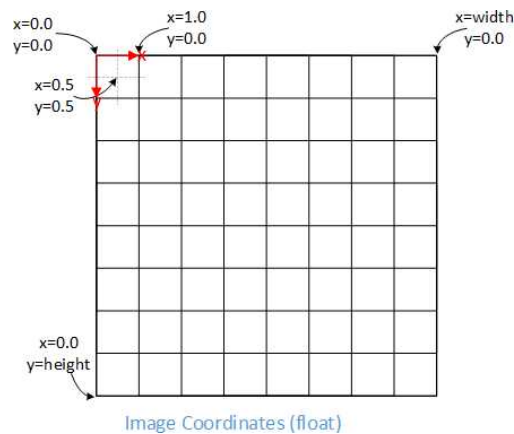
Integer coordinates

- The origin (0,0) of the coordinate system is the upper left pixel of the image.
- The lower right pixel is (width-1, height-1).



Real coordinates

- With floating point (x,y) coordinates, the origin is the upper left corner of the upper left pixel.
- The first pixel area ranges in $[0,1[$ for X and Y axis.
- Coordinates greater or equal than the width or the height are outside the image.



2.6. Image Drawing and Overlay

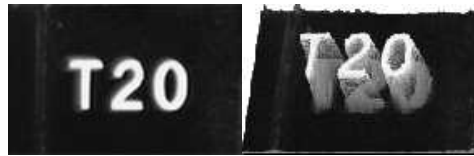
- Drawing uses Windows GDI (Graphics Device Interface) system calls.
 - MFC (Microsoft Foundation Class) applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
 - Borland/CodeGear OWL or VCL use a `Paint` event handler.
- The color palette in 256-color display mode gives optimal rendering.
- Gray-level images can be improved using LUTs (LookUp Tables) (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.

2.7. 3D Rendering of 2D Images

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

Gray 3D rendering

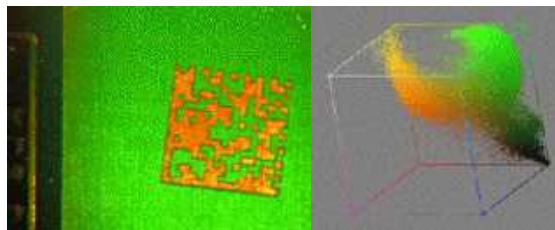
Easy: :Render3D prepares a 3-dimensional rendering where gray-level values are altitudes. Magnification factors in the three directions (X = width, Y = height and Z = depth) can be given. The rendered image appears as independent dots whose size can be adjusted to make the surface more or less opaque.



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 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.

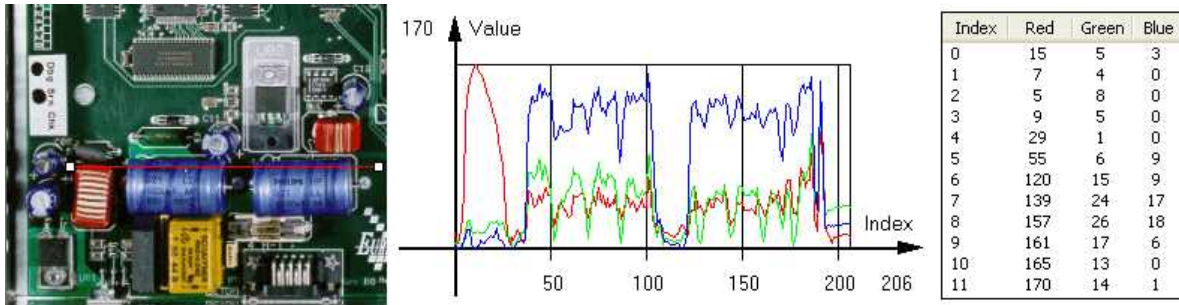


Color histogram rendering

2.8. 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 and RGB values array ([EC24Vector](#))

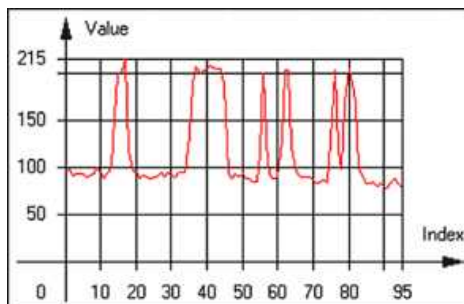
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

- To create a vector of the appropriate type:
 - Use its constructor and preallocate elements if required.
- To fill a vector with values:
 - Call the [EVector::Empty](#) member to empty it.
 - Call the [EC24Vector::AddElement](#) member to add elements one by one.
 - Use the indexing to access any element.
- To access a vector element, either for reading or writing:
 - Use the brackets operator [EC24Vector::operator\[\]](#).
- To determine the current number of elements:
 - Use the [EVector::NumElements](#) member.
- To 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 and annotations in black. You can define: `graphicContext`, `width`, `height`, `originX`, `originY`, `color0`, `color1` and `color2`.
 - The [EC24Vector](#) has three curves drawn instead of one, each corresponding to a color component. By default the red, blue and green pens are used.

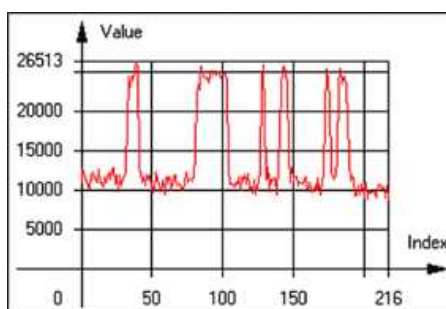
Vector types

- **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...`).



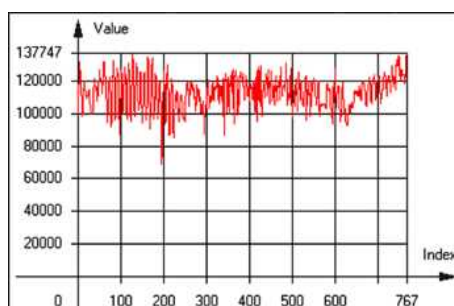
Graphical representation of an **EBW8Vector** (see [Draw](#) method)

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



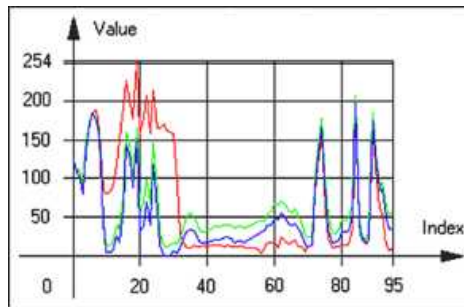
Graphical representation of an **EBW16Vector**

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



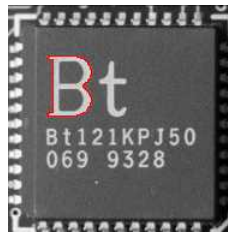
Graphical representation of an **EBW32Vector**

- **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`, ...).



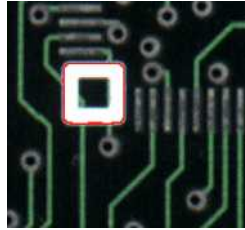
Graphical representation of an **EBW8PathVector** (see `Draw` method)

- **EBW6PathVector**: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).



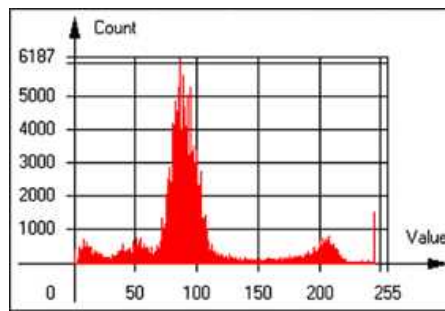
Graphical representation of an **EBW6PathVector** (see `Draw` method)

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



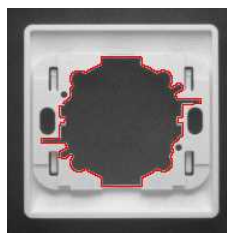
Graphical representation of an **EC24PathVector** (see [Draw method](#))

- **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`).

2.9. 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](#).

- [EROIBW](#)
- [EROIBW8](#)
- [EROIBW-6](#)
- [EROIBW32](#)
- [EROIC](#)
- [EROIC-5](#)
- [EROIC-6](#)
- [EROIC24](#)
- [EROIC24A](#)

Attachment

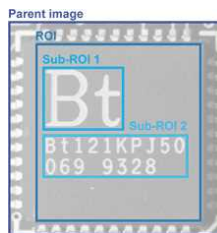
An ROI must be [attached](#) to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.

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).

2.10. 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
	<code>EasyImage::Threshold</code>
	<code>EasyImage::Copy</code>
	<code>EasyImage::ConvolKernel</code>
	<code>EasyImage::ConvolSymmetricKernel</code>
	<code>EasyImage::ConvolLowpass</code>
	<code>EasyImage::ConvolLowpass2</code>
	<code>EasyImage::ConvolLowpass3</code>
	<code>EasyImage::ConvolUniform</code>
	<code>EasyImage::ConvolGaussian</code>
	<code>EasyImage::ConvolHighpass</code>
	<code>EasyImage::ConvolHighpass2</code>
	<code>EasyImage::ConvolGradientX</code>
	<code>EasyImage::ConvolGradientY</code>
	<code>EasyImage::ConvolGradient</code>
	<code>EasyImage::ConvolSobelX</code>
	<code>EasyImage::ConvolSobelY</code>
	<code>EasyImage::ConvolSobel</code>
	<code>EasyImage::ConvolPrewittX</code>
	<code>EasyImage::ConvolPrewittY</code>
	<code>EasyImage::ConvolPrewitt</code>
	<code>EasyImage::ConvolRoberts</code>
	<code>EasyImage::ConvolLaplacianX</code>
	<code>EasyImage::ConvolLaplacianY</code>
	<code>EasyImage::ConvolLaplacian8</code>
	<code>EasyImage::DilateBox</code>
	<code>EasyImage::ErodeBox</code>
	<code>EasyImage::OpenBox</code>
	<code>EasyImage::CloseBox</code>
	<code>EasyImage::WhiteTopHatBox</code>
	<code>EasyImage::BlackTopHatBox</code>
	<code>EasyImage::MorphoGradientBox</code>
	<code>EasyImage::ErodeDisk</code>

Library	Method
	EasyImage::DilateDisk
	EasyImage::OpenDisk
	EasyImage::CloseDisk
	EasyImage::WhiteTopHatDisk
	EasyImage::BlackTopHatDisk
	EasyImage::MorphoGradientDisk
	EasyImage::Median
	EasyImage::ScaleRotate
	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
EasyGauge	EPointGauge::Measure
	ELineGauge::Measure
	ERectangleGauge::Measure
	ECircleGauge::Measure
	EWedgeGauge::Measure
EasyMatch	EMatcher::LearnPattern
	EMatcher::Match
EasyQRCode	EQRCodeReader::SetSearchField
	EQRCodeReader::Read

**TIP**

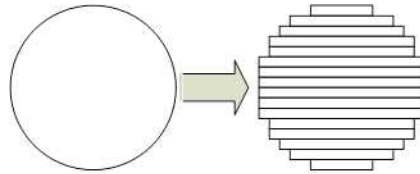
In the future Open eVision releases, the support of **ERegions** will be gradually extended to all operators.

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.

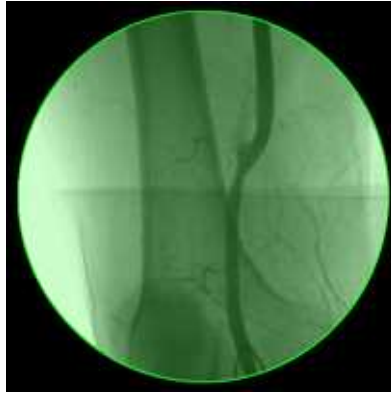
Use these classes to setup geometric regions and modify them with translation, rotation and scaling. The transformation operators return new regions, leaving the source object unchanged.

- [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.



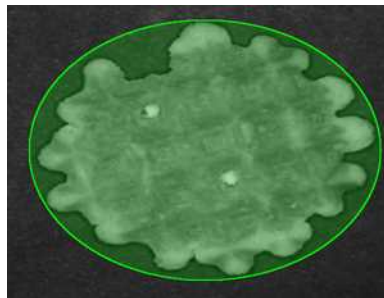
Rectangle region separating a bar code from the background

- **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`



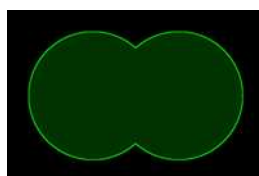
TIP

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

Combining regions

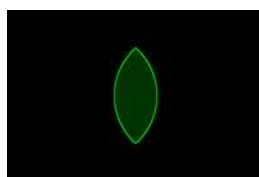
Use the following operations to create a new region by combining existing regions:

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



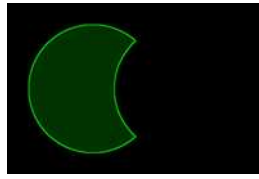
Union of 2 circles

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



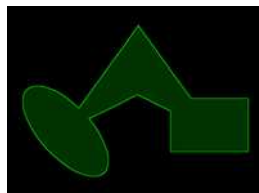
Intersection of 2 circles

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



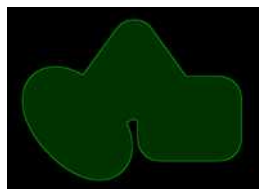
Subtraction of 2 circles

Morphological operations on regions



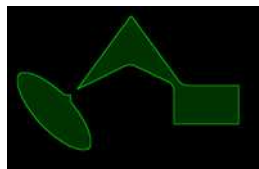
The initial arbitrary region used to illustrate the different morphological operations

- Grow
 - The `ERegion::Grow(int radius)` method returns a region that is the dilation of the region by a disk with a radius equals to the argument.



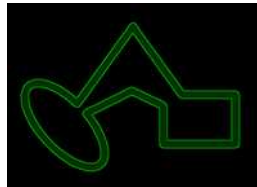
Grow of the arbitrary region

- Shrink
 - The `ERegion::Shrink(int radius)` method returns a region that is the erosion of the region by a disk with a radius equals to the argument.



Shrink of the arbitrary region

- Contour
 - The `ERegion::Contour(int thickness, bool centered = true)` method returns a region that is the contour of the region.



Contour of the arbitrary region

Free-hand drawing a region

- The `ERegionFreeHandPainter` class provides the methods that allow you to create a region by hand, using the mouse or any other user input method.
- The `RegionFreeHand` sample, available both in C++ and C#, shows how to use this class to draw a region on an image.

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 your first call to a method to take 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 methods 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.

2.11. Flexible Masks

ROIs vs flexible masks

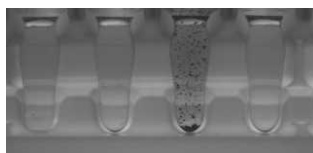
ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on page 26 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):

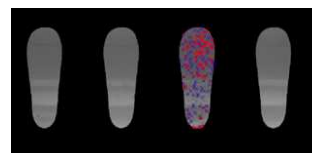
- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image



Associated mask

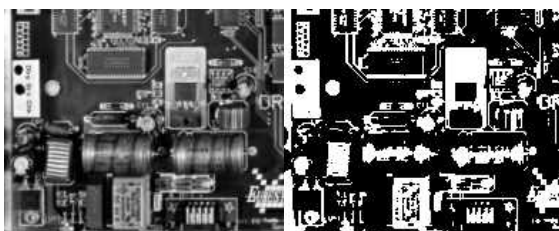


Processed masked image

A flexible mask can be generated by any application that outputs BW8 images and by some [EasyObject](#) and [EasyImage](#) functions.

Flexible Masks in EasyImage

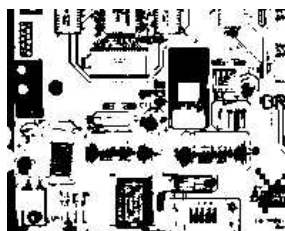
Code Snippets



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.

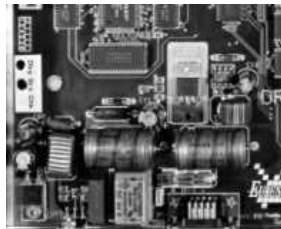
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

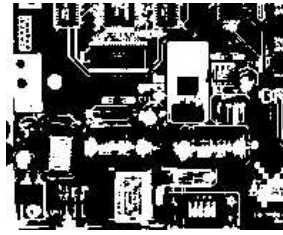


Source image

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())`.

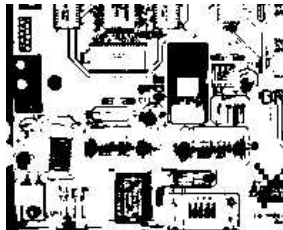
5. Exploit the flexible mask as an argument to `ECodedImage2::RenderMask`.



BW8 resulting image that can be used as a flexible mask

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

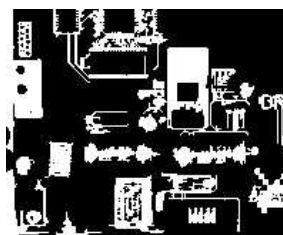
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

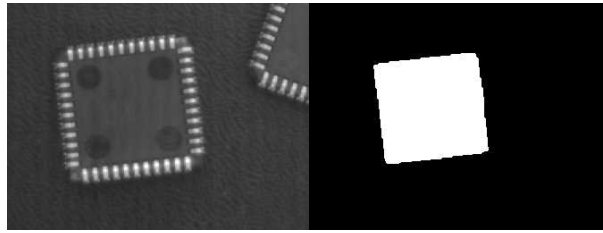
3) `EObjectSelection::RenderMask`: from a selection of blobs

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



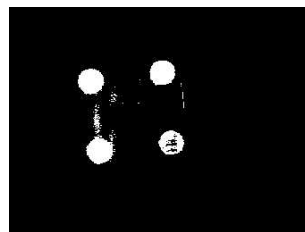
BW8 resulting image that can be used as a flexible mask

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.

2.12. Profile

Code Snippets

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 masks.
- 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 masks.

- 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 Supervised segments defects and/or various elements in images. In the supervised mode, the training images must be precisely annotated with their expected segmentation (also called the *ground truth*).
- EasySegment Unsupervised 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 advantages are that the tool can detect and segment defects that are not in your dataset or that are unexpected and that it doesn't require to annotate the images with their expected segmentation (this can be very time consuming).
- 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.
- You can use EasySegment Unsupervised to produce a rough annotation of the images required by EasySegment Supervised.
- EasyLocate locates objects and/or defects in images. The neural network predicts the location and the label of the object and/or defect.
 - The location of the object is represented by its bounding box.
 - EasyLocate works well with partially occluded objects.
 - You can use it for defect detection and counting applications.
 - Compared to EasySegment, it detects two objects or defects with the same label and that are overlapping or touching each other as two different objects or defects and not as a single blob.

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

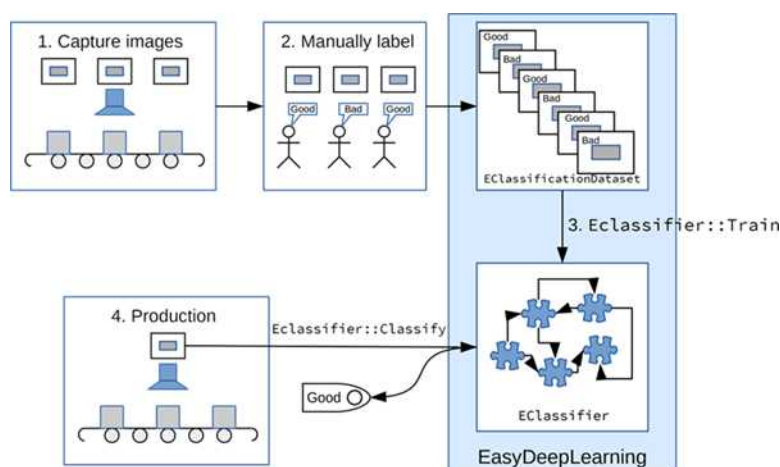
	EasyClassify	EasySegment Unsupervised	EasySegment Supervised	EasyLocate
Minimum image size	128 × 128	64 × 64		128 × 128
Maximum image size	1024 × 1024	10 000 × 10 000		500 000 pixels (for ex. 707 × 707 for square image)
Best image size	256 × 256 - 600 × 600	n.a.		n.a.
Number of channels	1 or 3 (grayscale and color images)			
Bit depth	8 bits, 16 bits			
Number of labels	2 - 1000	2 (good and defective)	2 - 64	
Minimum number of images per label	2	1 for the good label 0 for the defective label	1	
Supported formats	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 47 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 to 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.
 - With EasyClassify, you can use the training with as few as 10 images per label. Nevertheless, complex tasks may require more than 100 images per label.
 - With EasySegment Unsupervised, you can use less than 10 “good” images. Nevertheless, complex tasks may require more than 100 “good” images.
 - With EasySegment Supervised, the required number of images depends on the size and the number of elements to segment in each image. You can use as few as 10 elements per label. Nevertheless, complex tasks may require more than 100 elements per label.
 - With EasyLocate, the number of images depends on the number of objects/defects in the images. You can use as few as 10 objects per label. Nevertheless, complex tasks may require more than 100 objects per label.
 - 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 Unsupervised:
 - A single category for images of good samples.
 - As many categories as you want (including none) for images of defective samples.
- EasySegment Supervised:
 - You must annotate the pixels of the images with a ground truth segmentation
 - There must be at least one segmentation label in addition to the **Background** label.
- EasyLocate:
 - You must annotate the defects or the objects with a bounding box and a label.

Use the [EClassificationDataset](#) class to compile your labeled images.

3. Choose the deep learning tool that suits your needs.

All deep learning tools are child classes of the [EDeepLearningTool](#) class:

- EasyClassify: [EClassifier](#) class
- EasySegment: [EUnsupervisedSegmenter](#) and [ESupervisedSegmenter](#) classes.
- EasyLocate: [ELocator](#) 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 you can use to:

- Create datasets (load, label and segment images),
- Configure and visualize the data augmentation transformations,
- Train a deep learning tool,
- Analyze the performance of the tool,
- Apply the tool to new images.

**TIP**

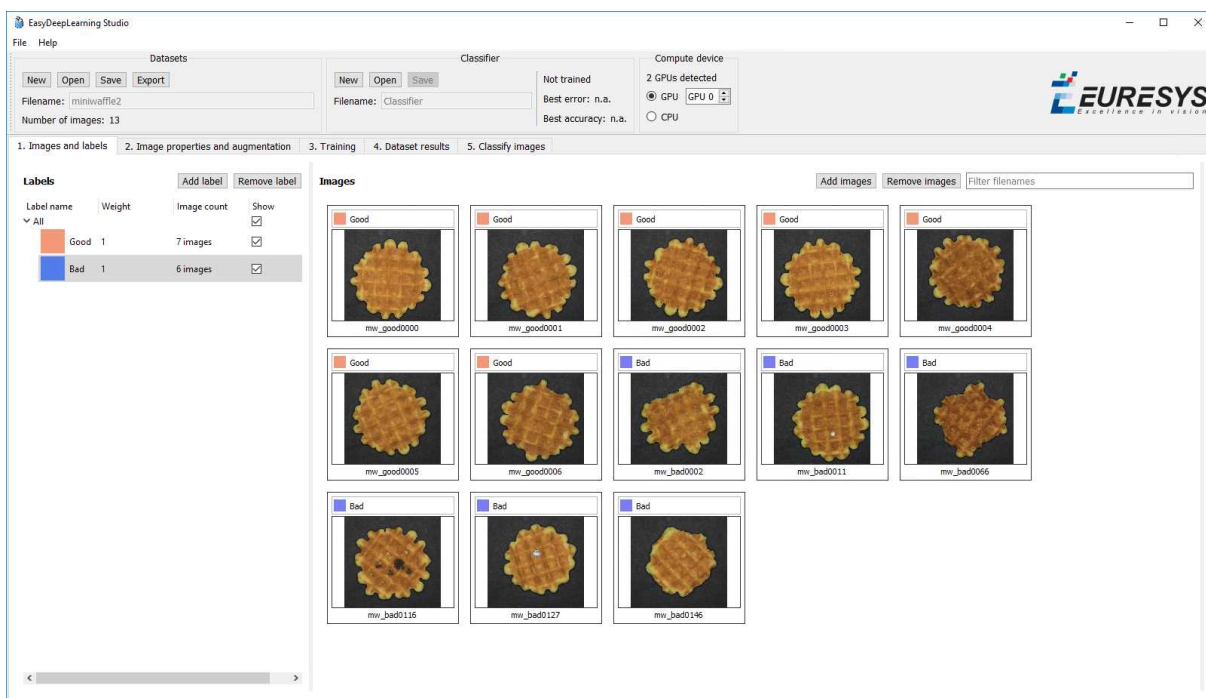
The Deep Learning Studio is available in the installation folder of Open eVision.

Resources and code snippets

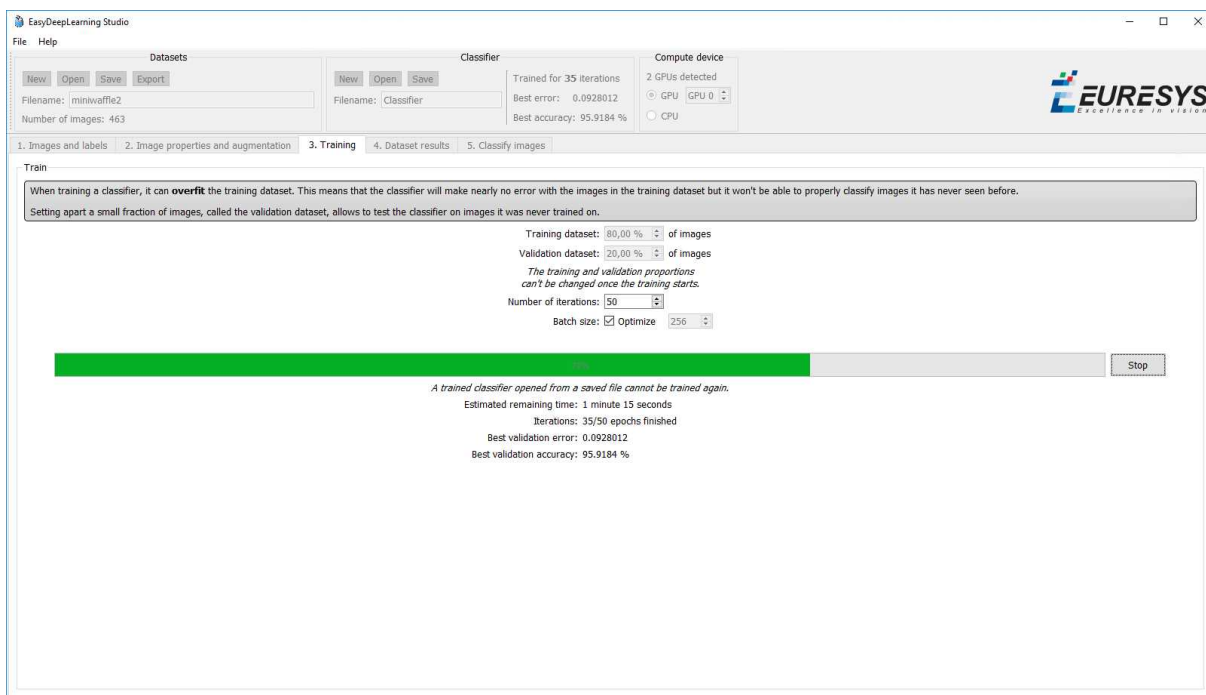
- The Deep Learning Additional Resources package, separate from the Open eVision installer, provide several sample datasets and deep learning tools trained with these datasets:
 - For EasyClassify: the **MiniWaffle** and **Stone Tiles** datasets
 - For EasySegment Unsupervised: the **Fabric** dataset
 - For EasySegment Supervised: the **Coffee** dataset
 - For EasyLocate: the **ElectronicComponentsBag** and **CeramicCapacitor** datasets
- 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

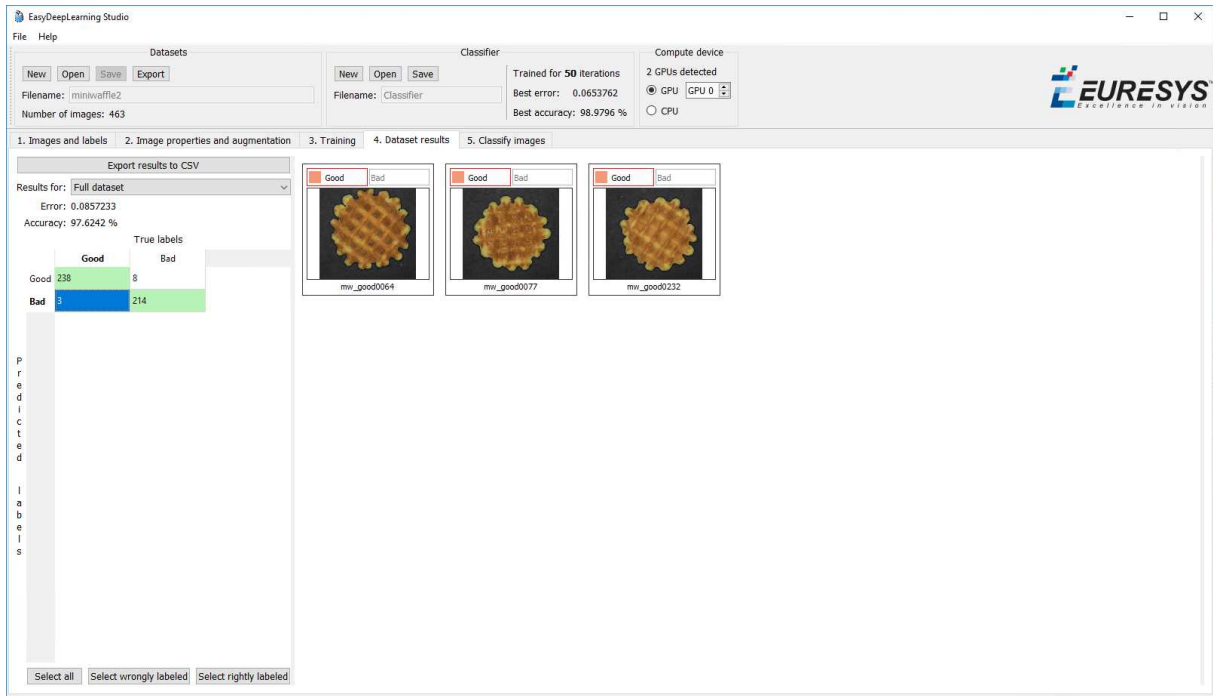
You can use Deep Learning Studio to perform steps 2 and 3 of the process described in section "Purpose and Workflow" on page 41.



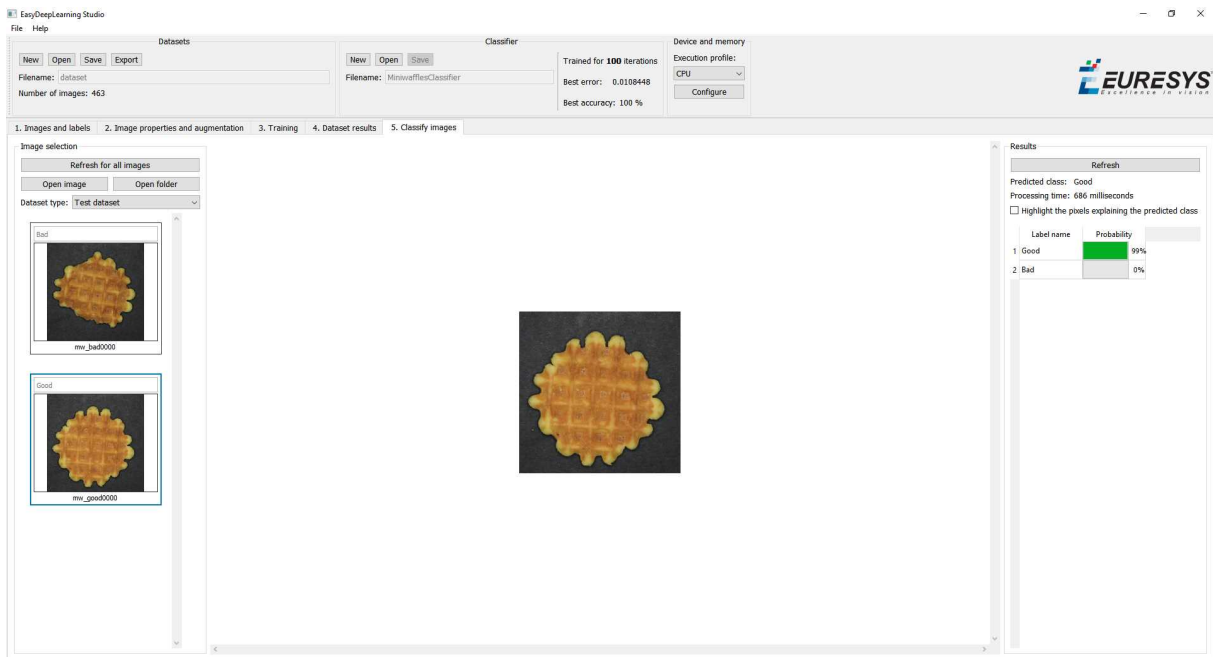
Manual labeling of images (step 2) and creating the dataset (steps 3a and 3b)



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 v11.1 (<https://developer.nvidia.com/cuda-toolkit>)
 - NVIDIA CUDA® Deep Neural Network library (cuDNN) v8.1 for CUDA 11.1 (<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\), a deep learning tool automatically finds what it needs.
 - Otherwise, copy the DLLs `cusolver64_*.dll`, `curand64_*.dll`, `cufft64_*.dll` and `cublas64_*.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 all the DLL files `cuda*8.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.

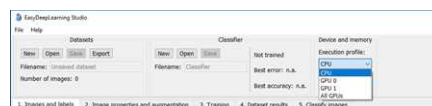


NOTE

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.

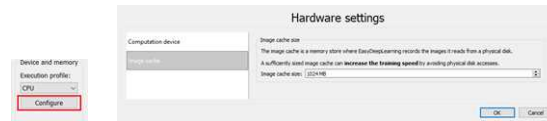
Image cache

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:

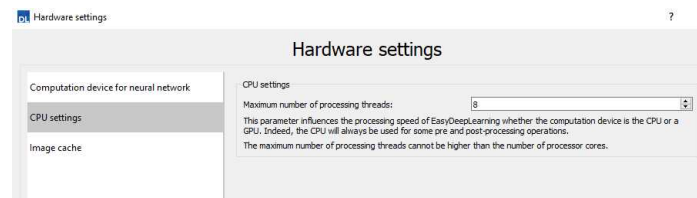
- In the API, use the `EDeepLearningTool::SetImageCacheSize` method.
- In Deep Learning Studio, click on the **Configure** button below the **Execution profile control** and select **Image cache** in the menu.



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 Images

Images and Labels

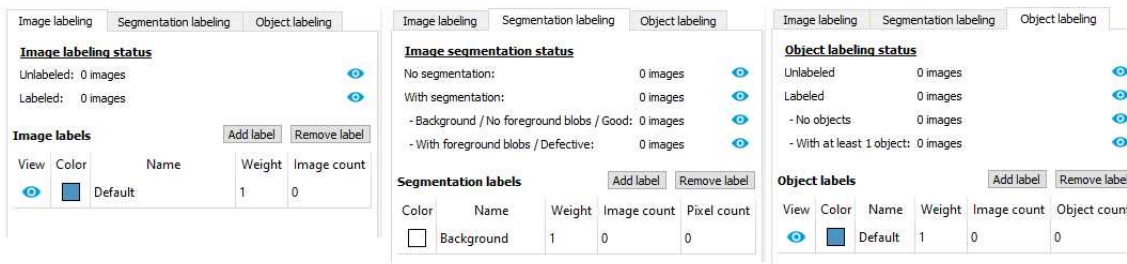
Images

- 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_BW-6`
 - `EImageType_C24`

Labels

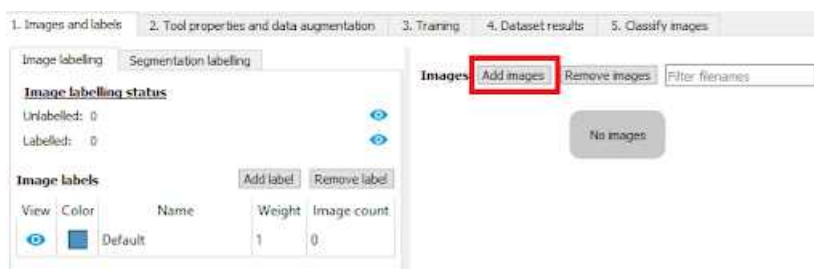
- There are 3 types of labels:
 - The *image labels* represent a characteristic of an image and its content. Use them to annotate images for EasyClassify or EasySegment Unsupervised.
 - The *segmentation labels* represent a characteristics of pixels. Use them to annotate image pixels for EasySegment Supervised.
 - The *object labels* represent a characteristic of a region of an image delimited by a bounding box. Use them to annotate images for EasyLocate.
- Images have the following labeling states:
 - *Labeled* or *Unlabeled* if the image is or is not associated with an image label.
 - Only labeled images are used to train an EasyClassify or an EasySegment Unsupervised tool.
 - In the API, use `EClassificationDataset::HasLabel(imageIndex)`.
 - *With* or *without segmentation* if the image has or has not a ground truth segmentation.
 - Only images with segmentation are used to train an EasySegment Supervised tool.
 - In the API, use `EClassificationDataset::HasSegmenation(imageIndex)`.
 - *With* or *without object labeling* if the image has or has not a ground truth object labeling.
 - Only images with object labeling are used to train an EasyLocate tool.
 - In the API, use `EClassificationDataset::HasObjectLabeling(imageIndex)`.
- The ground truth segmentation of an image has the following state:
 - *Background* when all the pixels of the image are associated with the **Background** segmentation label.
 - In defect detection applications, a background segmentation means that the image contains no defect.
 - *With foreground blobs* when the segmentation contains at least one pixel associated with a segmentation label different from **Background**.
 - In defect detection applications, a segmentation with foreground blobs means that the image contains defects.
 - In the API, use `EClassificationDataset::HasForegroundSegments(imageIndex)`.
- The ground truth object labeling of an image has the following state:
 - *No objects* when there is no object in the image.
 - In defect detection applications, an image with no object means that the image contains no defect.
 - *With objects* when there is at least one object in the image.
 - In defect detection applications, an image with objects means that the image contains defects.
 - In the API, use `EClassificationDataset::GetImageNumObjects(imageIndex)` to determine if the image has objects or not.

- In Deep Learning Studio, the icons (visible) and (hidden) represent the visibility state of the images with the corresponding state and/or image label in the image list.
 - Click on these icons to toggle the visibility state.

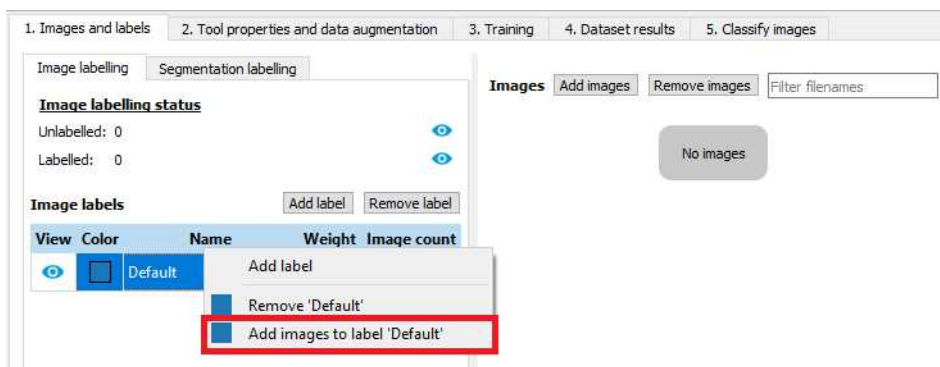


Adding Images

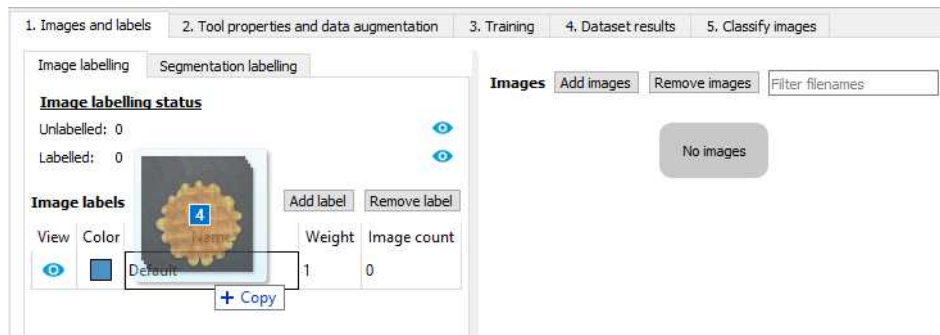
- In Deep Learning Studio, add image files (PNG, TIFF, JPEG, BMP and J2K types) to your datasets in one of the following ways:
 - Click on the **Add images** button to add images without any label nor segmentation.



- Right-click on an image label and click **Add images to label** to directly associate these images to the label.



- Drag and drop your files directly on an image label to directly associate these images to the label..



- Add a single image 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.
 - You can specify a label to immediately associate the image with the label. Otherwise, the image is unlabeled.
- Add several images with the `EClassificationDataset::AddImages(filter[, label])` method.

`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

The `EClassificationDataset` class automatically generates the set of labels from the labels of the images that you add to the dataset.



NOTE

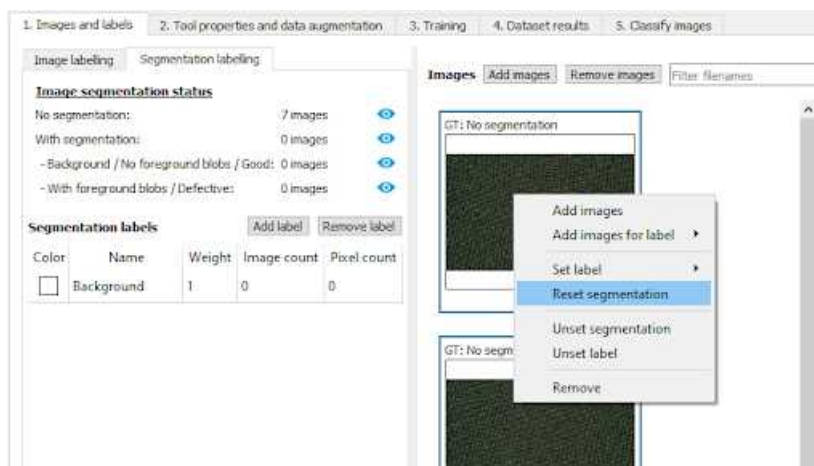
By default, all images are unlabeled and have no ground truth segmentation.

Editing the Segmentation of an Image

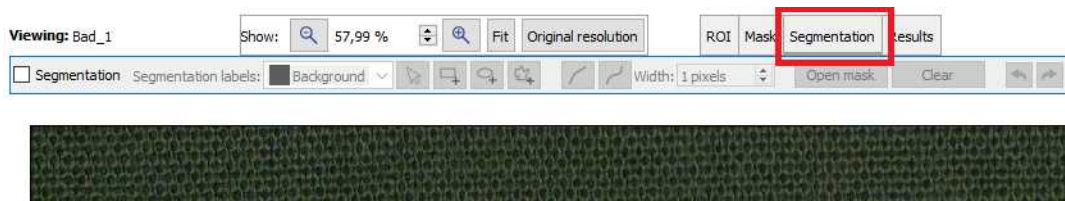
In Deep Learning Studio:

- To initialize or reset the segmentation of an image to all `Background` pixels:

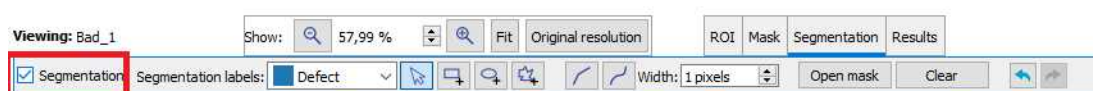
- a. Select one or more images in the image list.



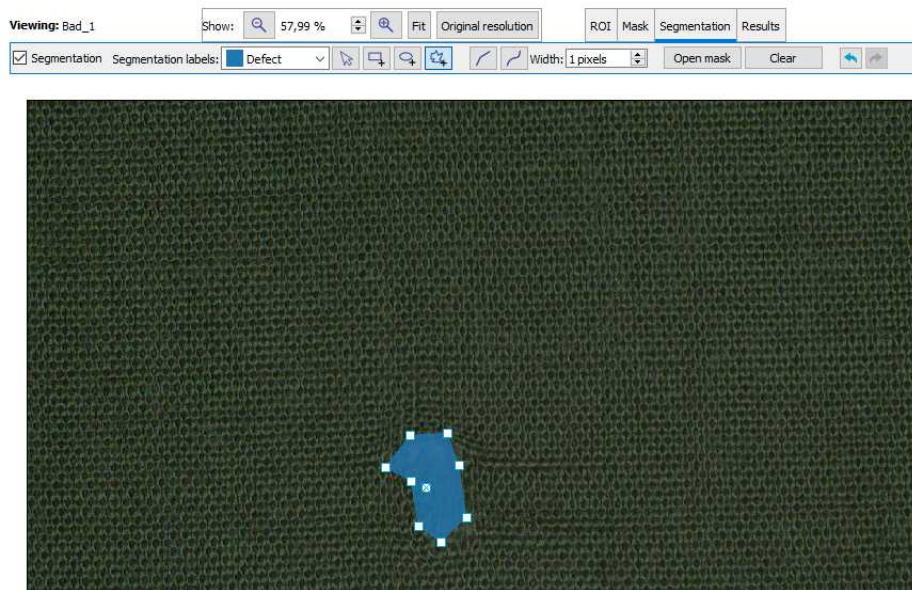
- b. Right click on the selection.
- c. Click on Reset segmentation.
- To edit the segmentation:
 - a. Double click on the image to open it in the image editor.
 - b. Click on the Segmentation button (ALT + S).



- c. To reset or unset the segmentation, uncheck the Segmentation checkbox (CTRL + S).



- d. Select a segmentation label, a drawing tool and enclose the segmentation.



- e. Change the segmentation label of a blob by right-clicking on it

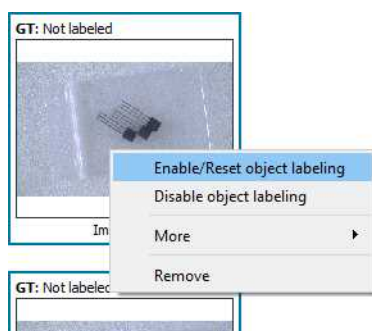


Editing the Objects of an Image

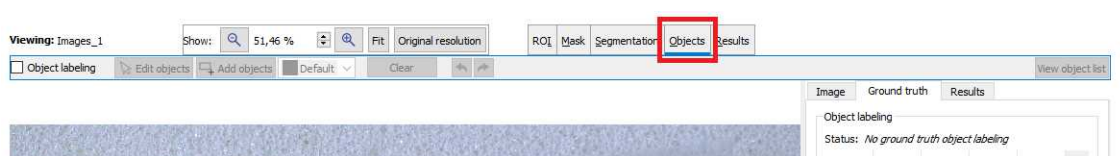
In Deep Learning Studio:

- To initialize or reset the object labeling of an image:

- a. Select one or more images in the image list.



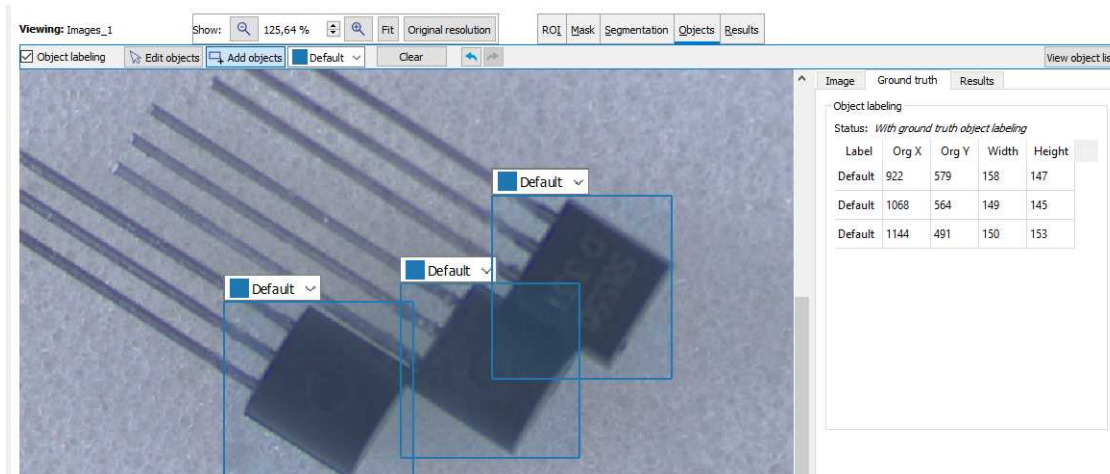
- b. Right click on the selection.
- c. Click on Reset object labeling.
- To edit the objects:
 - a. Double click on the image to open it in the image editor.
 - b. Click on the Objects button (ALT + 0).



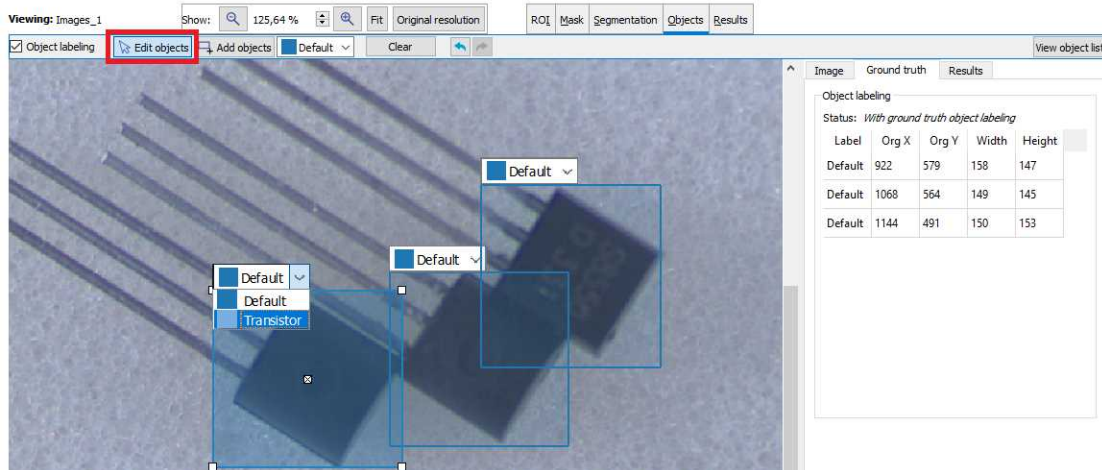
- c. To reset or unset the object labeling, uncheck the Object labeling checkbox (CTRL + L).



- d. Click on the Add objects button to add new objects with the label indicated next to the button.



- e. Click on the **Edit objects** button to modify the bounding box or the label of an object.



ROI and Mask

Setting a ROI

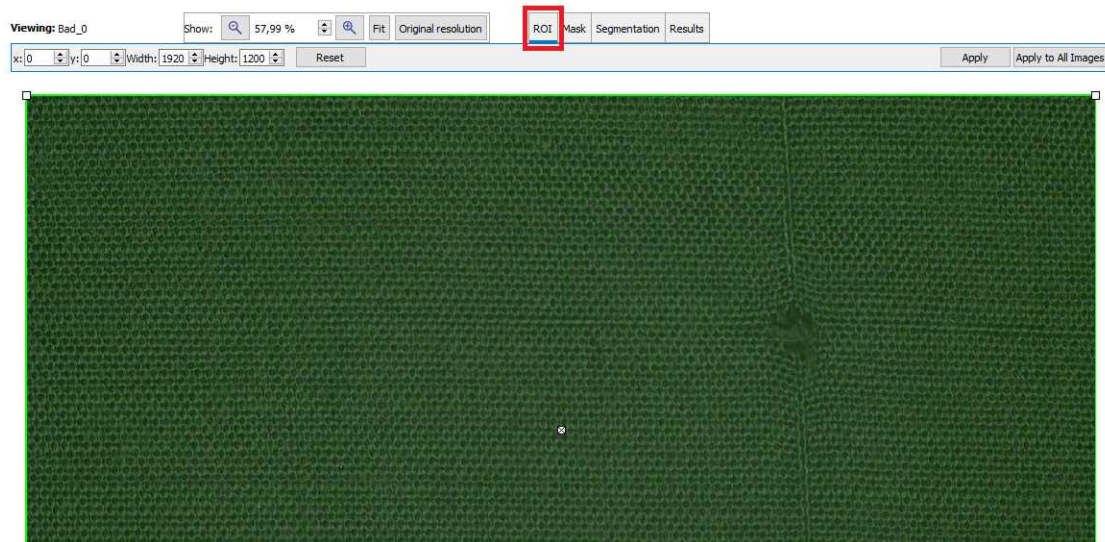
Use an ROI (region of interest) to crop an image or a whole dataset to a rectangular area aligned with the axis.

In the API:

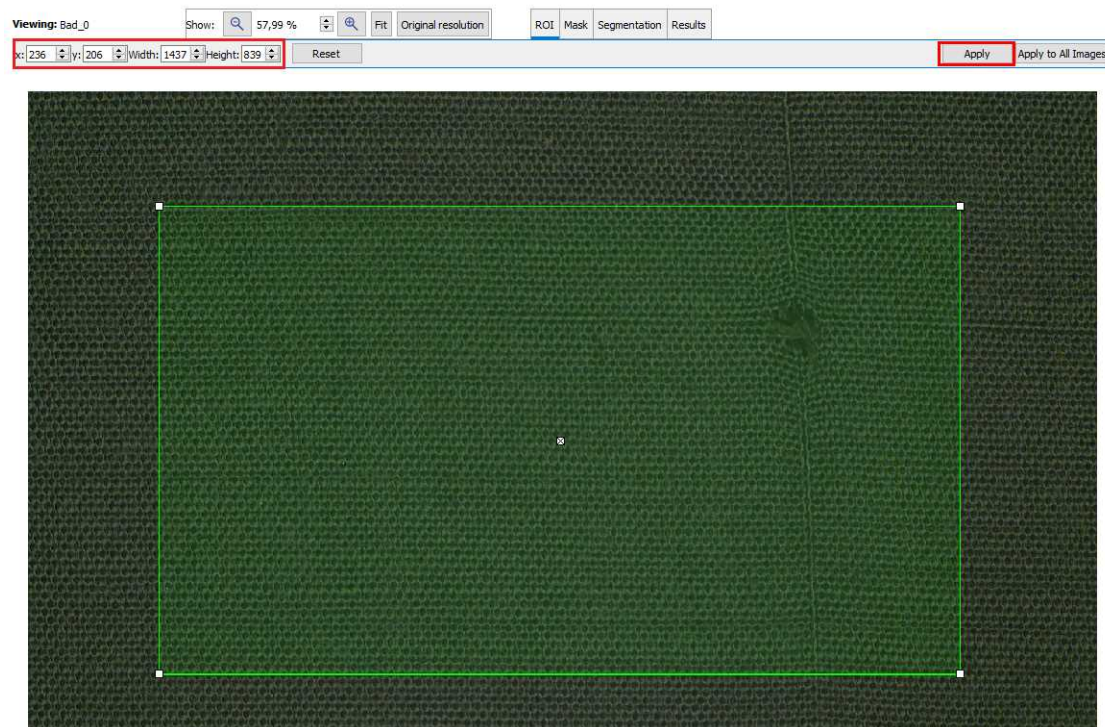
- To define an ROI for an image:
 - Specify the ROI when you add the image to the dataset.
 - Or use `EClassificationDataset::SetRegionOfInterest`.

In Deep Learning Studio:

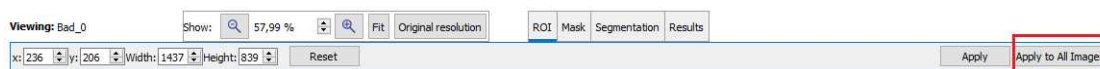
- To change the ROI:
 - a. Select an image from the dataset.
 - b. Click on the ROI button (ALT + I).



- c. Drag the ROI green box, or directly set the ROI origin (x and y), Width and Height.
- d. Click on the Apply button (CTRL + A).



- To set the same ROI for all the images of the dataset:
 - a. Set the ROI for one of the image.
 - b. Click on the Apply to All Images button (CTRL + SHIFT + A).



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:

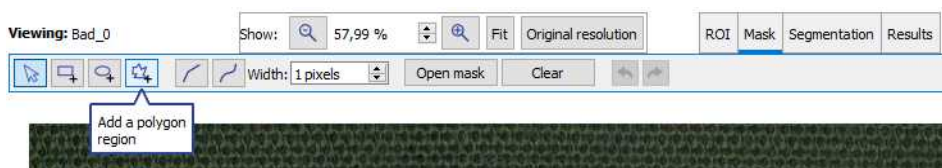
- To define a mask for an image:
 - Specify the mask when you add the image to the dataset.
 - Or use `EClassificationDataset::SetMask`.

In Deep Learning Studio:

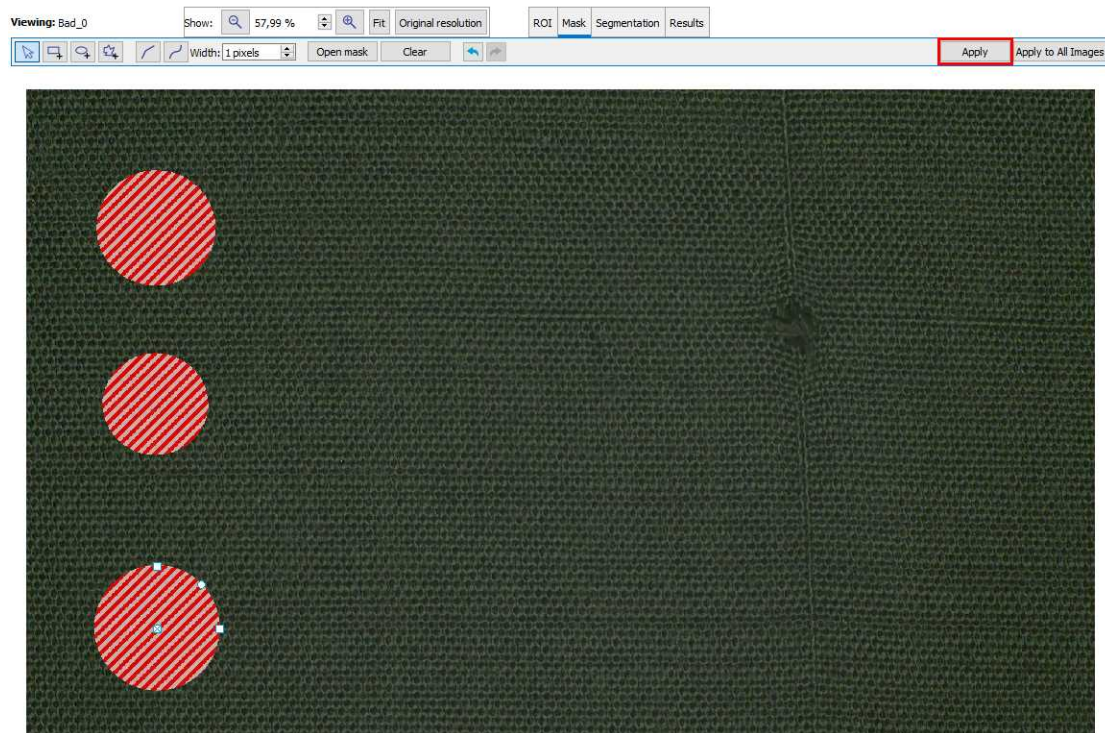
- To change the mask:
 - a. Select an image from the dataset.
 - b. Click on the Mask button (ALT + M).



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



- d. Draw the mask.
- e. Click on the **Apply** button (CTRL + A).



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. Specify the mask for one of the images.
 - b. Click on the **Apply to All Images** button (CTRL + SHIFT + A).



Managing the Dataset

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 [Test and results](#).

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 methods:
 - For EasyClassify and EasySegment Unsupervised: `EClassificationDataset::SplitDataset(trainingDataset, validationDataset, trainingProportion)`
 - For EasySegment Supervised: `EClassificationDataset::SplitDatasetForSegmentation(trainingDataset, validationDataset, trainingProportion)`
 - For Easylocate: `EClassificationDataset::SplitDatasetForLocator(trainingDataset, validationDataset, trainingProportion)`

Using Data Augmentation

Data augmentation performs random transformations on images given to a deep learning tool ([EClassifier](#), [EUnsupervisedSegmenter](#) or [ESupervisedSegmenter](#) object) during the training.

- Experiment different settings to choose the best parameters for your data augmentation.
- Configure data augmentation according to your problem. However, flips, shifts (20 - 40 px), brightness (5%), contrast (0.95 to 1.05) or salt and pepper noise (2%) can be useful on many datasets.
- 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.

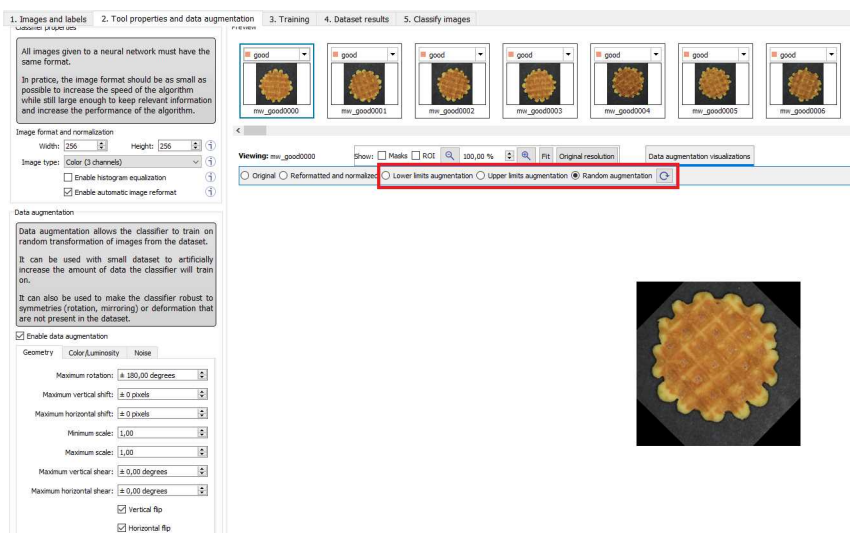


NOTE

With EasyLocate, we do not recommend to use rotation and shear data augmentation as it is not possible to compute the minimal bounding box surrounding the object after these geometric 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).

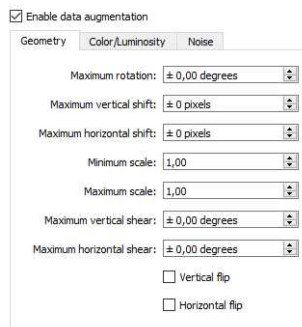


In the API:

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

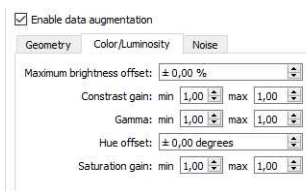
The possible transformations are:

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`)

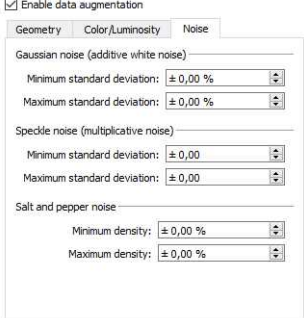
Color and luminosity transformations



- 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`).

Training a Deep Learning Tool

In the API, to train a deep learning tool, call the `EDeepLearningTool::Train(trainingDataset, validationDataset, numberOfIterations)` method.

- 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 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.
- *Multiple iterations:*
 - 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.
 - In successive calls to `EDeepLearningTool::Train`:
 - You can add images to the training and validation dataset to train the tool to recognize new instances of your problem.
 - We do not recommend that you remove images from the dataset as the tool might forget about these images during the new training phase.
- 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.

EasyClassify - Classifying Images

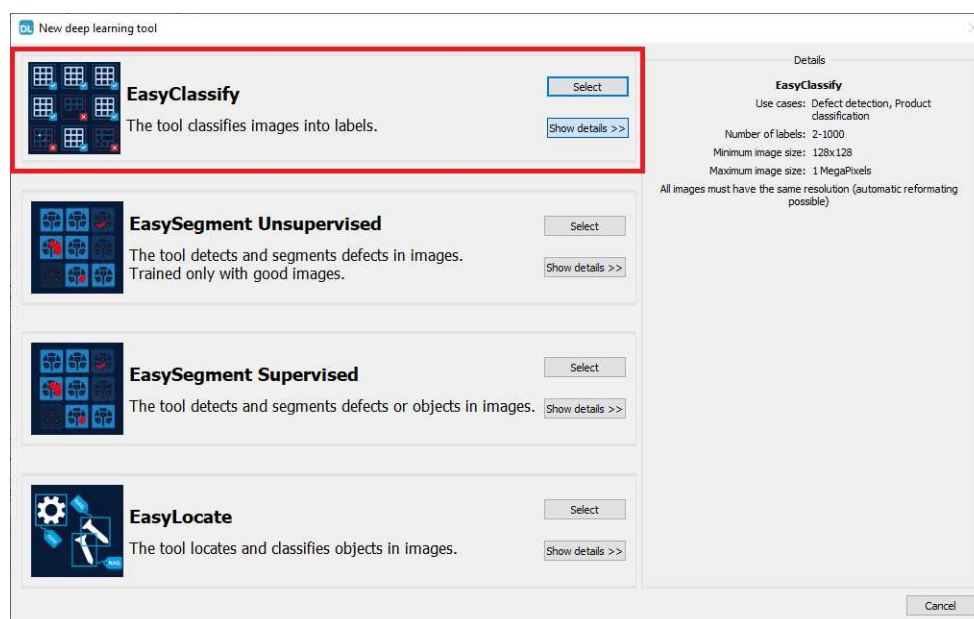
Tool and Images

EasyClassify is the deep learning classification library of Open eVision ([EClassifier](#) class).

Deep Learning Studio

To create a classification tool in Deep Learning Studio:

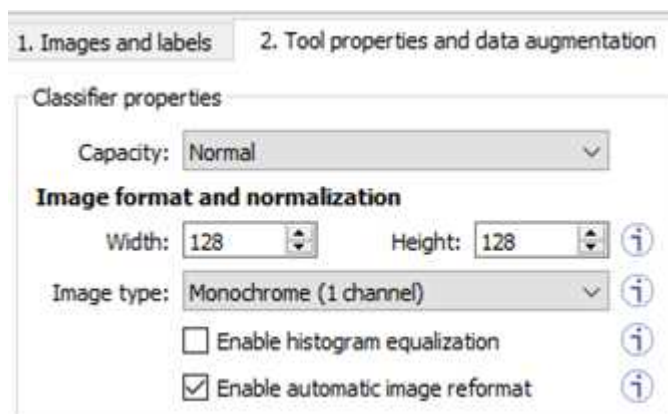
1. Start Deep Learning Studio.
2. Select EasyClassify in the New deep learning tool dialog.



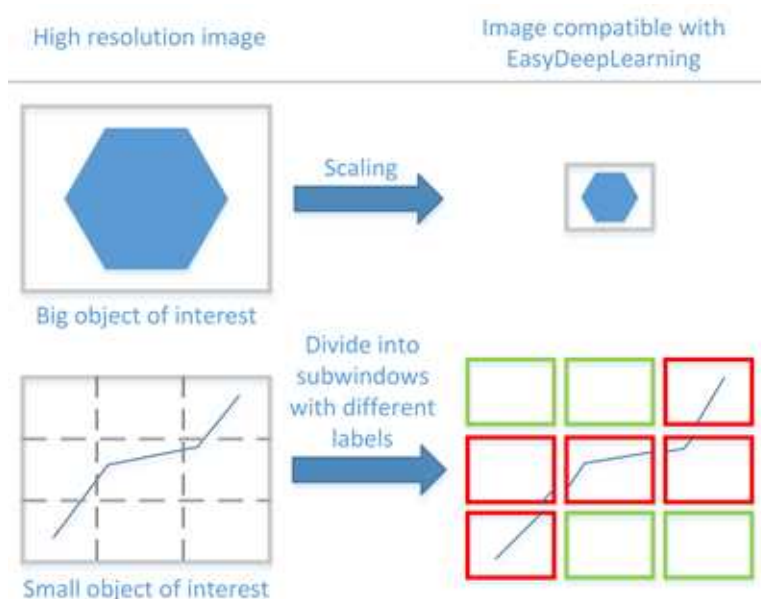
Input image format and normalization

- 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 Tool properties and data 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 of at least 128 x 128 for the normal and the large capacity or 64 x 64 for the small capacity and at most 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 predictions. This presents the additional advantage of localizing the "object of interest" inside the original image.



- The **Capacity** of the neural network (default: **Normal**) represents the quantity of information that it is capable of learning.
 - The small network is much smaller in memory and faster at inference.
 - The large network can handle more complex datasets. It is also better for datasets with a lot of noise.

In the API:

- The capacity values are defined by the enumerate type `EClassifierCapacity`.
- Use `EClassifier::SetCapacity` to set the capacity of your tool.

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.

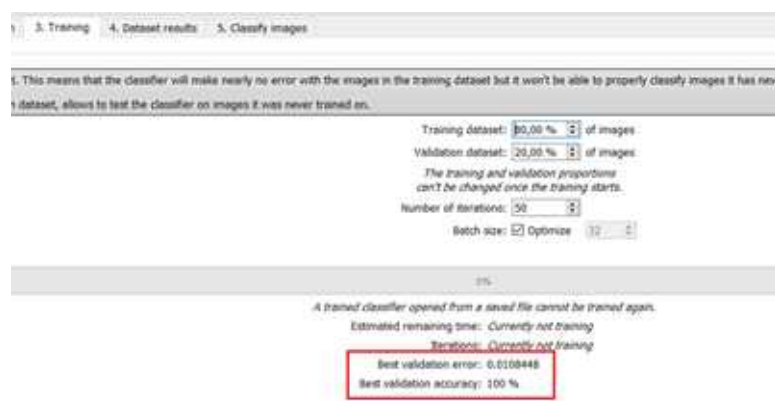
Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 65.

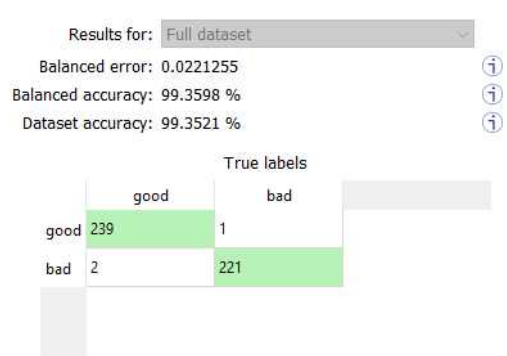
Validating the Results

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 `EDeepLearningTool::GetNumTrainedIterations()-1`.
 - Call `EDeepLearningTool::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

Classify images

- In Deep Learning Studio, open the `Test` and `results` 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.
 - The heat map is colored, and the important parts are displayed in red.

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:

$$\text{TrainingMemoryCPU} = 0.000453 \times W \times H \times B - 292 \text{ (MB)}$$

$$\text{TrainingMemoryGPU} = 0.000440 \times W \times H \times B + 25 \text{ (MB)}$$

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

$$\text{ClassificationMemoryCPU} = 0.000232 \times W \times H \times B - 97 \text{ (MB)}$$

$$\text{ClassificationMemoryGPU} = 0.000226 \times W \times H \times B + 13 \text{ (MB)}$$

- 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 slower due to the memory allocations.

Benchmarks for EasyClassify

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Capacity small

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	19	2.7 ms	10.2 ms	-
	4	29	0.88 ms	-	54
	16	67	0.6 ms	-	134
	64	232	0.39 ms	-	463
256 × 256	1	30	3.3 ms	45.3 ms	-
	4	74	1.99 ms	-	144
	16	247	1.35 ms	-	497
	64	959	1.14 ms	-	1924
512 × 512	1	74	7.9 ms	199 ms	-
	4	260	4.56 ms	-	519
	16	998	5.89 ms	-	2001
	64	3908	4.74 ms	-	7936

Capacity normal

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	92	4.4 ms	13 ms	-
	4	102	1.29 ms	-	166
	16	141	0.66 ms	-	320
	64	277	0.41 ms	-	608
256 × 256	1	103	5.3 ms	50 ms	-
	4	145	2.33 ms	-	253
	16	315	1.3 ms	-	1219
	64	1042	1.1 ms	-	2144

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
512 × 512	1	155	17.8 ms	199 ms	-
	4	332	5.8 ms	-	1110
	16	1069	4.3 ms	-	2605
	64	4122	4.22 ms	-	8533

Capacity large

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	32	2.4 ms	17.1 ms	-
	4	50	1.2 ms	-	100
	16	131	0.9 ms	-	255
	64	421	0.6 ms	-	848
256 × 256	1	54	4.2 ms	69 ms	-
	4	137	2.7 ms	-	275
	16	473	1.8 ms	-	990
	64	1830	1.3 ms	-	3660
512 × 512	1	145	15.9 ms	324 ms	-
	4	502	8.6 ms	-	1004
	16	1932	7.2 ms	-	3961
	64	7690	6.2 ms	-	15380

EasySegment - Detecting and Segmenting Defects

Unsupervised vs Supervised Modes

EasySegment is the deep learning segmentation library of Open eVision.

It contains 2 different modes:

- The *unsupervised* mode:
 - The tool is trained only with good images ([EUnsupervisedSegmenter](#) class).
 - This mode does not require a ground truth segmentation and the creation of the dataset is thus much quicker than for the supervised mode.
 - This mode can detect unexpected defects while the supervised mode is only capable of detecting defects similar to those in the dataset.
- The *supervised* mode:
 - The tool is trained using the ground truth segmentation defined for the images ([ESupervisedSegmenter](#) class).
 - This mode can detect and segment more types of defects with better accuracy than the unsupervised mode. It directly builds a model for the defects while the unsupervised mode builds a model of the good images and tries to detect variations from this model.
 - You can also use this mode to segment other types of objects than defects.

EasySegment Unsupervised

Tool and Configuration

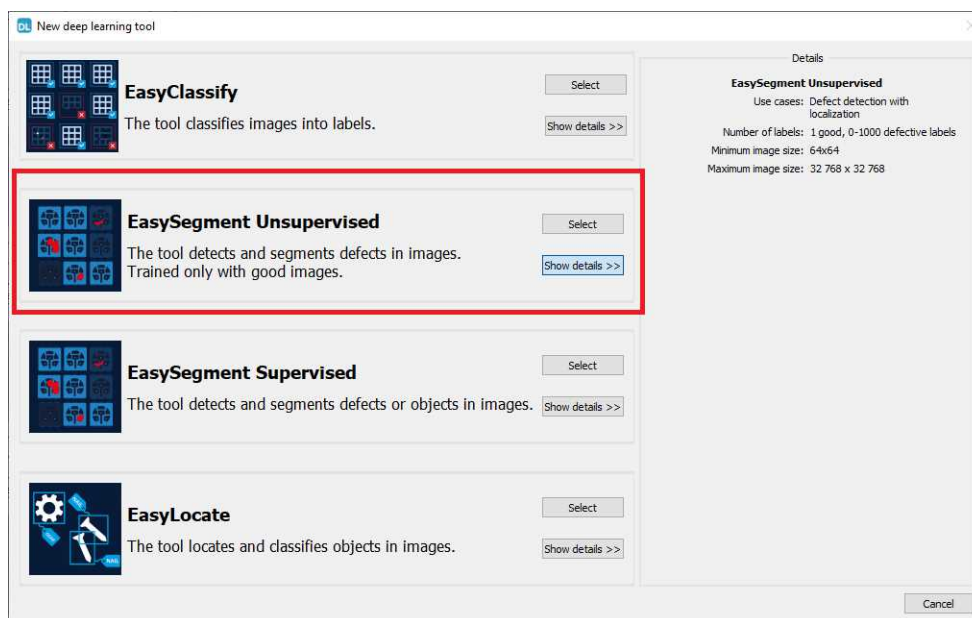
EasySegment Unsupervised is the deep learning tool part of the EasySegment segmentation library of Open eVision. It detects and segments defects in images.

This tool trains in an unsupervised way. This means that it is trained only with good images. So it does not require any ground truth segmentation of the defects.

Deep Learning Studio

To create an unsupervised segmentation tool in Deep Learning Studio:

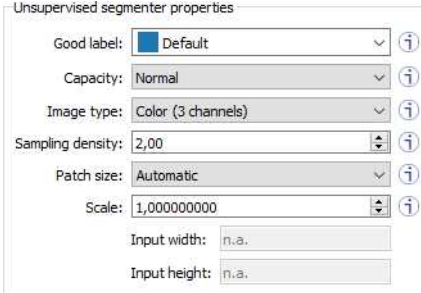
1. Start Deep Learning Studio.
2. Select EasySegment Unsupervised in the New deep learning tool dialog.



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 6 parameters:

1. The **Good label** is the name of the class that contains the good images.
2. 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`.
- `EUnsupervisedSegmenter::Capacity` sets the capacity of the tool.

3. The **Image type** (default: *Monochrome* (↖ channel)):

In the API:

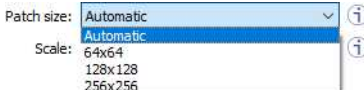
- To use monochrome (grayscale, 1 channel) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `true`.
- To use color (3 channels) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `false`.

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:
 $100 - 100 / \text{SamplingDensity} (\%)$
- In practice, the stride between 2 consecutive patches is:
 $\text{PatchSize} / \text{SampleDensity} (\text{pixels})$

5. The **Patch size** (`EUnsupervisedSegmenter::PatchSize`) is the size of the patches processed by the neural network.

- By default, the patch size is determined automatically from the images in the training dataset.
- You can also select the resolution of the patch size from the drop down list.



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

In Deep Learning Studio:

- If the dataset contains images with different resolutions, the `Input width` and the `Input height` indicate the range of the resolutions with the given scale.
- If all the images in the dataset have the same resolution, set either the `Input width` or the `Input height` to change the scale.

Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 65.

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 require 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).

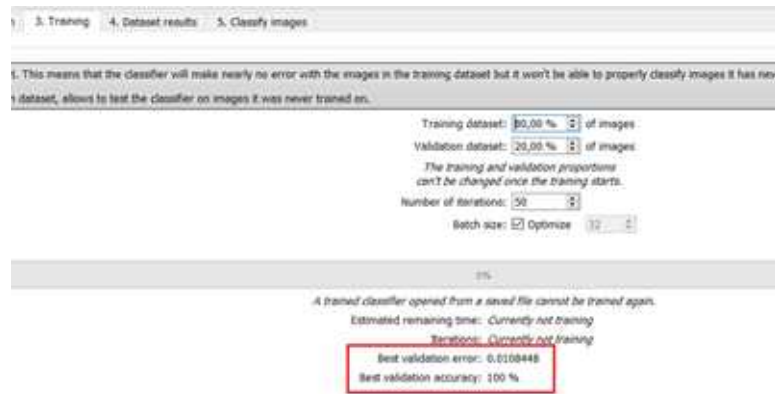
The unsupervised segmentation tool computes a score for each image (see `EUnsupervisedSegmenterResult::ClassificationScore`). The label of a result is obtained by thresholding this score with the segmenter classification threshold (`EUnsupervisedSegmenter::ClassificationThreshold`). So, the supervised 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 positive 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 (`EDeepLearningDefectDetectionMetrics::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 (`EDeepLearningDefectDetectionMetrics::GetAccuracy`)
 - The Area under ROC curve (`EDeepLearningDefectDetectionMetrics::GetAreaUnderROCCurve`)
 - The ROC point corresponding to the classification threshold (`EDeepLearningDefectDetectionMetrics::GetROCPoint`)

Applying the Tool to New Images

In Deep Learning Studio:

- Open the **Test** and **results** 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).
- `EUnsupervisedSegmenterResult::Draw` draws the segmentation mask.

Benchmarks for EasySegment Unsupervised

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Image size

- The inference times are reported for 1024×1024 RGB images with all other settings at their default values.
- The inference times increase linearly with the width and height of the image. The inference times of a 512×512 image will be 25% of the time reported below.

Capacity small

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	2	641 ms	3082 ms	-
	4	4	299 ms	-	7
	16	13	202 ms	-	31
	64	56	199 ms	-	134

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	18	389 ms	3234 ms	-
	4	28	215 ms	-	46
	16	66	174 ms	-	132
	64	223	172 ms	-	472
256 × 256	1	71	227 ms	2519 ms	-
	4	109	170 ms	-	226
	16	256	159 ms	-	526
	64	875	166 ms	-	1777

Capacity normal

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	6	835 ms	6208 ms	-
	4	10	341 ms	-	18
	16	30	215 ms	-	67
	64	104	213 ms	-	234
128 × 128	1	65	577 ms	6245 ms	-
	4	82	261 ms	-	172
	16	152	192 ms	-	312
	64	447	189 ms	-	921
256 × 256	1	182	309 ms	5111 ms	-
	4	250	194 ms	-	620
	16	596	171 ms	-	1220
	64	1756	184 ms	-	3710

Capacity large

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	18	1188 ms	12880 ms	-
	4	25	439 ms	-	115
	16	61	245 ms	-	179
	64	228	251 ms	-	473
128 × 128	1	170	911 ms	13344 ms	-
	4	203	359 ms	-	527
	16	409	240 ms	-	846
	64	1008	219 ms	-	2214
256 × 256	1	464	853 ms	11642 ms	-
	4	594	293 ms	-	1103
	16	1613	227 ms	-	3226
	64	3968	215 ms	-	8044

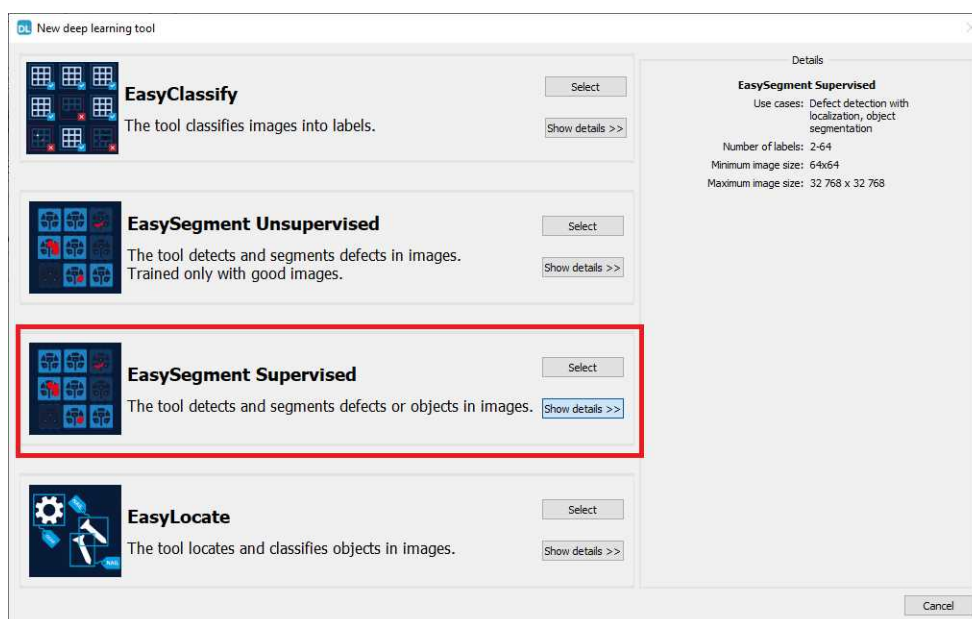
EasySegment Supervised

Tool and Configuration

Deep Learning Studio

To create an supervised segmentation tool in Deep Learning Studio:

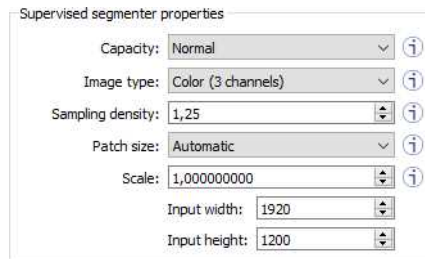
1. Start Deep Learning Studio.
2. Select EasySegment Supervised in the New deep learning tool dialog.



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 supervised 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 `ESupervisedSegmenterCapacity`.
- `ESupervisedSegmenter::Capacity` sets the capacity of the tool.

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

In the API:

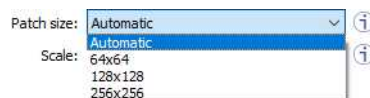
- To use monochrome (grayscale, 1 channel) images, set `ESupervisedSegmenter::ForceGrayscale` to `true`.
- To use color (3 channels) images, set `EUnsupervisedSegmenter::ForceGrayscale` to `false`.

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

- It indicates how much overlap there is between the image patches:
 $100 - 100 / \text{SamplingDensity} (\%)$
- In practice, the stride between 2 consecutive patches is:
 $\text{PatchSize} / \text{SampleDensity} (\text{pixels})$

4. The **Patch size** (`ESupervisedSegmenter::PatchSize`) is the size of the patches processed by the neural network.

- By default, the patch size is determined automatically from the images in the training dataset.
- You can also select the resolution of the patch size from the drop down list.



5. Use the **Scale** (`ESupervisedSegmenter::Scale`) to automatically resize your images to a lower resolution and accelerate the processing.

In Deep Learning Studio:

- If the dataset contains images with different resolutions, the **Input width** and the **Input height** indicate the range of the resolutions with the given scale.
- If all the images in the dataset have the same resolution, set either the **Input width** or the **Input height** to change the scale.

Training

To train your tool, see "Training a Deep Learning Tool" on page 65.

Using the Supervised Segmenter

To get the result, a supervised segmenter follows these steps:

1. For each pixel, the supervised segmenter tool computes the probabilities that it belongs to each of the segmentation labels.
2. From these probabilities, it extracts a set of potential foreground blobs (groups of contiguous pixels for which the highest probability corresponds to the same foreground segmentation label).
3. For each one of these potential foreground blobs:
 - It computes a score.
 - It removes, from the predicted segmentation map, the blobs with a score that is below or equal to the threshold of the supervised segmenter tool.
4. The score of an image is the maximum among the scores of the potential foreground blobs.

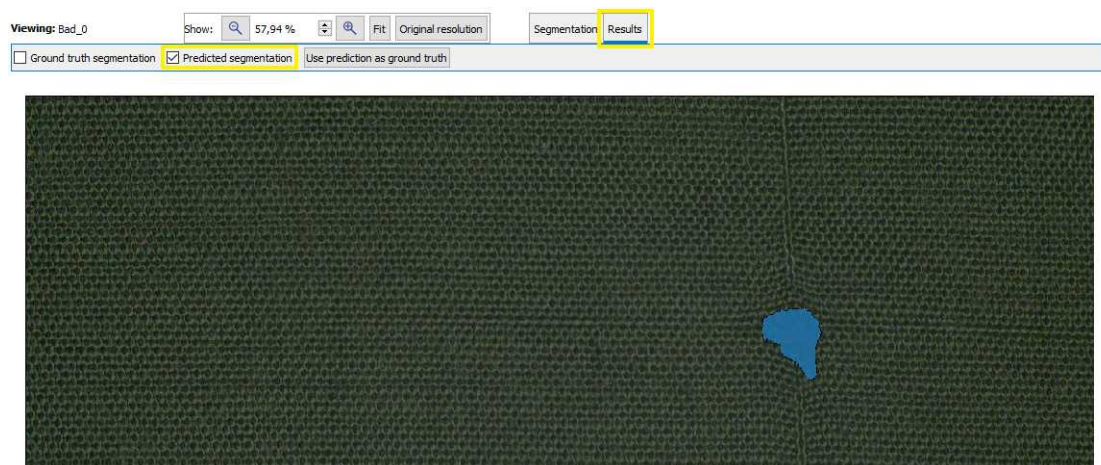


NOTE

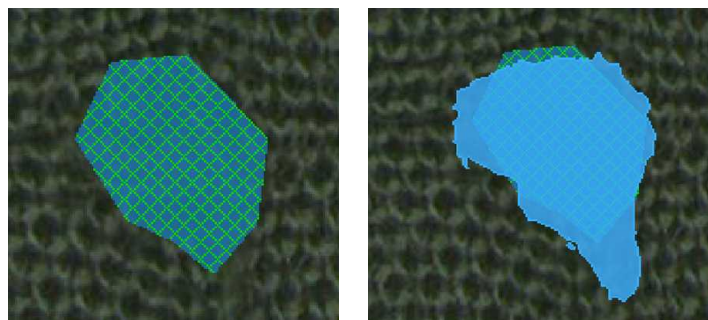
In the context of defect detection, an image is considered to be without defect when its score is below or equal to the threshold of the supervised segmenter tool.

In Deep Learning Studio:

- To apply the segmenter to new images, add these images to the dataset or use the Test and results tab.
- To visualize the segmentation of an image, check the Predicted segmentation option (CTRL + P) in the Result menu (ALT + R) of the image viewer.



- If the image has a ground truth, check the Ground truth segmentation option (CTRL + G) to display it. It appears with a green pattern drawn over it.

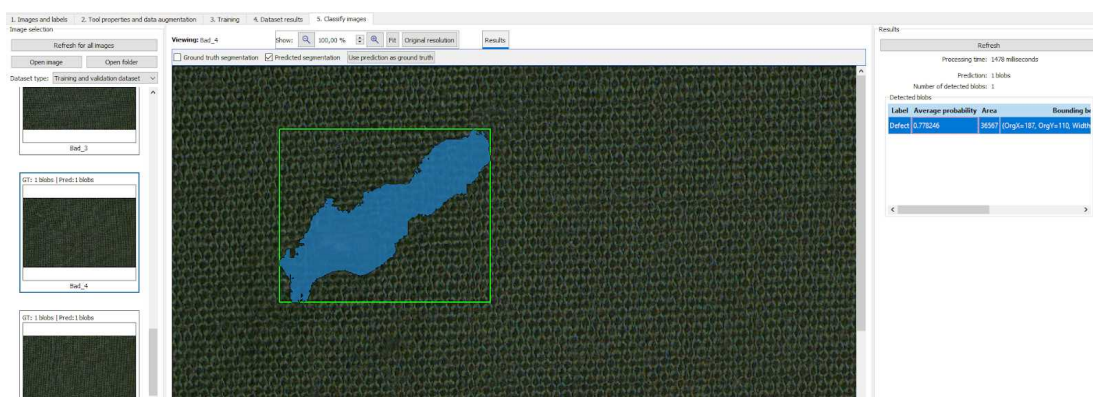


Ground truth (left) and prediction on top of ground truth (right)

- To accept the whole predicted segmentation as ground truth, click on the Use prediction as ground truth button (CTRL + U).
- To accept a single predicted blob as ground truth, right click on the blob and select Accept into ground truth in the menu.



- A list of blobs with various characteristics is available in the Classify tab.



In the API:

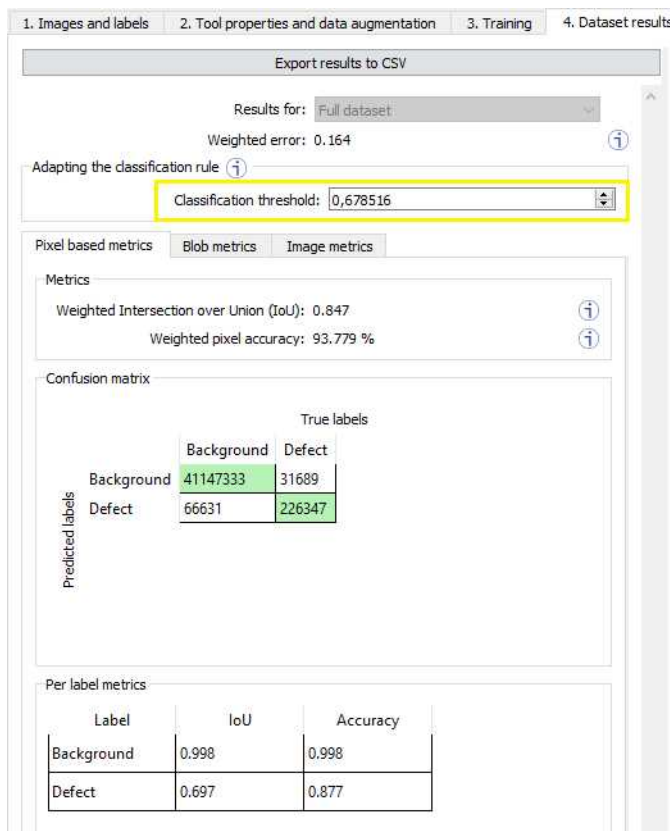
- To apply the supervised segmenter to an image use `ESupervisedSegmenter::Apply`. This method returns a `ESupervisedSegmenterResult` object.
 - Use `ESupervisedSegmenterResult::GetProbabilityMap` to retrieve the probability map for a given label. The probability map pixels contain the index of the predicted label.
 - Use `ESupervisedSegmenterResult::Draw` to draw the segmentation with the segmentation label colors of the dataset used for training.
 - Use `ESupervisedSegmenterResult::GetBlobs` to retrieve the filtered list of blobs.
 - Use `ESupervisedSegmenterResult::Score` to retrieve the score of an image.
 - Use `ESupervisedSegmenterResult::GetRegionForLabel` to obtain an `ERegion` object containing the pixels of the specified label.

Evaluating the Results

There are 3 types of metrics for the supervised segmentation tool:

- The *pixel-based metrics* that quantify the performance of the tool at the pixel level.
- The *blob-based metrics* that quantify the performance of the tool at the blob level. A blob is a contiguous region of pixels that have the same foreground segmentation label. By definition, there is no background blob.
- The *image-based metrics* that quantify the performance of the tool at the image level. These metrics are related to the capacity of the tool to correctly detect background images (images with no blobs) and foreground images (images with blobs).

- In Deep Learning Studio:
 - The metrics are available in the **Dataset results** tab.
 - Most metrics depends on the value of the **Classification threshold**.



1. Images and labels | 2. Tool properties and data augmentation | 3. Training | 4. Dataset results

Export results to CSV

Results for: Full dataset

Weighted error: 0.164

Adapting the classification rule

Classification threshold: 0,678516

Pixel based metrics | Blob metrics | Image metrics

Metrics

Weighted Intersection over Union (IoU): 0.847

Weighted pixel accuracy: 93.779 %

Confusion matrix

		True labels	
		Background	Defect
Predicted labels	Background	41147333	31689
	Defect	66631	226347

Per label metrics

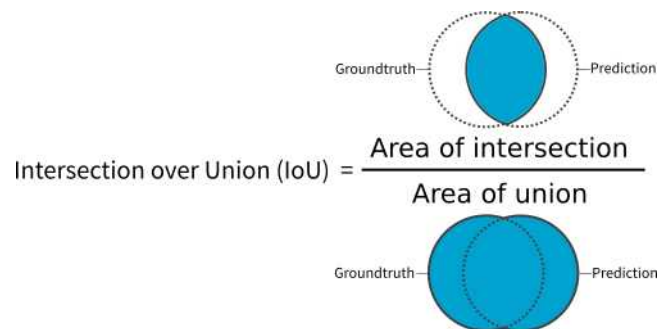
Label	IoU	Accuracy
Background	0.998	0.998
Defect	0.697	0.877

- In the API:
 - The metrics are represented by an `ESupervisedSegmenterMetrics` object.

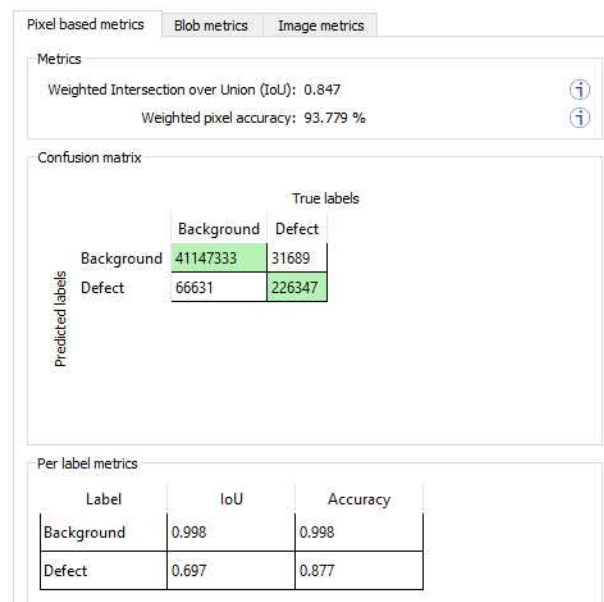
The pixel-based metrics

- The pixel-based metrics are:
 - The weighted *Intersection over Union* (IoU) that is the weighted average of the IoU over all the labels (see per-label metrics).
 - The weighted *pixel accuracy* that is the weighted average of the accuracy over all the labels (see per-label metrics).
 - The *pixel confusion matrix* that shows the number of pixels from a given label that are predicted to belong to another label.

- The per label metrics are:
 - The *Intersection over Union* (IoU) that is the ratio of the intersection between the ground truth and the prediction for the label to the union of the ground truth and prediction for the label.



- The *accuracy* that is the proportion of the pixels of the label that are correctly predicted.



The blob-based metrics

- The metrics related to the correct prediction of blobs are:
 - The *recall* that is the ratio of the correctly predicted blobs to the total number of ground truth blobs.
 - The *precision* that is the ratio of correctly predicted blobs to the total number of predicted blobs.
 - The *F1-Score* that is the harmonic mean of the *recall* and the *precision*.
 - The *average precision* that is the average of the *precision* for different threshold weighted by the *recall* values.
 - The *best achievable F1-Score* that is the maximum *F1-Score* achievable by selecting an appropriate threshold.

- The confusion matrix:
 - It shows the number of blobs of a given true label that are predicted to be of the corresponding predicted label.
 - The image list of the **Dataset results** tab only shows the images containing the blobs corresponding to the selected cells of the matrix.
 - Each matrix element shows the number of ground truth blob and the number of corresponding predicted blobs separated by a "/" as a ground truth blob can correspond to one or more blobs in the prediction and inversely.
A dash (-) indicates that blobs are not defined for this category.
 - For example, in the screenshot below, there are:
 - 3 predicted blobs of the label **Defect** that correspond to the **Background**.
 - 1 ground truth blob of the label **Defect** that does not correspond to any predicted blob.
 - 12 predicted blobs of the label **Defect** that correspond to 12 ground truth blob of the same label.
- The metrics for each individual foreground label are:
 - *Recall*
 - *Precision*
 - *F1-Score*



The image-based metrics

- The metrics related to the correct detection of the class of the images (background / foreground or good / defective in the context of defect detection):
 - The *image detection accuracy* that is the proportion of image correctly predicted to have foreground blobs or not.
 - The *foreground image detection rate* that is the proportion of correctly predicted images with foreground blobs.
 - The *background image detection rate* that is the proportion of correctly predicted images with no foreground blobs.

- The *image detection AUC* (Area under the ROC Curve, see ROC Curve below).
 - The *best achievable image detection accuracy* that is the maximum *image detection accuracy* obtained by changing the threshold.
- The confusion matrix:
 - It shows the number of images of a given true label that are predicted to be of the corresponding predicted label.
 - The image list of the [Dataset results](#) tab only shows the images corresponding to the selected cells of the matrix.
- The 2 available graphics are:
 - The *ROC curve* that plots the true positive rate (*foreground image detection rate*) versus the false positive rate (1 minus the *background image detection rate*) for various threshold values.
Click on a point on the plot to set the threshold at the corresponding value.



- The *score histogram* that plots:
 - In green: the cumulative histogram of the scores of the background images.
 - In orange: the cumulative histogram of the scores of the foreground images.
 - The blue line corresponds to the current value of the threshold.



Benchmarks for EasySegment Supervised

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Image size

- The inference times are reported for 1024 × 1024 RGB images with all other settings at their default values.
- The inference times increase linearly with the width and height of the image. The inference times of a 512 × 512 image will be 25% of the time reported below.

Capacity small

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	16	545 ms	10090 ms	-
	4	29	187 ms	-	151
	16	87	123 ms	-	263
	64	335	122 ms	-	573

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	27	261 ms	14085 ms	-
	4	83	133 ms	-	306
	16	366	102 ms	-	757
	64	1441	114 ms	-	3139
256 × 256	1	148	544 ms	17606 ms	-
	4	369	151 ms	-	1185
	16	1497	111 ms	-	3091
	64	7050	148 ms	-	12564

Capacity normal

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	18	789 ms	24458 ms	-
	4	45	271 ms	-	229
	16	213	160 ms	-	450
	64	828	171 ms	-	1753
128 × 128	1	110	1952 ms	34032 ms	-
	4	216	346 ms	-	878
	16	884	188 ms	-	1864
	64	3446	177 ms	-	6124
256 × 256	1	301	327 ms	42868 ms	-
	4	864	280 ms	-	1822
	16	3027	156 ms	-	6632
	64	14048	-	-	25796

Capacity large

Patch size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
64 × 64	1	91	7443 ms	58658 ms	-
	4	140	1076 ms	-	727
	16	440	437 ms	-	1201
	64	1666	346 ms	-	3332
128 × 128	1	226	1043 ms	83118 ms	-
	4	562	820 ms	-	996
	16	1822	289 ms	-	4222
	64	6925	-	-	16156
256 × 256	1	877	1153 ms	106446 ms	-
	4	2249	567 ms	-	3887
	16	7417	-	-	15603
	64	28225	-	-	50306

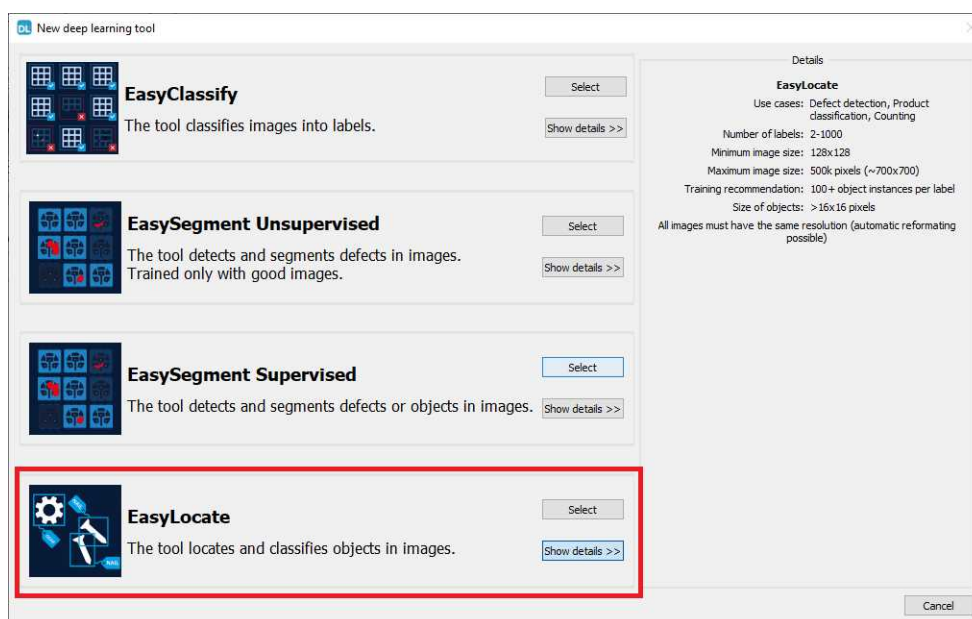
EasyLocate - Locating Objects and Defects

Tool and Configuration

Deep Learning Studio

To create an EasyLocate tool in Deep Learning Studio:

1. Start Deep Learning Studio.
2. Select EasyLocate 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 (main parameters)

The EasyLocate tool has 3 main 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 `ELocatorCapacity`.
- `ELocator::Capacity` sets the capacity of the tool.

2. The **Image type** (default: *Monochrome* (1 channel) if the dataset contains only grayscale images, otherwise *color* (3 channels)):

In the API:

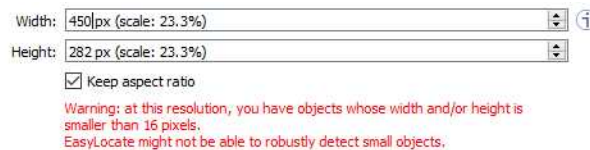
- To use monochrome (grayscale, 1 channel) images, set `ELocator::Channels` to 1.
- To use color (3 channels) images, set `ELocator::Channels` to 3.

3. The size of the images (**Width** and **Height**). You must configure EasyLocate for a specific image size.

- The image must contain less than 500 000 pixels (about 707×707 pixels for a square image).
- The width and the height must be at least 128 pixels.
- The images are automatically resized to the specified size before EasyLocate processes them.
- The lower the image size, the faster EasyLocate is.
- EasyLocate works best with objects equal to or bigger than 16×16 pixels.

In Deep Learning Studio:

- Use the **Width** and **Height** controls to change the size of the images.
- Uncheck **Keep aspect ratio** if you want to control the width and height independently of each other.
- By default, the width and height are set to the size of the images in the dataset. or, when they are bigger than 500 000 pixels, to the maximum possible size so that the aspect ratio of the image is kept and it contains at most 500 000 pixels.
- A warning is displayed when the selected size makes the ground truth objects smaller than 16×16 pixels.



Width: 450px (scale: 23.3%) ⓘ

Height: 282px (scale: 23.3%) ⓘ

Keep aspect ratio

Warning: at this resolution, you have objects whose width and/or height is smaller than 16 pixels.
EasyLocate might not be able to robustly detect small objects.

In the API:

- Use `ELocator::Width` and `ELocator::Height` to specify the image size.

Configuration (advanced parameters)

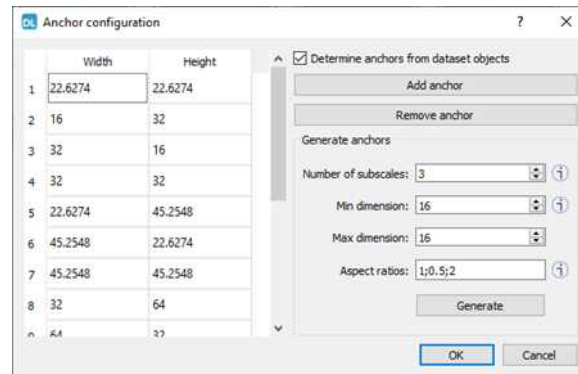
The EasyLocate tool has 4 advanced parameters linked to EasyLocate neural network design.

- The EasyLocate neural network works as an image pyramid where the size of the input image is halved at each level. EasyLocate attempts to detect objects using one or more pyramid levels depending on the size of the objects to detect.
- To do so, EasyLocate uses a set of typical object size, called *anchors*, that are assigned to pyramid levels according to their surface. Then, for each pixel and anchor of a pyramid level, EasyLocate predicts whether there is an object or not located around that pixel and whose size approximately matches the anchor.
- EasyLocate then performs a post-processing on the prediction of the neural network. Indeed, the neural network can predict the same object several times using different levels of the pyramid, different anchors or neighboring positions in the image. EasyLocate keeps only the prediction with the highest probability and removes duplicates based on the overlap between objects.

The advanced parameters are:

1. The **anchors**. By default, the anchors are determined automatically from the objects in your dataset. The set of anchors must reflect the variety of object sizes that must be detected.

To check or manually edit the anchors, click on **See/edit anchors** to open the following dialog:



The dialog lists the current anchors and enables the following operations:

- To edit an existing anchor, double-click on its width or on its height in the list.
- To add or remove an anchor, click on the corresponding button.
- To generate a new set of anchors, specify the number of subscales, the minimum and the maximum dimensions of the anchors and one or more aspect ratios.

The dimension of an anchor is the square root of its surface and determines the pyramid level assigned to the anchor. The number of subscales represent the number of dimensions to generate for each pyramid level. For each of those dimensions, the anchors with the specified aspect ratios are generated.

In the API:

- Use `ELocator::SetPredictionAnchors` and `ELocator::GenerateAnchors`.

2. The **Maximum number of objects in an image**. By default the value is 100. A lower value can speed up the post-processing of the results.

In the API:

- Use `ELocator::MaxNumberOfObjects`.

3. The **Same label maximum overlap** is the maximum overlap between objects with the same label. By default the value is 0.5.

In the API:

- Use `ELocator::SameLabelMaxOverlap`.

4. The **Absolute maximum overlap** is the maximum overlap between objects, regardless of their label. By default the value is 1 and it means that the tool can predict two objects with different labels but with the exact same bounding box.

In the API:

- Use `ELocator::SameLabelMaxOverlap`.

The overlap between two objects is their intersection over union (IoU), defined as the ratio between the surface of the intersection of their bounding boxes and the surface of the union of their bounding boxes.

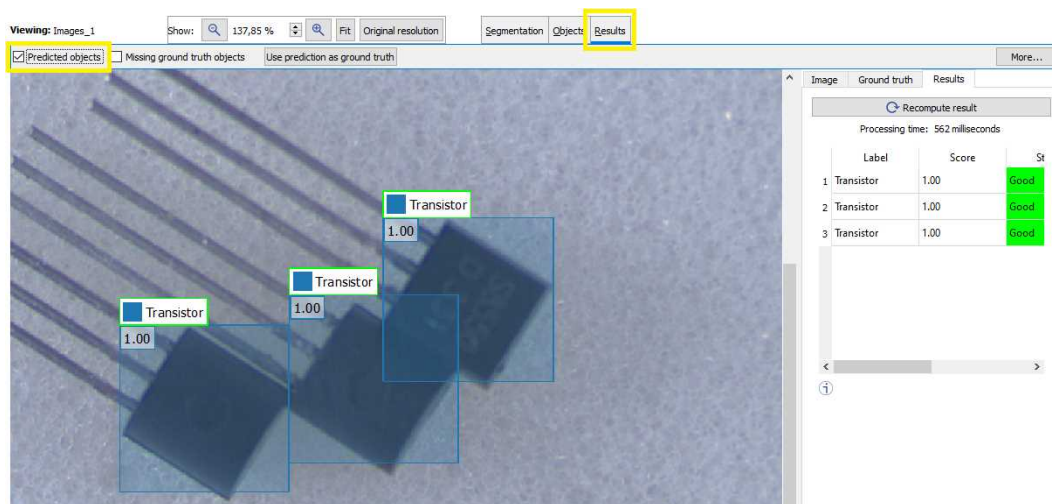
Training

To train your tool, see "Training a Deep Learning Tool" on page 65.

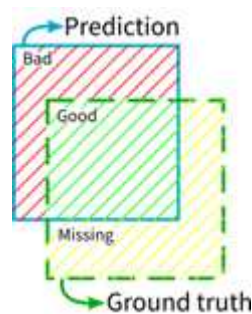
Locating Objects

In Deep Learning Studio:

- To apply EasyLocate to new images:
 - Add these images to the dataset.
 - Or use the Tests and results tab.
- To visualize the predicted objects of an image:
 - a. Open the Results menu (ALT + R) of the image viewer.
 - b. Check the Predicted objects option (CTRL + P).



- If the image has a ground truth, check the **Missing ground truth object** option (CTRL + G) to display missing objects. They appear with a yellow pattern drawn over it.
- In the **Results** tab on the right side:
 - The list of detected objects shows their label, their score, whether they are matched to a ground truth object and their predicted bounding box.
 - To see how close a predicted object is to a ground truth object, select the object in the list on the right side of the image. The ground truth object is displayed on top of the predicted object with the following color code:



- To accept all the predicted objects as ground truth:
 - Click on the **Use prediction as ground truth** button (CTRL + U).
 - Note that it removes any previous ground truth object already present in the image.
- To accept a single predicted object as ground truth:
 - a. Right click on the object.
 - b. Select **Accept into ground truth** in the menu.

Viewing: Images_2 Show: 45,17 % Fit Original resolution Objects Results

Predicted objects Missing ground truth objects **Use prediction as ground truth** More...

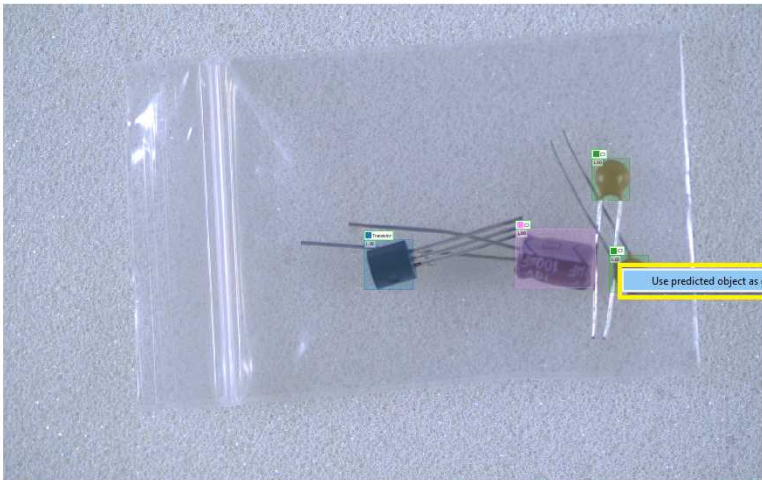


Image Ground truth Results

Recompute result

Processing time: 531 milliseconds

	Label	Score	St
1	C3	1,00	Good
2	C2	1,00	Good
3	C3	1,00	Good
4	Transistor	1,00	Good

< >

ⓘ

In the API

- To apply EasyLocate to an image, use `ELocator::Apply`.

This method returns an object `ELocatorResult`.

- Use `ELocatorResult::GetDetectedObjects` to retrieve the predicted objects as an array of `ELocatorPredictedObject`.
- Use `ELocatorResult::Draw` to draw all the predicted objects.
- For each predicted object, use:
 - `ELocatorObject::Width` to get its width.
 - `ELocatorObject::Height` to get its height.
 - `ELocatorObject::Label` to get its label.
 - `ELocatorPredictedObject::Probability` to get its predicted probability.

Validating the Results

The EasyLocate tool exposes 2 types of metrics. These object-based metrics quantify the performance of the tool :

- At the object level.
- At the image level. These metrics are related to the tool ability to correctly detect images without object and images with objects.

In Deep Learning Studio:

- The metrics are available in the **Dataset results** tab.
- Most metrics depends on the value of the **Detection threshold**.

Export results to CSV

Results for: Full dataset

Adapting the detection rule ⓘ

Detection threshold: 0,930796

Objects
Images

Metrics ⓘ

Average precision: 0.98787 ⓘ

		Label weighted	
Recall:	95.3051%	95.1773%	ⓘ
Precision:	96.8254%	97.2048%	ⓘ
F-Score:	0.959943	0.962456	ⓘ

Per label metrics

Label	# Detected	# Correct	# Bad	# Not detected	Recall
Transi...	177	163	14	2	98.79%
Diode	128	126	2	21	85.71%
C1	126	126	0	2	98.44%
C2	139	135	4	8	94.41%
C3	123	121	2	1	99.18%

In the API:

- The metrics are represented by an object `ESupervisedSegmenterMetrics`.

The object-based metrics

The object-based metrics are computed by matching actual, ground truth objects to detected objects. A ground truth object and a detected object are matched together if:

- They have the same label.
- Their overlap ("Intersection over Union") is higher or equal to the `Same label maximum overlap` parameter of the tool.
- There is no other ground truth that has a higher overlap with the detected object and there is no other detected object that has a higher overlap with the ground truth object.

The object-based metrics are:

- The `Average precision (AP)` is the average of the precision (proportion of detected objects that are matched to a ground truth objects) for different values of recall (true positive rate, proportion of ground truth objects that are matched to detected objects) obtained by varying the `Detection threshold`.
 - Its value is between 0 (bad detector) and 1 (good detector).
 - It is a standard metric for evaluating object detector.
- The `recall`, also called the "true positive rate", is the proportion of ground truth objects matched with a predicted object.
- The `weighted recall` is the weighted average of the `recall` for each label.
- The `precision`, also called the "positive predicted value", is the proportion of predicted objects matched with a ground truth object.
- The `weighted precision` is the weighted average of the `precision` for each label.
- The `F-Score` is the harmonic mean of the recall and the `precision`.
- The `weighted F-Score` is the weighted average of the `F-Score` for each label.
- The `Per label metrics` table shows various metrics of the objects of each label. The columns starting with a "#" indicate the number of objects in the corresponding category.
 - Selecting one or more cells from these columns filters the image list to show only the images that have objects falling in the corresponding categories.
 - If there is no selection, all the images are listed.
 - Use CTRL + Left Click to add cells to the current selection.

For example, the following selection shows the images containing either:

- A badly detected “transistor” object
- A detected (correctly or badly) “diode” object
- A correctly detected “C2” object
- An undetected ground truth “C3” object

Per label metrics

Label	# Detected	# Correct	# Bad	# Not detected	Recall
Transi...	177	163	14	2	98.79%
Diode	128	126	2	21	85.71%
C1	126	126	0	2	98.44%
C2	139	135	4	8	94.41%
C3	123	121	2	1	99.18%

The image-based metrics



The metrics related to the correct detection of the class of the images (background / with object or good / defective in the context of defect detection) are:

- The **image detection accuracy** is the proportion of images correctly predicted to have objects or not.
- A **Confusion matrix** listing the number of images in each category.
 - Selecting one or more cells of the confusion matrix filters the image list to show only the images in the corresponding categories.

Benchmarks for EasyLocate

Test conditions

- These numbers are only indicative and represent only the memory required for the neural network.
- Your actual memory requirements may be bigger or lower according to your GPU model.
- The GPU must have more memory than the indicated amount to work because storing images and results may require additional GPU memory and because of memory fragmentation.
- The training time is approximately twice the inference time per image. An iteration is equivalent to a loop over all the images in the dataset.
- The GPU used for these benchmarks is a NVIDIA GeForce 1080 Ti.
- The CPU used is an Intel Core i9 7900X.

Capacity small

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	58	6.7 ms	78.91 ms	-
	4	123	2.39 ms	-	265
	16	453	1.05 ms	-	988
	64	1711	1 ms	-	3506
256 × 256	1	123	7.52 ms	375.43 ms	-
	4	383	4.49 ms	-	893
	16	1711	3.22 ms	-	3720
	64	6745	3.09 ms	-	14651
512 × 512	1	383	23 ms	2508 ms	-
	4	1429	13.69 ms	-	3365
	16	6745	11.12 ms	-	14651
	64	26878	-	-	53758

Capacity normal

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	63	7.2 ms	81.55 ms	-
	4	132	2.55 ms	-	279
	16	475	1.13 ms	-	1032
	64	1795	1.04 ms	-	3674
256 × 256	1	132	6.93 ms	393 ms	-
	4	408	4.30 ms	-	937
	16	1795	3.20 ms	-	3888
	64	7074	3.28 ms	-	15310

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
512 × 512	1	405	24.18 ms	2583 ms	-
	4	1513	12.5 ms	-	3532
	16	7074	11.24 ms	-	15310
	64	28191	-	-	56386

Capacity large

Image size	Batch	Inference			Training
		GPU Memory (MB)	GPU inference time /image	CPU inference time	GPU Memory (MB)
128 × 128	1	196	12.04 ms	141 ms	-
	4	327	3.65 ms	-	726
	16	893	1.85 ms	-	1938
	64	3201	1.33 ms	-	7065
256 × 256	1	334	38 ms	684 ms	-
	4	857	9.17 ms	-	2159
	16	3201	5.46 ms	-	6699
	64	12434	-	-	26030
512 × 512	1	851	34.6 ms	4266 ms	-
	4	2952	22.47 ms	-	6174
	16	12434	-	-	26030
	64	49367	-	-	98737

4. Code Snippets

4.1. Basic Types

Loading and Saving Images

Functional Guide | Reference: [Load](#), [Save](#), [SaveJpeg](#)

```

////////////////////////////////////
// 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

Functional Guide | Reference: [SetImagePtr](#)

```

////////////////////////////////////
// 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

Functional Guide | Reference: [GetImagePtr](#)

```

////////////////////////////////////
// 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;
int 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

Functional Guide | Reference: [Attach](#), [SetPlacement](#)

```

////////////////////////////////////
// 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

Functional Guide | Reference: [Empty](#), [AddElement](#)

```

////////////////////////////////////
// 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 < 28; i++)
{
    ramp.AddElement((EBW8)i);
}

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

```

Exception Management

Functional Guide | Reference: [GetPixel](#), [What](#)

```

////////////////////////////////////
// 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, 73);
}

catch(Euresys::Open_eVision_::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.SplitDataset(trainingDataset, validationDataset, 0.8f);

// 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++ ↪ 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 + ↪) % 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();
```