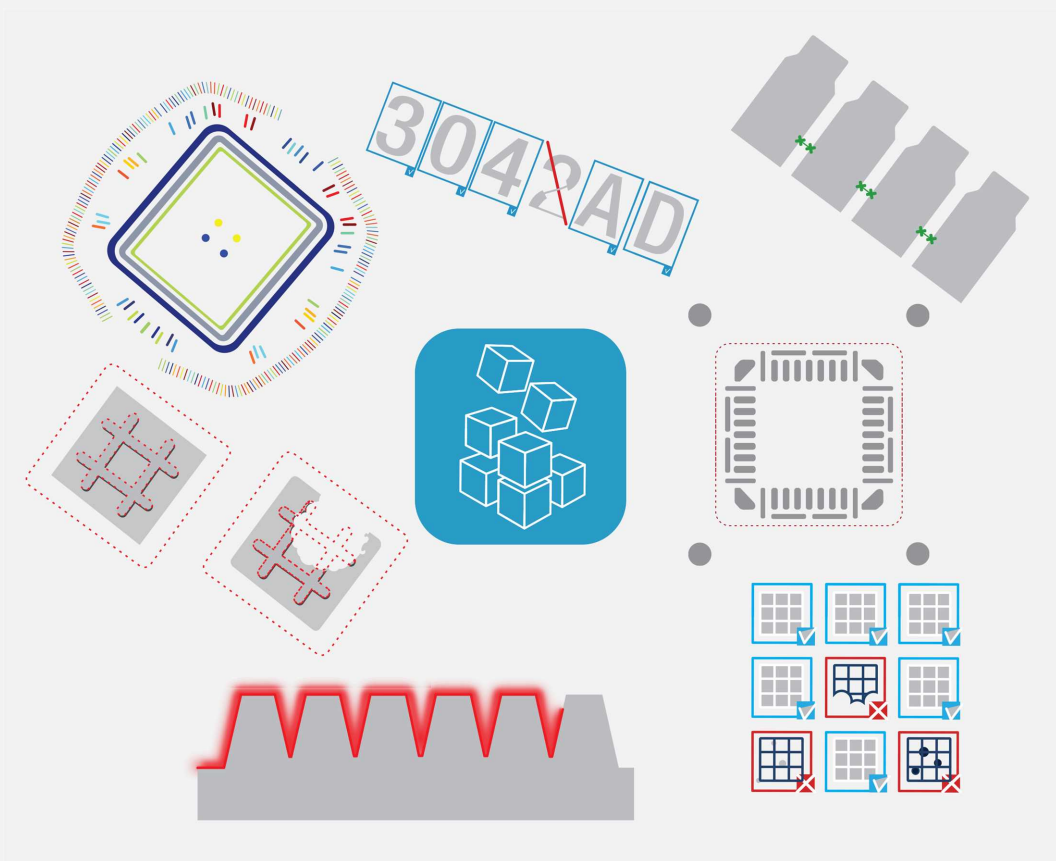


Open eVision

Deep Learning Inspection Tools



This documentation is provided with **Open eVision 24.02.0** (doc build **1198**).
www.euresys.com

This documentation is subject to the General Terms and Conditions stated on the website of **EURESYS S.A.** and available on the webpage <https://www.euresys.com/en/Menu-Legal/Terms-conditions>. The article 10 (Limitations of Liability and Disclaimers) and article 12 (Intellectual Property Rights) are more specifically applicable.

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	9
2. Conventions	10
2.1. Conventions for Strings	10
2.2. Image Coordinate Systems	10
2.3. Image and Depth Map Buffer	12
3. Basic Operations	14
3.1. Memory Allocation	14
3.2. Loading a Pixel Container File	15
3.3. Saving a Pixel Container File	16
3.4. Drawing in Open eVision	18
3.5. 3D Rendering of 2D Images	21
3.6. Vector Types and Main Properties	22
3.7. ROI Main Properties	26
3.8. Arbitrarily Shaped ROI (ERegion)	28
3.9. Flexible Masks	50
3.10. Profile	54
4. Deep Learning Tools	56
Deep Learning Tools - Inspecting Images with Deep Learning	56
Purpose and Workflow	56
Deep Learning Studio and Additional Resources	59
Engines and Hardware Support (CPU/GPU)	63
Managing the Dataset and the Annotations	68
Images and Labels	68
Adding Images	71
Editing the Label of an Image	73
Editing the Segmentation of an Image	74
Editing the Objects of an Image	75
ROI and Mask	77
Managing the Dataset Splits	80
Using Data Augmentation	83
Training a Deep Learning Tool	86
Using a Deep Learning Tool	89
Benchmarking a Deep Learning Tool	90
EasyClassify - Classifying Images	95
Tool and Images	95
Validating the Results	99
Classifying New Images	101
Benchmarks for EasyClassify	104
EasySegment - Detecting and Segmenting Defects	108
Unsupervised vs Supervised Modes	108
EasySegment Unsupervised	109
Tool and Configuration	109
Validating the Results	112
Applying the Tool to New Images	114
Benchmarks for EasySegment Unsupervised	115
EasySegment Supervised	118
Tool and Configuration	118
Using the Supervised Segmenter	120

- Evaluating the Results 123
- Benchmarks for EasySegment Supervised 127
- EasyLocate - Locating Objects and Defects 130
 - Tool and Configuration 130
 - Locating Objects 135
 - Validating the Results 138
 - Benchmarks for EasyLocate 140
- 5. Code Snippets 144**
 - 5.1. Basic Types 145**
 - Loading and Saving Images 145
 - Interfacing Third-Party Images 145
 - Retrieving Pixel Values 146
 - ROI Placement 146
 - Vector Management 146
 - Exception Management 147
 - 5.2. Deep Learning Tools 147**
 - Creating a Dataset and Training a Classifier 147
 - Loading a Classifier and Classifying a New Image 148
 - Using Multithreading for Classification 149
 - Loading an Unsupervised Segmenter and Segmenting an Image 150

1. Dealing with Pixel Containers and Files

1.1. Pixel Container Definition

Image objects

The **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.

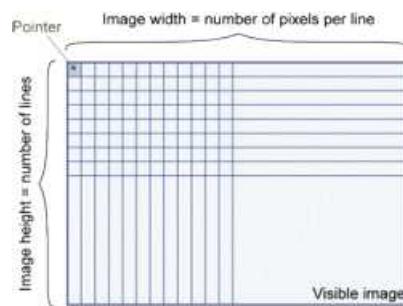


Image main parameters

The rectangular array of pixels of an **Open eVision** image object is characterized by the **EBaseROI** parameters:

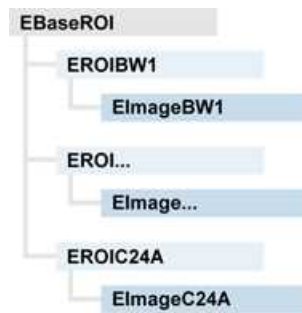
- The **Width** is the number of pixels per row of the image.
- The **Height** is the number of rows of the image.
- The **Size** contains both the **Width** and the **Height** of the image.

The maximum size for the width and the height is:

- 32,767 ($2^{15}-1$) in **Open eVision** 32-bit
- 2,147,483,647 ($2^{31}-1$) in **Open eVision** 64-bit
- The **Plane** contains the number of color components.
 - For gray-level images: **Plane** = 1
 - For color images: **Plane** = 3

Classes

The image and ROI classes derive from the abstract class `EBaseROI` and inherit all its properties.



Depth maps

A *depth map* represents a 3D object using a 2D grayscale image in which each pixel represents a 3D point.

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

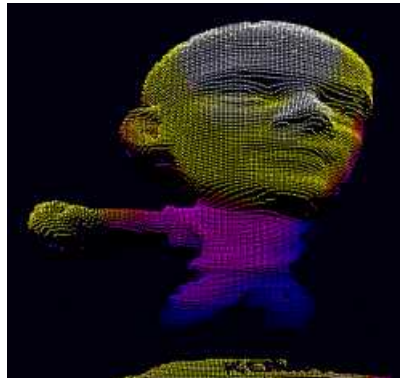


Point clouds

A *point cloud* is an unstructured set of 3D points representing discrete positions on the surface of an object.

The point clouds are produced by various 3D scanning techniques, such as laser triangulation, time of flight or structured lighting.

 For details, see, for example, en.wikipedia.org/wiki/Point_cloud.



1.2. Pixel Container Types

 For the enumeration of the available types, see "[EImageType Enum](#)" on page 1.

Images

Open eVision supports the following image types according to their pixel types.

Open eVision	Genicam PNFC	Definition	Class
BW1	Mono1	1-bit black and white image (8 pixels are stored in 1 byte).	EImageBW1
BW8	Mono8	8-bit grayscale image (each pixel is stored in 1 byte).	EImageBW8
BW16	Mono16	16-bit grayscale image (each pixel is stored in 2 bytes).	EImageBW16
BW32	Mono32	32-bit grayscale image (each pixel is stored in 4 bytes).	EImageBW32
C15	RGB5	15-bit color image (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images.	EImageC15
C16	RGB565	16-bit color image (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB16 color images.	EImageC16
C24	RGB8	24-bit color image (each pixel is stored in 3 bytes). Compatible with Microsoft® Windows RGB24 color images.	EImageC24
C24A	BGRa8	32-bit color image (each pixel is stored in 4 bytes). Compatible with Microsoft® Windows RGB32 color images.	EImageC24A



TIP

[Easy.GetBestMatchingImageType](#) returns the best matching image type for a given file on disk.

Depth Maps

Open eVision	Genicam PNFC	Definition	Class
EDepth8	Coord3D_C8	8-bit depth map (each pixel is stored in 1 byte as an integer)	EDepthMap8
EDepth16	Coord3D_C16	16-bit depth map (each pixel is stored in 2 bytes as a fixed point)	EDepthMap16
EDepth32f	Coord3D_C32	32-bit depth map (each pixel is stored in 4 bytes as a float)	EDepthMap32f



TIP

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

Point Clouds

Open eVision	Genicam PNFC	Definition	Class
Point Cloud	Coord3D_ABC32	Set of points coordinates (each coordinate is stored in 4 bytes as a float)	EPointCloud

1.3. Supported Image File Types

 For the enumeration of the available types, see "[EImageFileType Enum](#)" on page 1.

Type	Description
BMP	Uncompressed image data format (Windows Bitmap Format).
JPEG	A lossy data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 10918-1. The compression irretrievably loses quality.
JFIF	JPEG File Interchange Format.
JPEG-2000	A 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. - The <i>code stream</i> describes the image samples. - The <i>file format</i> includes meta-information such as the image resolution and color space.
PNG	Lossless data compression method (Portable Network Graphics).
Serialized	The Euresys proprietary image file format obtained from the serialization of Open eVision image objects.
TIFF	The Tag Image File Format is currently controlled by Adobe Systems and uses the LibTIFF third-party library to process images written for the 5.0 or 6.0 TIFF specification. - The file <i>save</i> operations are lossless and save the images without any compression. - The file <i>load</i> operations support all the TIFF variants listed in the LibTIFF specification.

1.4. Pixel and File Types Compatibility

For the compatible combinations in the following table, the image integrity is preserved with no data loss (except from JPEG and JPEG2000 with lossy compression).

The other combinations are not supported and an exception occurs if you use them.

Type	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
BW1	✓	–	–	✓	✓	✓
BW8	✓	✓	✓	✓	✓	✓
BW16	–	–	✓	✓	✓ ²	✓
BW32	–	–	–	–	✓ ²	✓
C15	✓	✓ ¹	✓ ¹	✓ ¹	✓ ¹	✓
C16	✓	✓ ¹	✓ ¹	✓ ¹	✓ ¹	✓
C24	✓	✓	✓	✓	✓ ¹	✓
C24A	✓	–	–	✓	–	✓
Depth8	✓	✓	✓	✓	✓	✓
Depth16	–	–	✓	✓	✓ ²	✓
Depth32f	–	–	–	–	–	✓

- ✓¹: C15 and C16 formats are automatically converted into C24 during the save operation.
- ✓²: BW16 and BW32 are not supported by Baseline TIFF readers.

1.5. Color Types

Open eVision supports the following color systems:

EISH	Intensity, Saturation, Hue
ELAB	CIE Lightness, a*, b*
ELCH	Lightness, Chroma, Hue
ELSH	Lightness, Saturation, Hue
ELUV	CIE Lightness, u*, v*
ERGB	NTSC/PAL/SMPTE Red, Green, Blue
EVSH	Value, Saturation, Hue
EXYZ	CIE XYZ
EYIQ	CCIR Luma, Inphase, Quadrature
EYSH	CCIR Luma, Saturation, Hue
EYUV	CCIR Luma, U Chroma, V Chroma

2. Conventions

2.1. Conventions for Strings

Since **Open eVision** 23.08, the only character encoding used in the **Open eVision** libraries and tools is UTF-8.

- All methods taking `std.string` as argument expect an UTF-8 encoded `std.string`.
- All methods returning a `std.string` always return it as UTF-8 encoded.

[Backward compatibility on Windows](#)

On **Windows** (but not on **Linux**), there is also a sanitization process to preserve backward compatibility with older releases that didn't use the UTF-8 encoding.

- The content of each input string is checked to ensure it is UTF-8 encoded.
If it is not the case:
 - The string is assumed to be encoded using the current Windows Language for Non-Unicode Programs parameter.
 - It is converted to UTF-8.
- The output strings of all libraries and tools are always UTF-8.



TIP

Despite the presence of this backward compatibility layer it is recommended to use exclusively UTF-8 to interact with **Open eVision** on all platforms to ensure the best performance and compatibility.

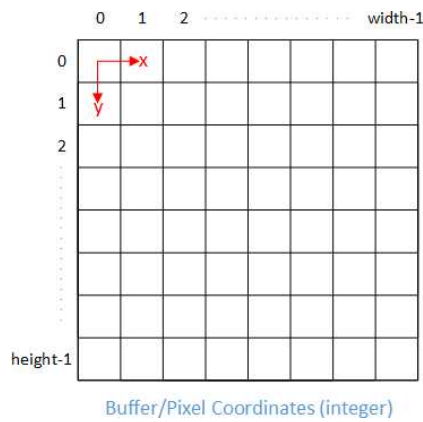
2.2. Image Coordinate Systems

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 library.

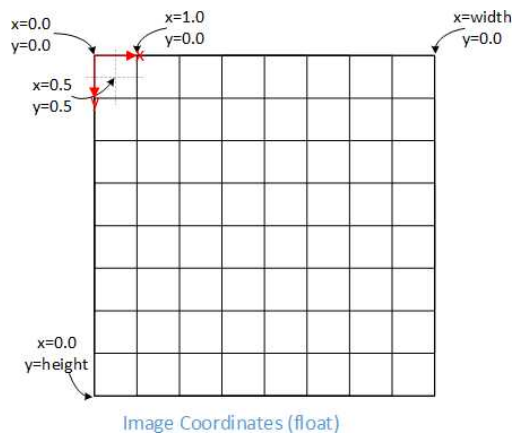
Integer coordinates

- 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.

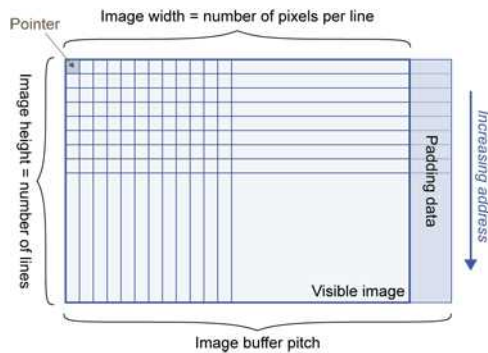


2.3. Image and Depth Map Buffer

The pixels of an image and of a depth map are stored contiguously into a buffer, from left to right and from top to bottom, in the Windows bitmap format (top-down DIB -device-independent bitmap-).

The buffer address is a pointer to the address that contains the top left pixel of the image.

- Image buffer pitch
 - The alignment must be a multiple of 4 bytes.
 - The default pitch in **Open eVision** is 32 bytes for performance reasons.



Memory layout

Image format	Layout	Illustration
EImageBW1	Stores 8 pixels in 1 byte	
EImageBW8 EDepthMap8	Store 1 pixel in 1 byte	
EImageBW16	Stores 1 pixel in 2 bytes	
EImageC15	Stores 1 pixel in 2 bytes - Each color component is coded with 5 bits - The 16th bit is unused	

Image format	Layout	Illustration
<p>EImageC16</p>	<p>Stores 1 pixel in 2 bytes</p> <ul style="list-style-type: none"> - The colors 1 and 3 are coded with 5 bits - The color 2 is coded with 6 bits 	
<p>EDepthMap16</p>	<p>Stores 1 pixel in 2 bytes (fixed point format)</p>	
<p>EImageC24</p>	<p>Stores 1 pixel in 3 bytes</p> <ul style="list-style-type: none"> - Each color component is coded with 8 bits 	
<p>EImageC24A</p>	<p>Stores 1 pixel in 4 bytes.</p> <ul style="list-style-type: none"> - Each color component is coded with 8 bits - The alpha channel is coded with 8 bits 	
<p>EDepthMap32f</p>	<p>Stores 1 pixel in 4 bytes (float format)</p>	

3. Basic Operations

3.1. Memory Allocation

You can construct an image using an internal or an external memory allocation.

Internal memory allocation

The image object dynamically allocates and deallocates a buffer:

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

To declare an image with an internal memory allocation:

1. Construct an image object, for instance `EImageBW8`, either with width and height arguments or using the `SetSize` function.
2. Access a given pixel using one of the multiple available functions.
For example, use `GetImagePtr` to retrieve a pointer to the first byte of the pixel at the given coordinates.

External memory allocation

Control the buffer allocation or link a third-party image in the memory buffer to an **Open eVision** image.

- You must specify the image size and the buffer address.
- When an image object is destroyed, the buffer is unaffected.

 For details, see "Image and Depth Map Buffer" on page 12 and "Interfacing Third-Party Images" on page 145.

To declare an image with an external memory allocation:

1. Declare an image object, for instance [EImageBW8](#).
2. Create a suitably sized and aligned buffer.
3. Assign the buffer to the image with [SetImagePtr](#).

**NOTE**

Using the copy constructor of the EImage object to copy the externally allocated image does not copy the buffer. The copied image points to the same external buffer as the original image.

**NOTE**

If your buffer rows are not aligned on 4 bytes, use [InitializeFromUnalignedBuffer](#) instead of [SetImagePtr](#). Please note that this allocates the memory internally and copies the external buffer into the internal one instead of using the external one.

3.2. Loading a Pixel Container File

Loading images and depth maps

- Use the method [Load](#) to load image data into an image object.
 - It has only the argument path that includes the path, filename and file name extension.
 - The file type is determined by the file format.
 - The destination image is automatically resized according to the size of the image on disk.
- [Load](#) throws an exception when:
 - The file type identification fails.
 - The file type is incompatible with the pixel type of the image object.

NOTE: 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 (supported in ASCII and binary modes).
- [EPointCloud.LoadPLY](#): PLY file (supported only in ASCII mode).
- [EPointCloud.LoadXYZ](#): XYZ file.

3.3. Saving a Pixel Container File

Images and depth maps

- Use the method `Save` of an image or the method `SaveImage` of a depth map or a ZMap to save image data of the object into a file.
 - The argument `Path` includes the path, file name and file name extension.
 - The argument `Image File Type` can be omitted. In this case, the file name extension is used.
- `Save` throws an exception when:
 - The requested image file format is incompatible with the pixel type of the image object.
 - The file name extension is not supported while using the Auto file type selection method.

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



TIP

The images with a width or a height larger than 65,536 must be saved in **Open eVision** proprietary format.

Image File Type arguments

Argument	Image file type
<code>EImageFileType_Auto</code>	(Default) Automatically determined by the file name 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

If the argument is `EImageFileType_Auto` or is missing, the assigned image file type is:

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

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.

3.4. Drawing in Open eVision

Introduction

- Whenever relevant, the **Open eVision** tools provide methods Draw to render their contents and/or configuration. This is, for instance, the contents of an EImage or the frame of an EROI.
- A given tool can have multiple methods Draw, usually one for each feature available.
- The **Open eVision** methods Draw take an object DrawAdapter as their main parameter, and additional parameters for zoom and pan:

```
Tool::Draw(EDrawAdapter* adapter, float zoomX, float zoomY, float panX, float panY);
```

- zoomX and zoomY are expressed in percentage, 1 is the default value and means no zoom.
- It can be different in the horizontal and vertical directions (which can be useful in the case of non-square pixels for instance).
- If you don't provide a vertical zoom, or set it to 0, it will be set identical to the horizontal one.
- panX and panY are expressed in pixels, but in image coordinates. It means that the value you pass to panX and panY are multiplied by the corresponding zoom before being applied.

Example: How to draw an image and a ROI frame on a window under Windows:

```
EImageBW8 image;
EROIBW8 roi;
EWindowsDrawAdapter adapter(windowHdc);
image.Draw(adapter);
roi.DrawFrame(adapter);
```

Graphical interactions

- You can configure some of the **Open eVision** tools graphically and use the provided methods to put your configuration in place.
- Graphical Interaction-enabled tools provide special parameters to some of their methods Draw to draw handles on the tool representation.
- To capture the user interactions with those handles, these tools also provide two specialized methods:
 - HitTest detects if a handle is under the mouse when providing it with the current cursor coordinates. You typically use this test during a mouse button down event.
 - Drag moves the detected handle to the given coordinates. This in turn modifies the tool configuration to match the new handle position. Drag is typically associated with the mouse button up event.

NOTE: HitTest and Drag use the same zoom and pan parameters as Draw. You must set them the same way (with the same values) to achieve the desired result.

Draw adapters

- The draw adapters are objects that, in addition to representing the context in which to draw, provide methods to draw the selected primitives in that context.
- They are initialized by providing the targeted context to the constructor.

- Some of the drawing methods provided by the draw adapters are (but are not limited to):
 - `EDrawAdapter::Line` / `Lines` draws one or more lines on the context
 - `EDrawAdapter::Rectangle` / `FilledRectangle` draws a rectangle, filled or not, on the context
 - `EDrawAdapter::Ellipse` / `FilledEllipse` draws an ellipse, filled or not, in the context
 - `EDrawAdapter::Text` / `BackedText` renders a text in the context, with or without background
 - `EDrawAdapter::Image` renders an image in the context
- For more information about the drawing primitives provided by the draw adapters, please refer to the reference documentation.
- To set the color of the primitives, provide a pen and/or a brush and use the methods `EDrawAdapter::SetPen` and `EDrawAdapter::SetBrush`.
 - If you do not provide a pen and/or a brush, the default colors are used.
- To set the font of the text, provide a font with the method `EDrawAdapter::SetFont`.

Standard draw adapters

Open eVision provides a set of off-the-shelf draw adapters that you can use in different situations:

- `EWindowsDrawAdapter` allows to draw on **Windows** systems. To draw on a window, provide the window's HDC to its constructor, or, to draw in an EImage buffer, provide that EImage.
 - It relies on GDI and GDI+ to provide its services.
 - This is the preferred way to draw on **Windows**.
- `QtDrawAdapter` allows you to draw using **Qt** on a QPainter context. To draw on a QPainter context, provide the QPainter to the constructor, or, to draw on an EImage buffer, provide that EImage.
 - You can use the `QtDrawAdapter` both on **Windows** and **Linux**.
 - This is the preferred way to draw on **Linux**.

NOTE: `QtDrawAdapter` is using an external resource (namely **Qt**) and as such is provided as source code in its own header rather than in the global **Open eVision** header. For more information about external and custom draw adapters, see below.
- `EGenericDrawAdapter` is a draw adapter that can only render on an EImage, but it can do it in a consistent manner on all supported OSes.
 - It is available on both **Windows** and **Linux**.

Drawing in an EImage

- As said above, you can draw in an EImage (usually an `EImageBW8` or `EImageC24`) by initializing a draw adapter with that image and using either the **Open eVision** methods `Draw` or the draw adapter drawing primitives:

```
EImageBW8 image;  
EMatrixCode code;  
EWindowsDrawAdapter adapter(image);  
code.DrawPosition(adapter);
```

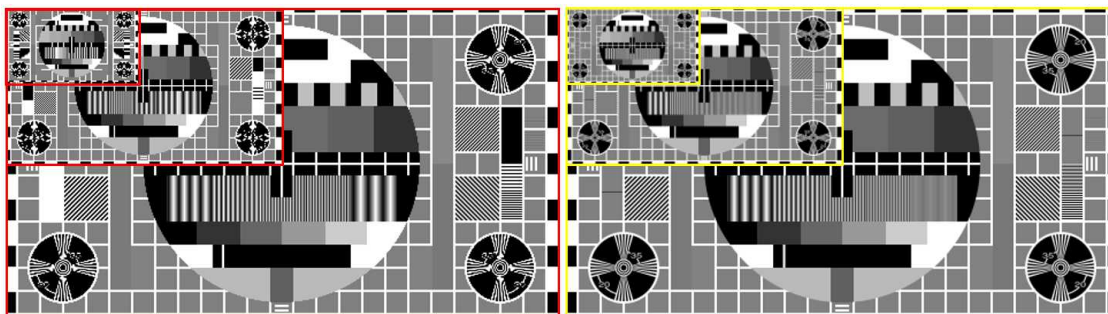
Custom draw adapters

- If you require a draw adapter to render in a specific, unsupported type of context (for ex. a DirectDraw surface, an OpenGL context...), you can build your own draw adapter by deriving from the interface `EExternalDrawAdapter` provided by **Open eVision** and implementing all the required methods.
- Once this work is done, you will be able to use your new, custom draw adapter in the same way as the off-the-shelf ones, taking advantage of **Open eVision** methods `Draw`.
- The provided `QtDrawAdapter` is a draw adapter built using that mechanism, you can use it as a reference on how to build a custom draw adapter. The sources of the `QtDrawAdapter` are bundled with the Qt Samples.

Enhanced Image Display

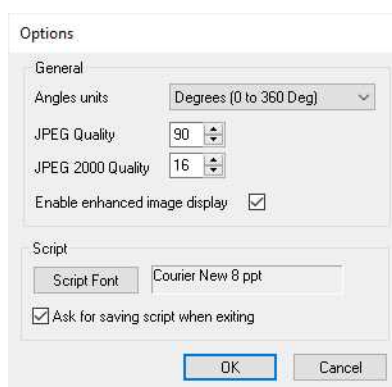
When the enhanced image display mode is enabled, a high-quality interpolation method is used to display the resized images.

- Set `Easy::SetEnableEnhancedImageDisplay(bool)` to `TRUE`, to enable the enhanced image display.
- By default, this option is disabled.
- Enhanced image display has a significant impact on display speed, the drawing can be 4x to 10x slower.
- The drawing of images with `EBW8Vector` or `EC24Vector` used as Look Up Table doesn't support enhanced image display



EnhancedImageDisplay disabled (left) and enabled (right)

- **Open eVision Studio** exposes this option in `View > Option` dialog:

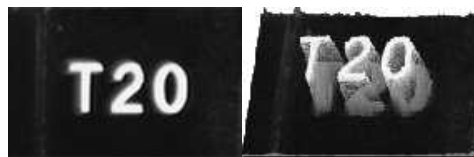


3.5. 3D Rendering of 2D Images

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.

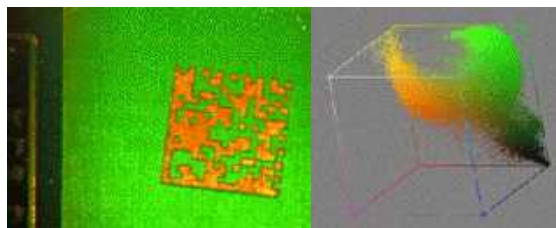


3D rendering

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.

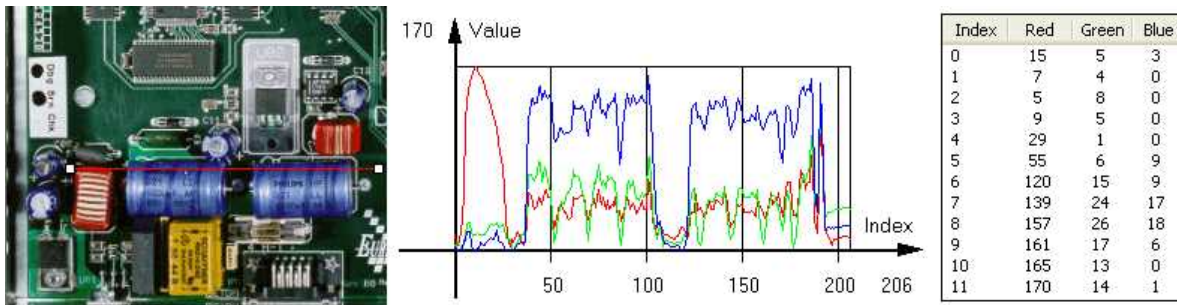


Color histogram rendering

3.6. 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](#))

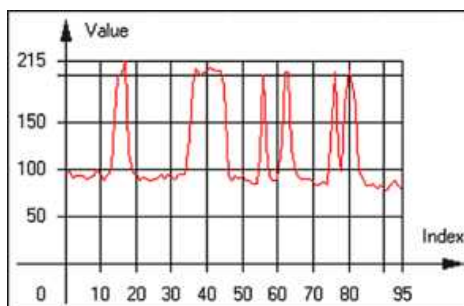
A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

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.

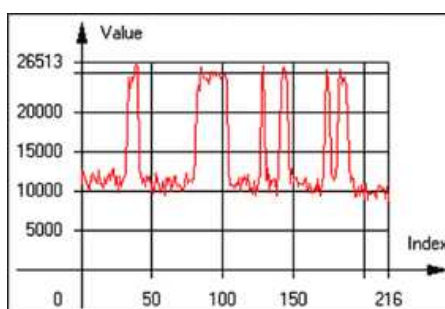
Vector types

- **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...`).



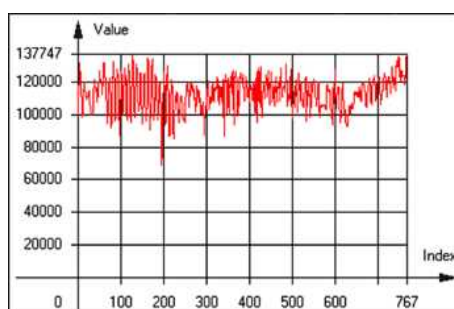
Graphical representation of an **EBW8Vector** (see `Draw` method)

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



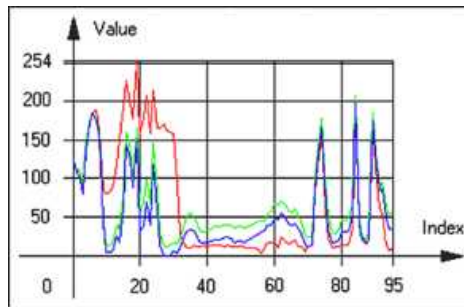
Graphical representation of an **EBW16Vector**

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



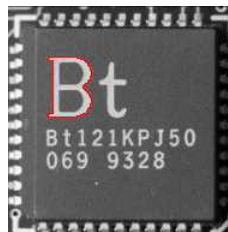
Graphical representation of an **EBW32Vector**

- **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`, ...).



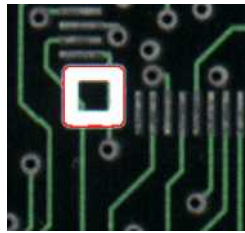
Graphical representation of an **EBW8PathVector** (see `Draw` method)

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



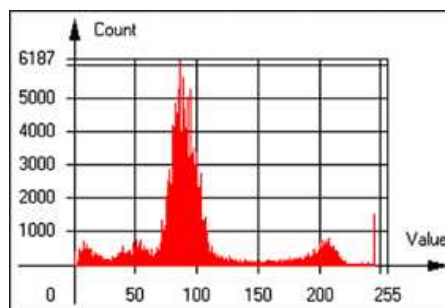
Graphical representation of an **EBW16PathVector** (see `Draw` method)

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



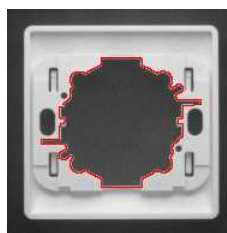
Graphical representation of an `EC24PathVector` (see `Draw` method)

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



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

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



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

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

3.7. ROI Main Properties

ROIs are defined by a [width](#), a [height](#), and **origin x and y coordinates**.

The origins are specified with respect to the top left corner in the parent image or ROI.

The ROI must be wholly contained in its parent image.

The processing/analysis time of a BW1 ROI is faster if [OrgX](#) and [Width](#) are multiples of 8.

Save and load

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

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

ROI Classes

An Open eVision ROI inherits parameters from the abstract class [EBaseROI](#).

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

- [EROIBW1](#)
- [EROIBW8](#)
- [EROIBW16](#)
- [EROIBW32](#)
- [EROIC15](#)
- [EROIC16](#)
- [EROIC24](#)
- [EROIC24A](#)

Attachment

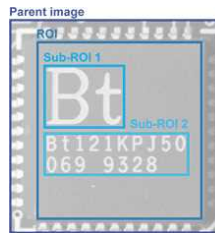
An ROI must be [attached](#) to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.

A normal image cannot be attached to another image or ROI.

Nesting

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

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



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

Cropping

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



NOTE

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

Resizing and moving

ROIs can easily be resized and positioned by two functions and dragging handles:

- `EBaseROI.Drag` adjusts the ROI coordinates while the cursor moves.
- `EBaseROI.HitTest` informs if the cursor is placed over a dragging handle.
 - Once the handle is known, the cursor shape can be changed by an `OnSetCursor` MFC event handler. `HitTest` is unpredictable if called while dragging is in progress.
 - `HitTest` can be used in an `OnSetCursor` MFC event handler to change the cursor shape, or before a dragging operation like `OnLButtonDown`, (or `EvSetCursor` and `EvLButtonDown` in Borland/CodeGear's OWL) (or `FormMouseMove` and `FormMouseDown` in Borland/CodeGear's VCL).

3.8. Arbitrarily Shaped ROI (ERegion)

See also: [example: Inspecting Pads Using Regions](#) / [code snippets: ERegion](#)

Regions or arbitrarily shaped ROI

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

In **Open eVision**:

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

Currently, only the following **Open eVision** methods support [ERegions](#):

Library	Method
EasyImage	EasyImage::Threshold

Library	Method
	EasyImage::AutoThreshold

Library	Method
	EasyImage: :Copy

Library	Method
	EasyImage::ConvolKernel

Library	Method
	EasyImage::ConvolSymmetricKernel

Library	Method
	EasyImage: :ConvolveLowpass1

Library	Method
	EasyImage: :ConvolveLowpass2

Library	Method
	EasyImage: :ConvolveLowpass3

Library	Method
	EasyImage::ConvolUniform

Library	Method
	EasyImage::ConvolGaussian

Library	Method
	EasyImage: :ConvolHighpass1

Library	Method
	EasyImage: :ConvolHighpass2

Library	Method
	EasyImage::ConvolGradientX

Library	Method
	EasyImage::ConvolGradientY

Library	Method
	EasyImage::ConvolGradient
	EasyImage::ConvolSobelX
	EasyImage::ConvolSobelY
	EasyImage::ConvolSobel
	EasyImage::ConvolPrewittX
	EasyImage::ConvolPrewittY
	EasyImage::ConvolPrewitt
	EasyImage::ConvolRoberts
	EasyImage::ConvolLaplacianX
	EasyImage::ConvolLaplacianY
	EasyImage::ConvolLaplacian8
	EasyImage::DilateBox
	EasyImage::ErodeBox
	EasyImage::OpenBox
	EasyImage::CloseBox
	EasyImage::WhiteTopHatBox
	EasyImage::BlackTopHatBox
	EasyImage::MorphoGradientBox
	EasyImage::ErodeDisk
	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
	EasyImage::ImageToLineSegment
	EasyImage::ImageToPath

Library	Method
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

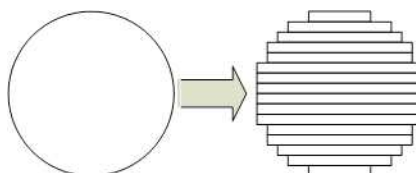
In the future **Open eVision** releases, the support of ERegions will be gradually extended to all operators.

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**

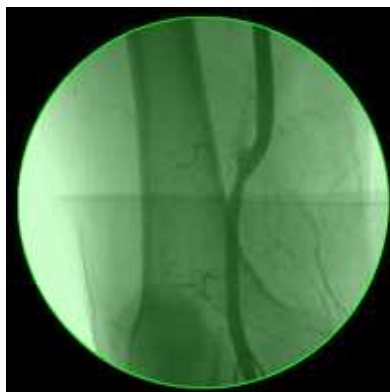
- The contour of an **ERectangleRegion** class is a rectangle.
- Define it using its center, width, height and angle.
- Alternatively, use an **ERectangle** instance, such as one returned by an **ERectangleGauge** instance.



Rectangle region separating a bar code from the background

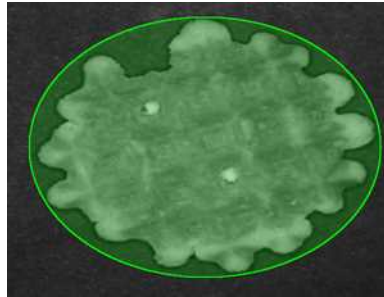
- **ECircleRegion**

- 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](#)

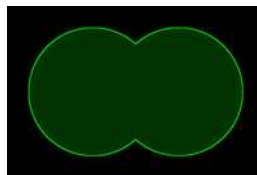
**TIP**

When compatible, **Open eVision** also provides specialized constructors for the geometry-based regions. For instance, [ECircleRegion](#) provides a constructor using an [ECircle](#).

Combining regions

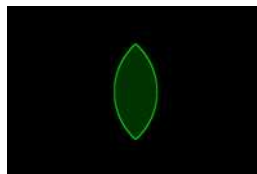
Use the following operations to create a new region by combining existing regions:

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



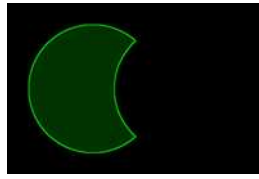
Union of 2 circles

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



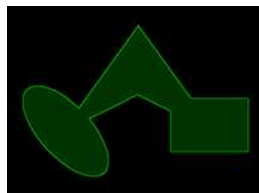
Intersection of 2 circles

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



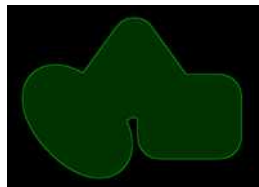
Subtraction of 2 circles

Morphological operations on regions



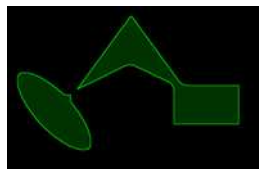
The initial arbitrary region used to illustrate the different morphological operations

- 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.



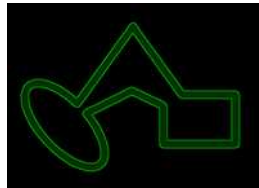
Grow of the arbitrary region

- 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.

3.9. Flexible Masks

ROIs vs flexible masks

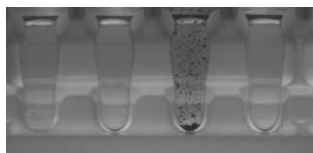
ROIs and masks restrict processing to part of an image:

- "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.

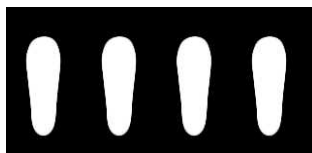
Flexible Masks

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):

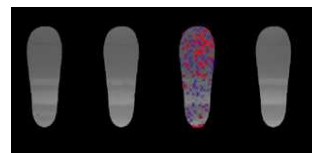
- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.



Source image



Associated mask

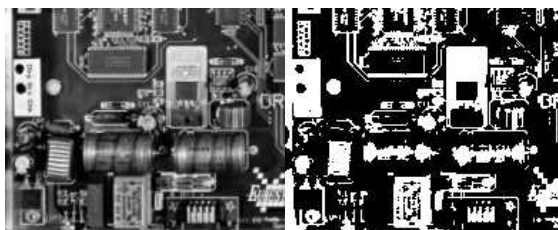


Processed masked image

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

Flexible Masks in EasyImage

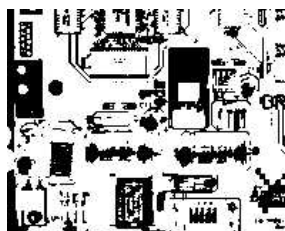
Code Snippets



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.

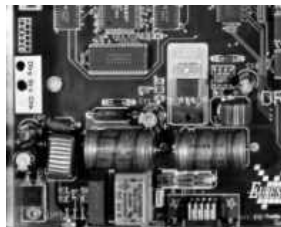
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

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

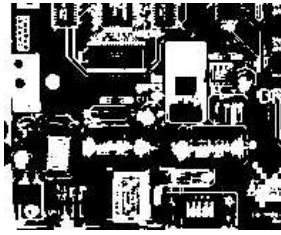


Source image

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())`.

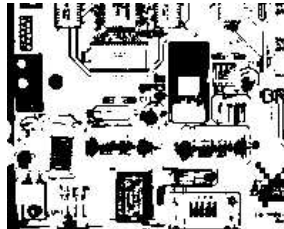
5. Exploit the flexible mask as an argument to `ECodedImage2.RenderMask`.



BW8 resulting image that can be used as a flexible mask

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

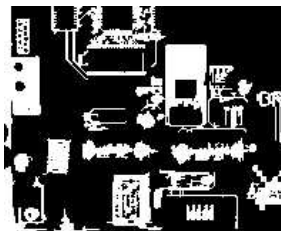
1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement.RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.



BW8 resulting image that can be used as a flexible mask

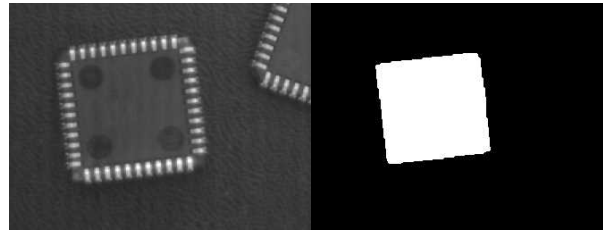
3) `EObjectSelection.RenderMask`: from a selection of blobs

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



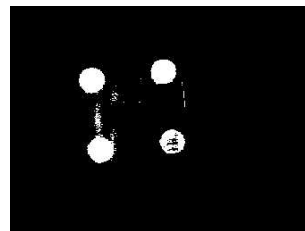
BW8 resulting image that can be used as a flexible mask

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.

3.10. 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.

4. 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.
 - **EasyLocate** has two modes:
 - With **EasyLocate Axis Aligned Bounding Box**, the location of the object is represented by its bounding box.
 - With **EasyLocate Interest Point**, the location of the object is represented by its position only.
 - **EasyLocate** works well with partially occluded objects.
 - You can use it for defect detection, package verification 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

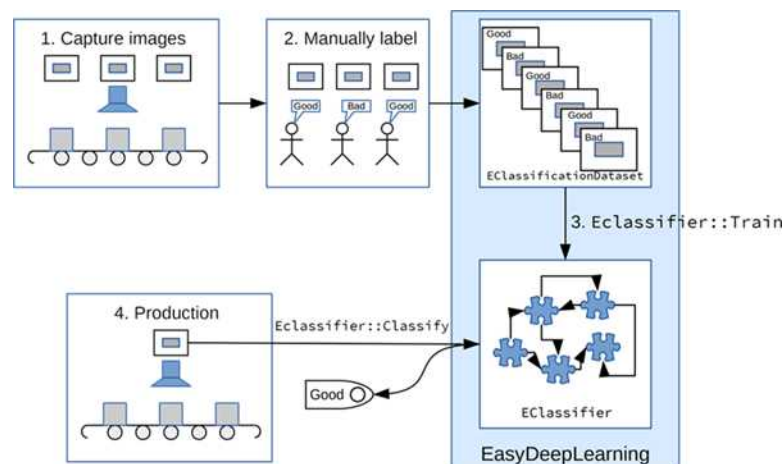
	EasyClassify	EasySegment Unsupervised	EasySegment Supervised	EasyLocate
Minimum image size	128 × 128	64 × 64		128 × 128
Maximum image size	1024 × 1024	10 000 × 10 000		500 000 pixels (for ex. 707 × 707 for square image)
Best image size	256 × 256 - 600 × 600	n.a.		n.a.
Number of channels	1 or 3 (grayscale 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 "[Engines and Hardware Support \(CPU/GPU\)](#)" on page 63 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:**
 - For **EasyLocate Axis Aligned Bounding Box**, you must annotate the defects or the objects with a bounding box and a label.
 - For **EasyLocate Interest Point**, you must annotate the defects or the objects with their position and a label.

Use the class `EClassificationDataset` to compile your labeled images.

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

All deep learning tools are child classes of the class `EDeepLearningTool`:

- **EasyClassify:** class `EClassifier`.
- **EasySegment:** classes `EUnsupervisedSegmenter` and `ESupervisedSegmenter`.
- **EasyLocate:** classes `ELocator` and `EInterestPointLocator`.

4. Train the deep learning tool on the dataset.

5. Apply the trained tool in production.

Each tool returns a specific object.

Deep Learning Studio and Additional Resources

Deep Learning Studio

Deep Learning Studio is a graphical user interface that you can use to create datasets and train your deep learning tools,

Deep Learning Studio is organized around the concept of projects.

A project consists in:

- A tool type.
- A dataset (images and their annotations).
- Several dataset splits.
- The data augmentation settings.
- The tools that you can train on the dataset.
- The results and metrics for each tool to analyze their performance.

The project is stored as a folder containing:

- The project file with the `.edlproject` extension.
- The Images folder with the imported images.
- A dedicated folder for each tool.

By default, the projects are stored in the folder named Euresys Deep Learning Studio Projects located in your Documents folder. After loading a project, the project file is automatically locked when it is used. This means that when a program opens a **Deep Learning** project file, this project file is inaccessible by other programs as long as the initial program doesn't explicitly close the project.

Here are the various file extension used within a project:

- `.edlproject`: the project file containing the dataset, the splits, the list of tools and the data augmentation settings.
- `.edlsettings` (previously `.ecl`): the settings for a tool.
- `.edlmodel` (previously `.ecl`): a deep learning model for the tool.
 - The training model is the model obtained after the last training iteration.

- The inference model is the model that is applied to images. It is the model that gave the best performances during the training.

- .edltool (previously .ecl): a complete usable tool (obtained by exporting the tool from **Deep Learning Studio**).
- .edlmetrics (previously .edl): the metrics for the tools.
- .edlresult: a result for a given image and tool.
- .edldataset (previously .eds): a dataset (obtained by exporting the dataset from the project).

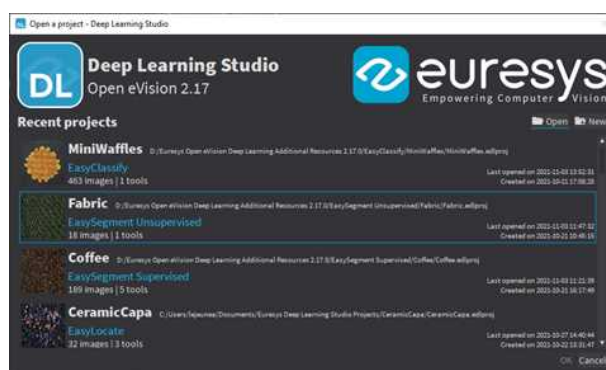


TIP

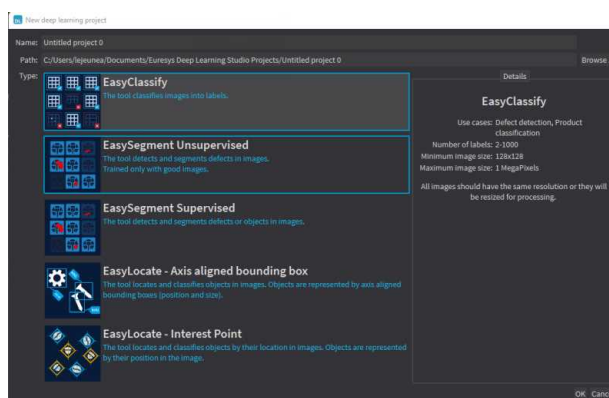
The **Deep Learning Studio** is available in the installation folder of **Open eVision**.

Deep Learning Studio workflow illustration

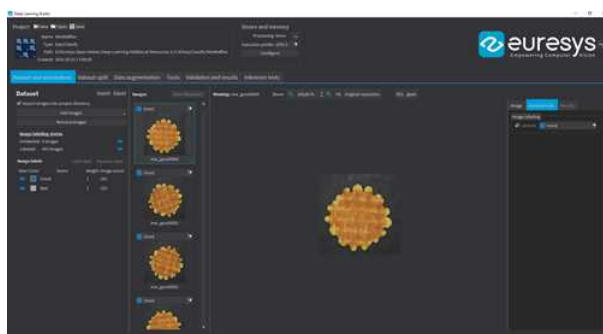
1. Load or create a project.
 - Quickly go back to your work with the recent project lists.
 - Open existing projects or create new projects.



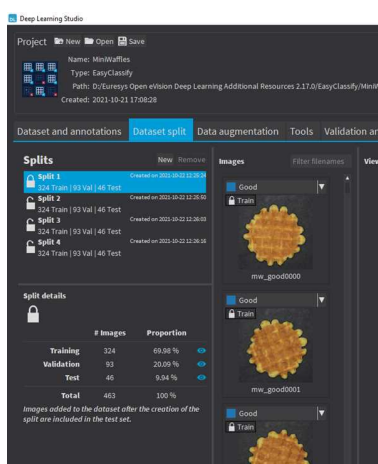
- Create a new project.
 - Specify the name and the type of the project.
 - If necessary, change the path to save the project.



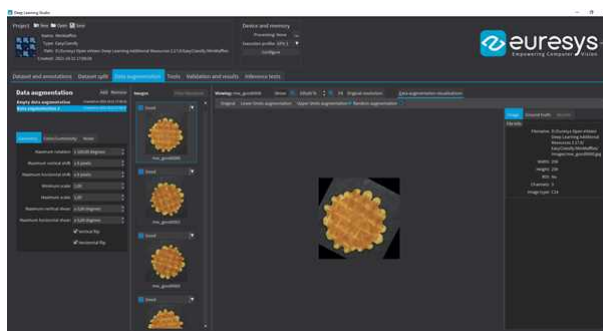
- 2. Import images in the project dataset and annotate the images.



- 3. Split your dataset into training, validation, and test images (for more details, refer to "Managing the Dataset Splits" on page 80).



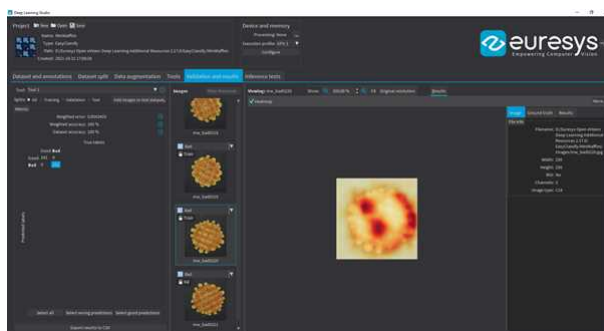
- 4. Configure and view the data augmentation settings (for more details, refer to "Using Data Augmentation" on page 83).



5. Create, configure, and train your tools

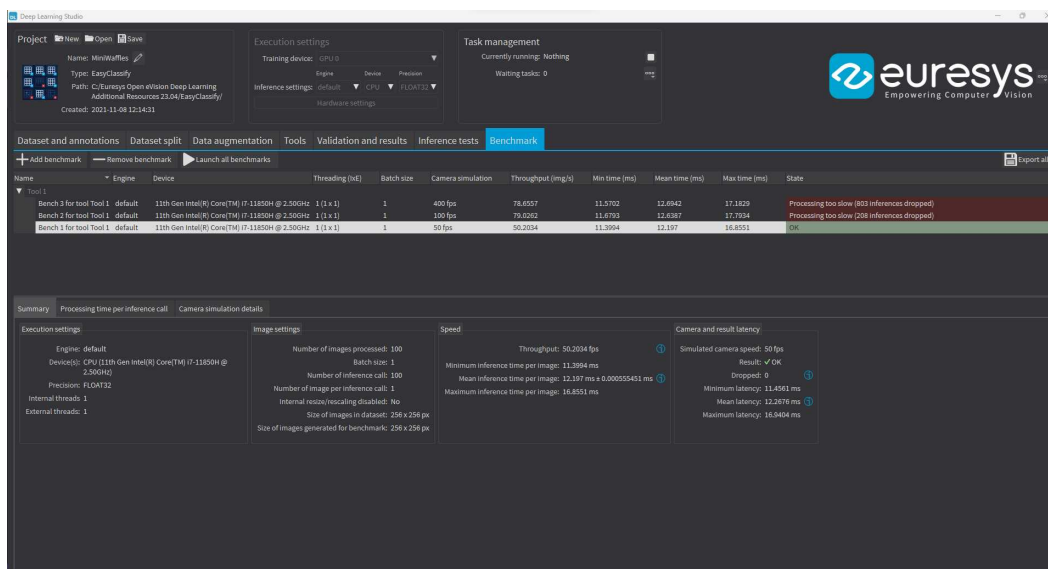


6. Analyze the performance and the results of the trained tools.



7. Benchmark your tools to optimize the execution settings and maximize the inference speed on your hardware.

See "Benchmarking a Deep Learning Tool" on page 90.



Resources and code snippets

- The **Deep Learning Additional Resources** package, separate from the **Open eVision** installer, provide several sample datasets as well as **Deep Learning Studio** projects containing a trained tool for these datasets:
 - For **EasyClassify**: the MiniWaffle, Stone Tiles and CopperSilverTape datasets.
 - For **EasySegment Unsupervised**: the Fabric dataset.
 - For **EasySegment Supervised**: the Coffee dataset.
 - For **EasyLocate Axis Aligned Bounding Box**: the ElectronicComponentsBag dataset.
 - For **EasyLocate Interest Point**: the CeramicCapacitor dataset.
- Some sample programs in the folder Sample Programs show how to train and use a deep learning tool.
- Some [code snippets](#) are also available for illustration and reference.

Engines and Hardware Support (CPU/GPU)

Engines

- Starting with **Open eVision 23.04**, running (inference) or training a **Deep Learning** tool is done through an engine. The engines are specified through their names.
 - To set an engine for a tool, use `EDeepLearningTool.Engine`
 - To list the available engines, use `EDeepLearningTool.AvailableEngines`
- An engine can support different devices:
 - To list the detected devices for the current engine, use `EDeepLearningTool.AvailableDevices`
 - To list the detected devices for another engine, use `EDeepLearningTool.GetAvailableDevicesForEngine`. Note that this actually loads the engine in memory.
 - The devices are represented by the class `EDeepLearningDevice` or directly through their name (`EDeepLearningDevice.Name`) that are guaranteed to be unique for a given engine.
 - To set a device, use `EDeepLearningTool.ActiveDevices` or `EDeepLearningTool.ActiveDevicesByName`. You can pass multiple devices for multi-GPU training for example.
 - To know if a device supports inference and/or training, use `EDeepLearningDevice.HasInferenceCapability` and `EDeepLearningDevice.HasTrainingCapability`. Check the matrix below for each engine and device type support.
- A device or an engine can support various inference precision (FLOAT32 or FLOAT16).
 - By default, the precision is FLOAT32.
 - Use `EDeepLearningTool.InferencePrecision` to set or retrieve the current precision.
 - A lower precision (for example FLOAT16 instead of FLOAT32) can improve the speed.
- The main engine is named default:
 - It is always available and always supports the CPU of the computer on which it is running.
 - It always supports both inference and training.

- The other engines:
 - They are mainly used to increase the inference speed.
 - The other engines and the support for a GPU are provided by the package named deep-learning-redist (previously cuda-redist).

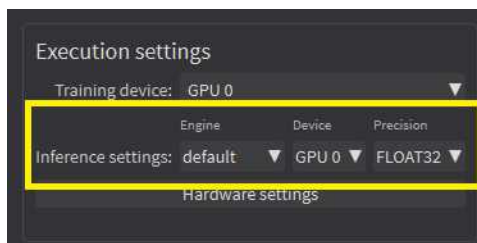
Engine name	Supported devices	Supported platforms	Training	Inference	Inference precision
default	CPU NVIDIA GPU *	All **	Yes	Yes	FLOAT32
EasyDeepLearningEngine_OpenVINO *	CPU Intel GPU	Windows 64-bit	No	Yes	FLOAT16 FLOAT32
EasyDeepLearningEngine_TensorRT *	NVIDIA GPU	Windows 64-bit Linux x64 Linux aarch64	No	Yes	FLOAT16 FLOAT32

* Requires the deep-learning-redist package

** The Windows 32-bit support is only for CPU and the limitation to 2 GB of the application memory can be a problem for the training or the inference on large images.

In Deep Learning Studio

- In the Execution settings panel, select the Engine, Device and Precision that you want to use for inference.



Legacy API for selecting CPU and GPU

- The API calls `EDeepLearningTool.EnableGPU` and `EDeepLearningTool.GPUIndexes` are deprecated.
- They can only be used with the default engine, otherwise, an exception is thrown.

Additional consideration for NVIDIA CUDA® GPU

TIP
Using a recent **NVIDIA** GPU greatly accelerates the processing speeds. Refer to the benchmarks for each tool type to compare the GPU and CPU speeds.

To use an **NVIDIA** GPU with the Deep Learning tools, install the deep-learning-redist package for your operating system. It is not recommended to use a system-wide installation of the **CUDA** libraries for GPU support.

1. To use an **NVIDIA** GPU with the **Deep Learning** tools, install the `deep-learning-redis` package for your operating system.

NOTE: It is not recommended to use a system-wide installation of the CUDA libraries for the GPU support.

In this version of **Open eVision**, the GPU acceleration is based on:

- For **Windows** and **Linux Intel x64**:
 - **NVIDIA CUDA® Toolkit** version v11.8
 - **NVIDIA CUDA® Deep Neural Network** library (**cuDNN**) v8.6
- For **Linux ARM (aarch64 Jetson)** platforms:
 - The **CUDA** and **cuDNN** packages distributed with the platforms.

NOTE: The following versions have been tested:

- **JetPack 4.6 (L4T 32.6.1)**
- **JetPack 5.0 (L4T 34.1)**
- **JetPack 5.1.2 (L4T 35.4.1)**

2. Check that you have or install the up-to-date **NVIDIA** drivers.

 Refer to your GPU documentation to install recent drivers for your operating system.

3. For **Linux ARM (aarch64 Jetson)** platforms, install the **NVIDIA JetPack SDK 4.6** minimum that includes the **NVIDIA Jetson Linux Driver Package (L4T) 32.6**.

Starting from **NVIDIA JetPack SDK 4.3**, it is possible to upgrade without flashing the device to the version 32.6 of the **NVIDIA Jetson Linux Driver Package**:

- Edit the file `/etc/apt/sources.list.d/nvidia-l4t-apt-source.list` so that its content is:

```
deb https://repo.download.nvidia.com/jetson/common r32.6 main
deb https://repo.download.nvidia.com/jetson/t194 r32.6 main
```

- Upgrade the system using `apt`:

```
$ sudo apt update
$ sudo apt upgrade
```

4. Use the engine default for training and inference or the engine `EasyDeepLearningEngine_TensorRT` for inference only.



TIP

For recent **NVIDIA** GPUs or **Jetson Orin** boards (**NVIDIA** GPUs with a compute capability higher than 8.0 on **Intel x64** and 7.2 on **aarch64**), using **Deep Learning** triggers a long initialization (over a minute).

To avoid this initialization each time you use **Deep Learning**, we recommend to increase the **CUDA** cache max size to 1024 MB:

- On Linux, set the environment variable `CUDA_CACHE_MAXSIZE` to `1073741824` before launching **Deep Learning Studio** or your program.

In a terminal, this means executing:

```
$ CUDA_CACHE_MAXSIZE=1073741824 /path/to/your/program
```

- On Windows, launch the **Advanced System settings**, go to the **Environment variables** dialog and add or modify the `CUDA_CACHE_MAXSIZE` variable with the value `1073741824`.

Using multiple GPUs

You can use multiple **NVIDIA** GPUs with the default engine for the training and the batch classification by specifying multiple **NVIDIA** GPUs devices with [EDeepLearningTool.ActiveDevices](#).

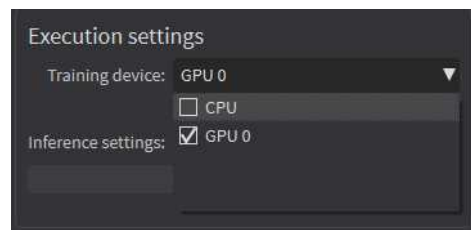
NOTE: Be careful to only set multiple GPUs that have similar performances: the performances are limited by the slowest device.



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 does not yield the same performance gain.

- In **Deep Learning Studio**, to choose the training devices, check all the devices that you want to use for training.



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

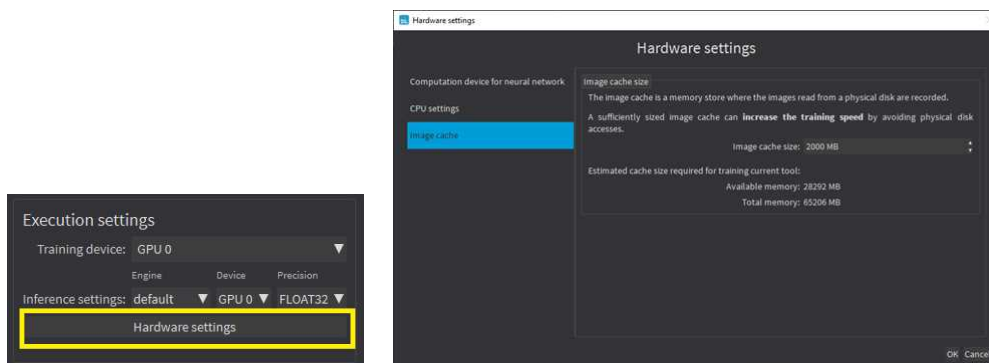
Image cache

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

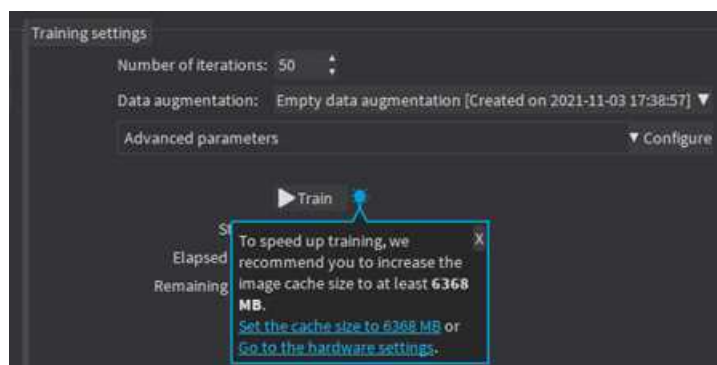
- The default size is 1 GB.
- The training speed is higher when the image cache is big enough to hold all the images of your dataset.
- With dataset too big to fit in the image cache, we recommend using a SSD drive to hold your images and project files as a SSD drive improves the training speed.

To specify the cache size in bytes:

- In the API, use the `EDeepLearningTool::SetImageCacheSize` method.
- In **Deep Learning Studio**, click on the **Hardware settings** button in the **Execution settings** panel and select **Image cache** in the menu.



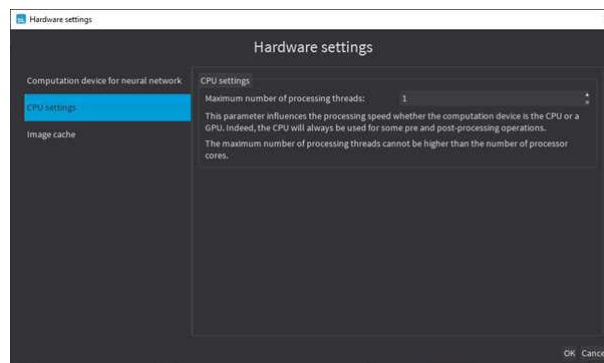
- When there is enough memory to increase the image cache so that it can hold all the images in the dataset, **Deep Learning Studio** displays a recommendation next to the training button.
 - Click on the recommendation to change the image cache size and improve the training speed.



Multicore processing

The deep learning tools support multicore processing with the engines default and EasyDeepLearningEngine_OpenVINO (see "[Multicore Processing](#)" on page 1):

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



Managing the Dataset and the Annotations

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_BW16](#)
 - [EImageType_C24](#)

- The supported image size depends on the type of the deep learning tool.
 - **EasyClassify** and **EasyLocate** require that all images have the same size. Images that do not have the size configured for the tool are automatically resized before being processed by the neural network.
 - **EasySegment** splits images into patches. As such, the tools can process images of different size and the images are processed at their native resolution.
 - In all cases, your dataset should cover the variability of sizes that you want to process in production.

File formats

The supported standard file formats for the dataset and image annotation are:

- The COCO Json for **EasySegment Supervised** and **EasyLocate Axis Aligned Bounding Box** annotations.
 - A COCO json file contains the annotation for a dataset (several images).
 - In **Deep Learning Studio**, use the **Import** feature to import COCO datasets.
 - 📄 See <https://cocodataset.org/#format-data> for a description of the COCO Json dataset format.
- The YOLO TXT annotation format for **EasyLocate Axis Aligned Bounding Box** annotations.
 - For each image, the annotations are in a file with the same filename as the image and the .txt extension.
 - In **Deep Learning Studio**, if the annotation files are located in the same folder as their corresponding images, use the **Add images** feature to import the images and their annotations.
 - 📄 See <https://pjreddie.com/darknet/yolo> for the original YOLO source code.
- PASCAL VOC XML annotation for **EasyLocate Axis Aligned Bounding Box** annotations.
 - For each image, the annotations are in a file with the same filename as the image and the .xml extension.
 - In **Deep Learning Studio**, if the annotation files are located in the same folder as their corresponding images, use the **Add images** feature to import the images and their annotations.
 - 📄 See <http://host.robots.ox.ac.uk/pascal/VOC> for resources on the PASCAL VOC XML annotations.

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**.

NOTE: **Deep Learning Studio** only displays the labels for the tool type of the project.




- 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.

Image labeling status

Unlabeled: 0 images 


Labeled: 605 images 

Image labels Add label Remove label









View	Color	Name	Weight	Image count
	■	Good	1	320
	■	Glue	1	106
	■	Damaged	1	83
	■	Broken	1	96

Image segmentation status



No segmentation: 0 images 

With segmentation: 189 images 


- Background / No foreground blobs / Good: 57 images 


- With foreground blobs / Defective: 132 images 


Segmentation labels Add label Remove label


View	Color	Name	Weight	Image count	Pixel count
	■	Backg...	1	189	433612770
	■	Defect	1	132	1843230

Object labeling status






Unlabeled: 0 images 

Labeled: 335 images 

- No objects: 2 images 

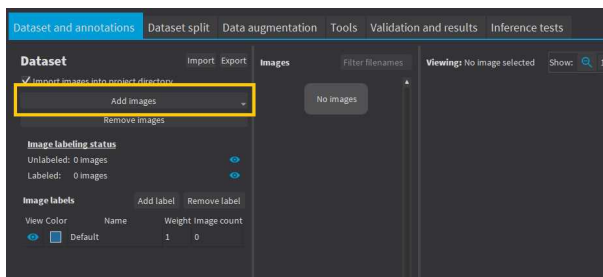
- With at least 1 object: 333 images 

Object labels Add label Remove label

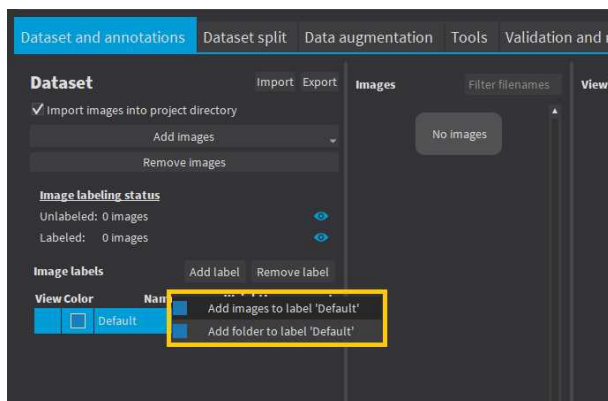
View	Color	Name	Weight	Image count	Object count
	■	Transis...	1	156	320
	■	Diode	1	155	324
	■	C1	1	151	299
	■	C2	1	173	343
	■	C3	1	135	287

Adding Images

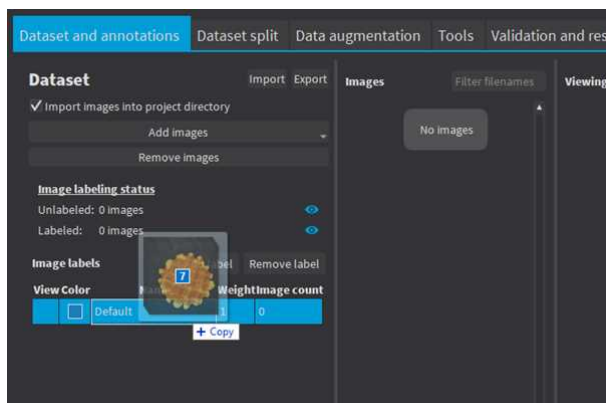
- In **Deep Learning Studio**, add image files (PNG, TIFF, JPEG, BMP and J2K types) to your datasets in one of the following ways:
 - Click on the **Add images** button to add images without any label nor segmentation.



- Right-click on an image label and click **Add images to label** or **Add folder 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.

**TIP**

If there is an .txt or .xml file with the same filename as the image next to it, **Deep Learning Studio** tries to load these files as YOLO TXT or PASCAL VOC XML annotations.

- 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.
- Import YOLO TXT and PASCAL VOC XML annotations with `EClassificationDataset::ImportYOLOTXTAnnotations(imgId)` or `EClassificationDataset::ImportPascalVOCXMLAnnotations(imgId)`.
 - Check the available annotations for a file with `EClassificationDataset::GetAvailableImageAnnotationFormat(filepath)`.

**TIP**

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

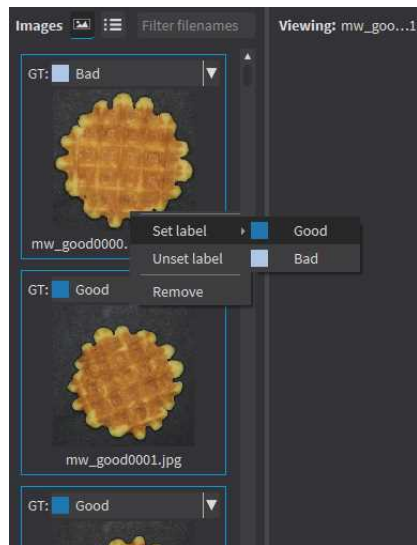
**NOTE**

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

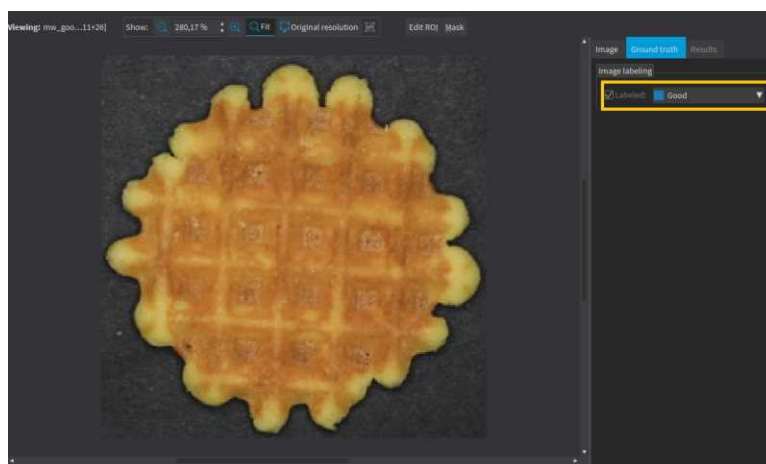
Editing the Label of an Image

In Deep Learning Studio:

- To change the label of images:
 - a. Select one or more images in the image list.
 - b. Right click on the selection.
 - c. Click on **Set label** and select the new label.



- To change the label of a single image:
 - a. Click on the image in the list to open the image in the viewer.
 - b. In the **Ground truth** tab on the right, select the new label.

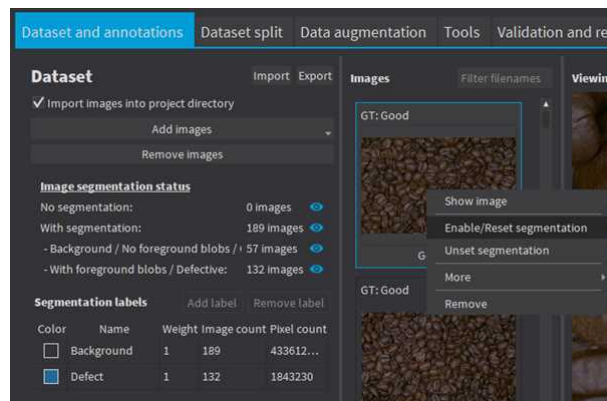


- c. Or use the keyboard keys **F1** ... **F10** to assign the 1st ... 10th label to the image.

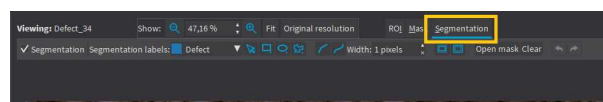
Editing the Segmentation of an Image

In Deep Learning Studio:

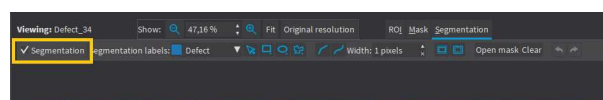
- To initialize or reset the segmentation of an image to all Background pixels:
 - Select one or more images in the image list.



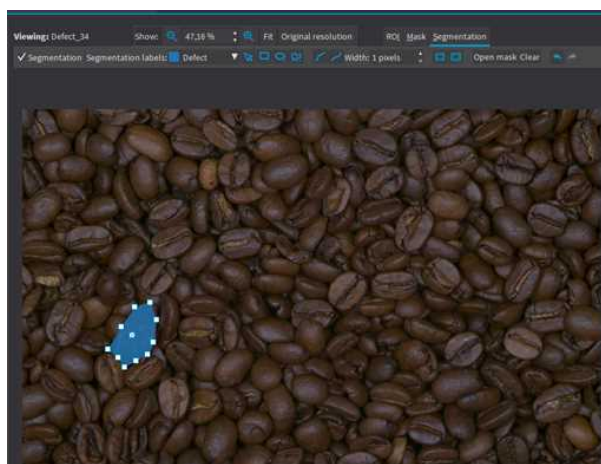
- Right click on the selection.
 - Click on **Enable/Reset segmentation**.
- To edit the segmentation:
 - Double click on the image to open it in the image editor.
 - Click on the **Segmentation** button (**ALT + S**).



- To enable or unset the segmentation, check or 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



- The following drawing tools are available:

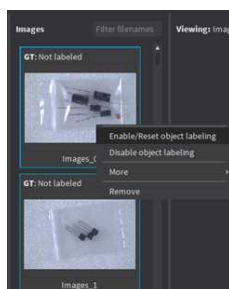
- Rectangle**: click and drag to create a rectangle.
 - Ellipse**: click and drag to create an ellipse.
 - Polygon**: click to create new vertices and double-click to close the polygon. When the polygon is closed, double click on an edge to add a new vertex.
 - Assisted segmentation**: click and drag to draw a rectangle around the zone you want to segment. To refine the segmentation, mark the pixels (click and drag) to include in the segmentation.
 - Lines**: click and drag to draw a line with a pen of the specified width.
 - Free-form drawing**: click and draw any shape with a pen of the specified width.
 - Eraser**: same as free-form drawing but it assigns a background label instead of the selected segmentation label.
 - Shrink**: shrinks the blobs of the selected segmentation label.
 - Grow**: grows the blobs of the selected segmentation label.
- ▶ Except for the eraser tool, all the tools operate on the label specified in the segmentation tool bar.

Editing the Objects of an Image

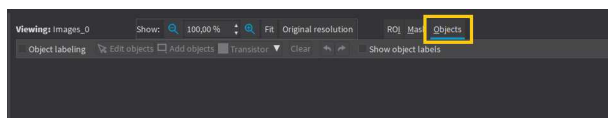
In Deep Learning Studio:

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

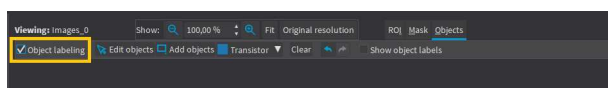
- a. Select one or more images in the image list.



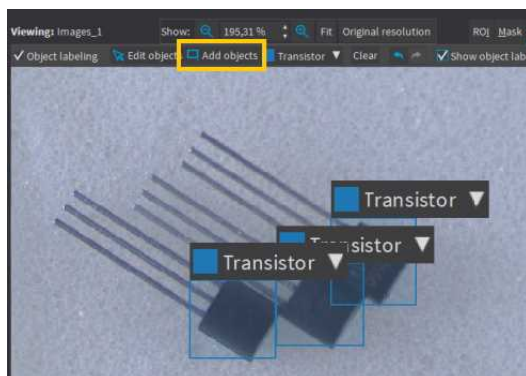
- b. Right click on the selection.
- c. Click on **Reset object labeling**.
- To edit the objects:
 - a. Double click on the image to open it in the image editor.
 - b. Click on the **Objects** button (**ALT + O**).



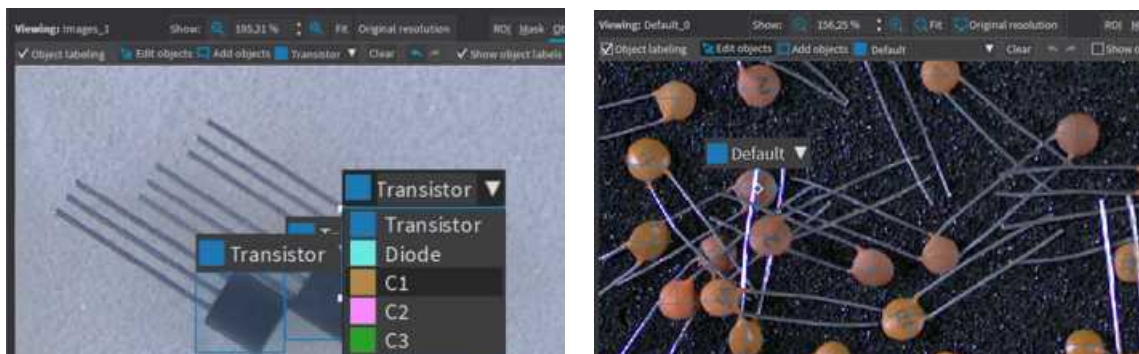
- c. To reset or unset the object labeling, uncheck the **Object labeling** checkbox (**CTRL + L**).



- 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 label of an object and the bounding box or the position, according to the **EasyLocate** mode.



ROI and Mask

Setting a ROI

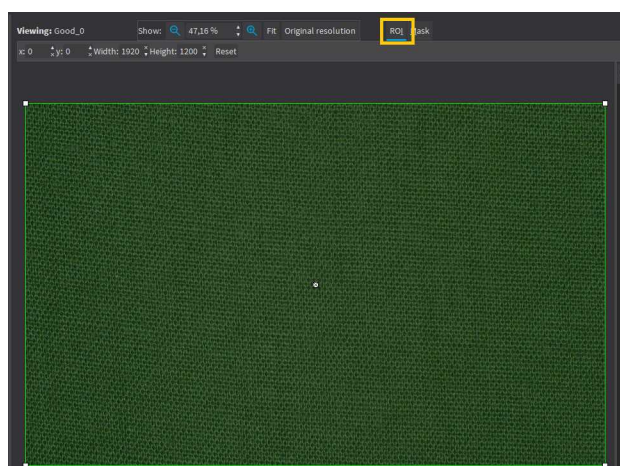
Use an ROI (region of interest) to crop an image or a whole dataset to a rectangular area aligned with the axis.

In the API:

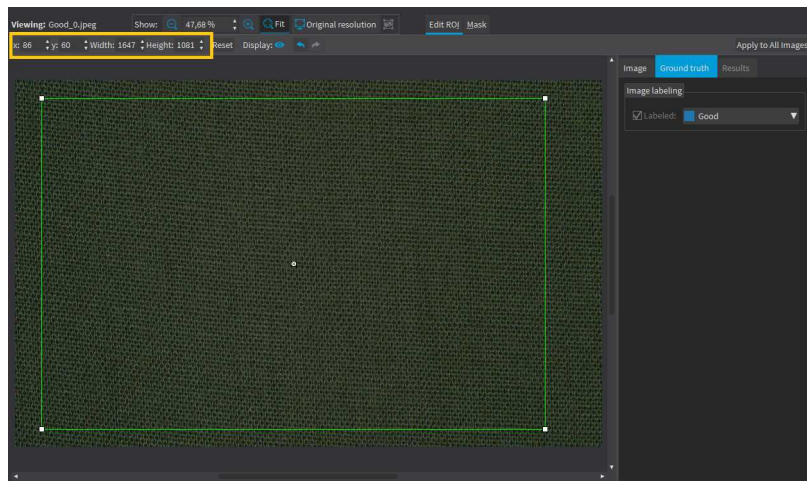
- To define an ROI for an image:
 - Specify the ROI when you add the image to the dataset.
 - Or use `EClassificationDataset::SetRegionOfInterest`.

In Deep Learning Studio:

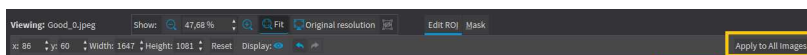
- To change the ROI:
 - a. Select an image from the dataset.
 - b. Click on the **ROI** button (**ALT+I**).



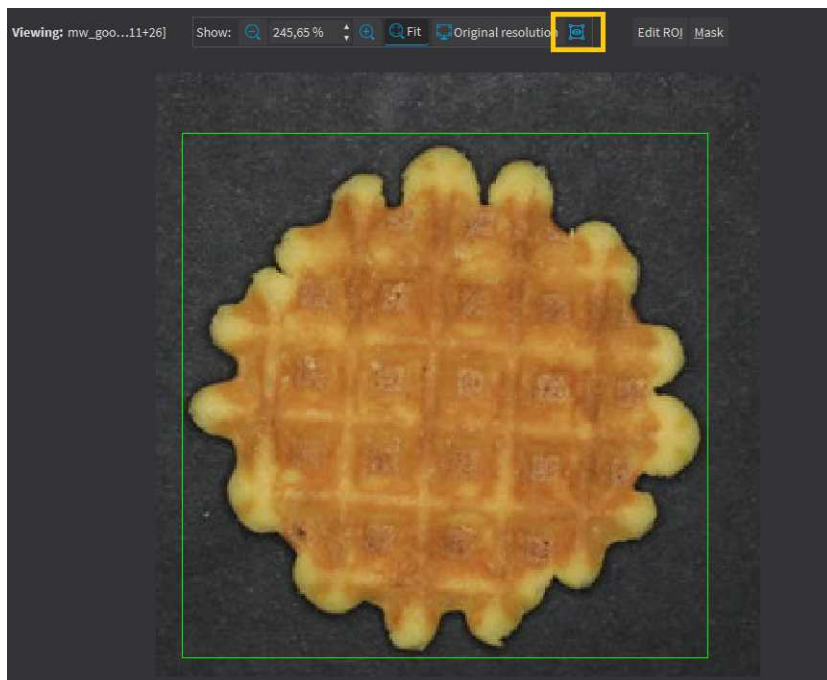
- c. Drag the ROI green box, or directly set the ROI origin (x and y), Width and Height.



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



- To visualize the ROI within its parent image:
 - Click on the button.



Setting a mask

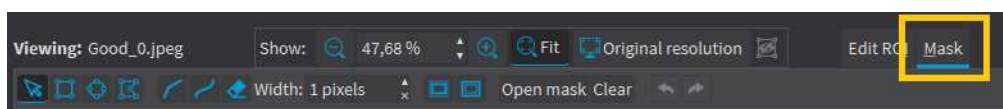
Set a mask on an image in a dataset to remove the pixels in the mask area from any computation. The mask works as a “don’t care area”.

In the API:

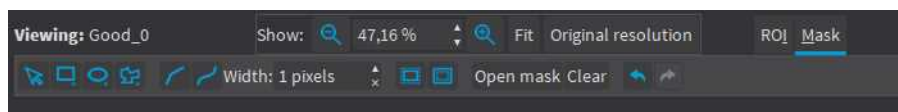
- To define a mask for an image:
 - Specify the mask when you add the image to the dataset.
 - Or use `EClassificationDataset::SetMask`.

In Deep Learning Studio:

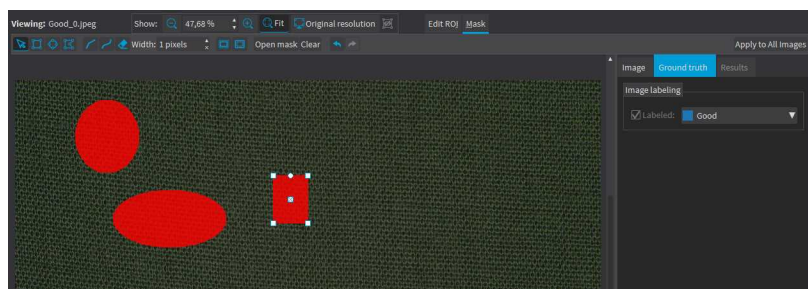
- To change the mask:
 - a. Select an image from the dataset.
 - b. Click on the **Mask** button (**ALT+M**).



- c. Select a drawing tool to draw the mask.



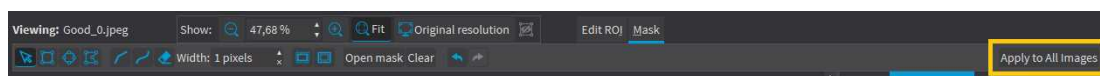
- d. Draw the mask.



TIP

Click on the **Open mask** button to use an image to specify a mask. All the pixels of the image (such as an EROI BW8) 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).

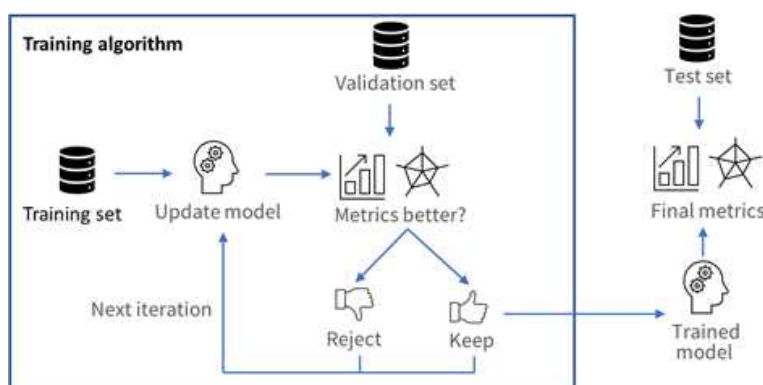


Managing the Dataset Splits

It is important to split the dataset into 3 sets:

- The *training set* contains the images that are used during training to update and optimize the deep learning model.
- The *validation set* contains the images that are used during training to select the model that gives the best performance.
- The *test set* contains images that are not used during training and that are used to evaluate the final performance of your classifier.

The following picture shows how and when each set is used.



WARNING

These sets MAY NOT contain:

- Images of the other sets.
- Images of an object for which there are other images in other sets.

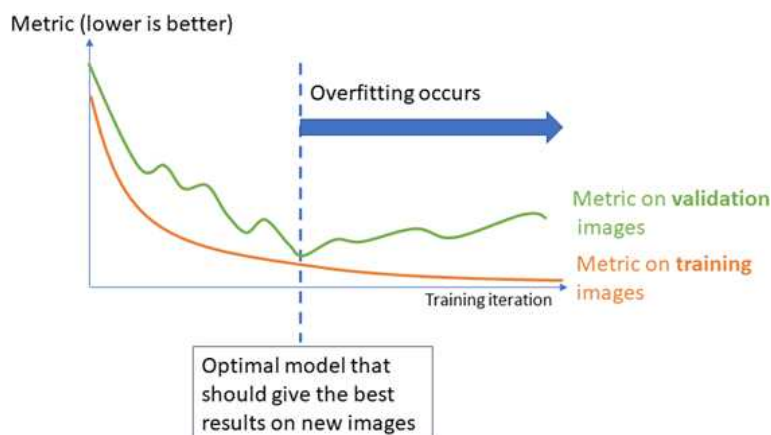
Why is it important?

Deep learning techniques can suffer from overfitting; this means that the trained classifier is too focused on the specific images present in the training set and it is not able to learn a general model of your data. For example, an overfitted model can learn to recognize the exact images present in the training set and not the underlying defects that you want to detect. Such tools perform poorly in production.

The validation set 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 set. Without the validation set, it is impossible to know if a tool that performs well on its training set can also perform well in production.

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

The training algorithm is designed to avoid overfitting by keeping the model at the iteration that gives the highest performance on the validation set.



To minimize overfitting and increase the performances:

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



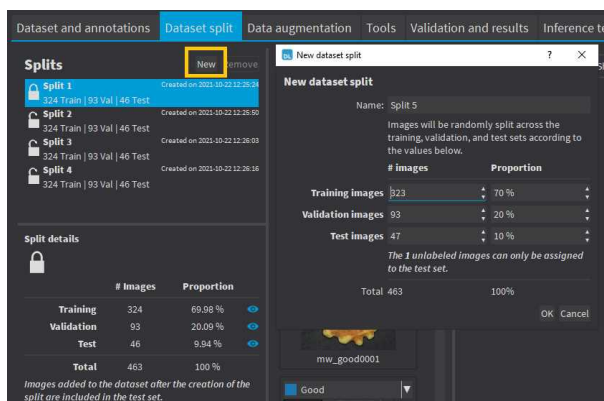
TIP

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.

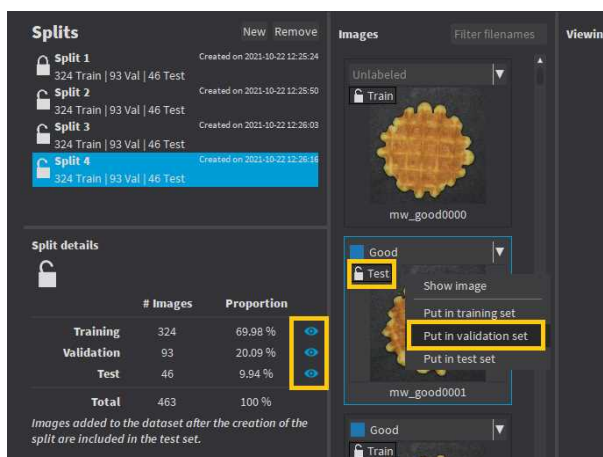
Splitting the dataset in Deep Learning Studio

In **Deep Learning Studio**, open the **Dataset split** tab:

1. To create new dataset splits, click on **New**.
 - A new split is created randomly according to the specified proportion or number of images.
 - The default proportion of images is 70% of training images, 20% of validation images and 10% of test images.
 - Images without a proper annotation must be in the test set.



2. The dataset splits can be locked or unlocked . A dataset split is locked when it was used to train a tool. When a dataset split is locked:
 - You cannot move images out of the training or validation sets.
 - You can only move images from the test set to either the training set or the validation set.
3. Select a dataset split to display the set of the images.
 - Use the icons or to filter the image list according to the set type.
 - If an image is unlocked , you can move it to another split by clicking on its split or by right-clicking on the image and selecting the set.



4. Double-click on a split to change its name.

NOTE
 Images added to the dataset after you have created a split are automatically assigned to the test set of that split.

TIP
 We recommend to experiment with several different splits to see how the training behaves with different training sets.

Splitting the dataset in the API

- Create directly `EClassificationDataset` objects for your training, validation and test sets.
- Randomly split an `EClassificationDataset` dataset into a training set and a validation set 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)`
- Or randomly split an `EClassificationDataset` dataset using `EClassificationDataset::GetSplit` to get a `EDatasetSplit` object.
 - Use `EClassificationDataset::ExtractSplit` with the `EDatasetSplit` object to get a `EClassificationDataset` object containing only the images of a given type

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).

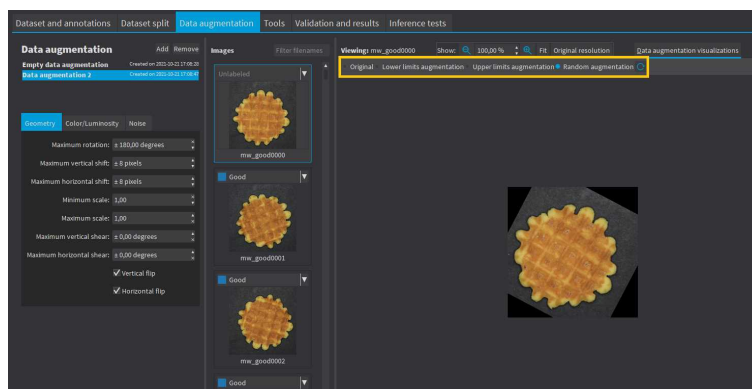


NOTE

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

- Create and configure the data augmentation settings in the **Data augmentation** tab.
- 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**).

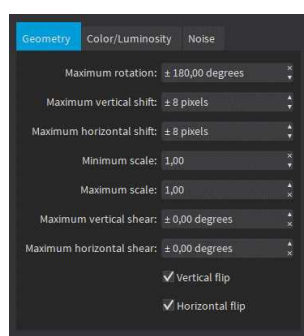


In the API

Use `EClassificationDataset::SetEnableDataAugmentation(true/false)` to enable or disable these transformations or directly use an object `EDataAugmentation` that you give to the method `EDeepLearningTool::Train`.

The transformations

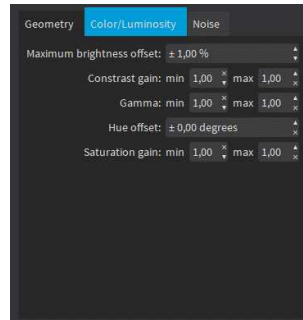
Geometric transformations



- 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)`)

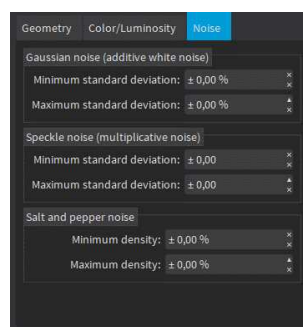
- Rotations (between 0 and a maximum value defined with `EClassificationDataset::SetMaxRotationAngle`)
- Horizontal and vertical shear (between $-\text{maxValue}$ and maxValue defined with `EClassificationDataset::SetMaxHorizontalShear` and `EClassificationDataset::SetMaxVerticalShear`)

Color and luminosity transformations



- Brightness offset (between $-\text{maxValue}$ 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 $-\text{maxValue}$ and maxValue defined with `EClassificationDataset::SetMaxHueOffset`)
- Saturation gain (between a minimum and maximum value defined with `EClassificationDataset::SetMinSaturationGain` and `EClassificationDataset::SetMaxSaturationGain`)

Noise transformations





TIP

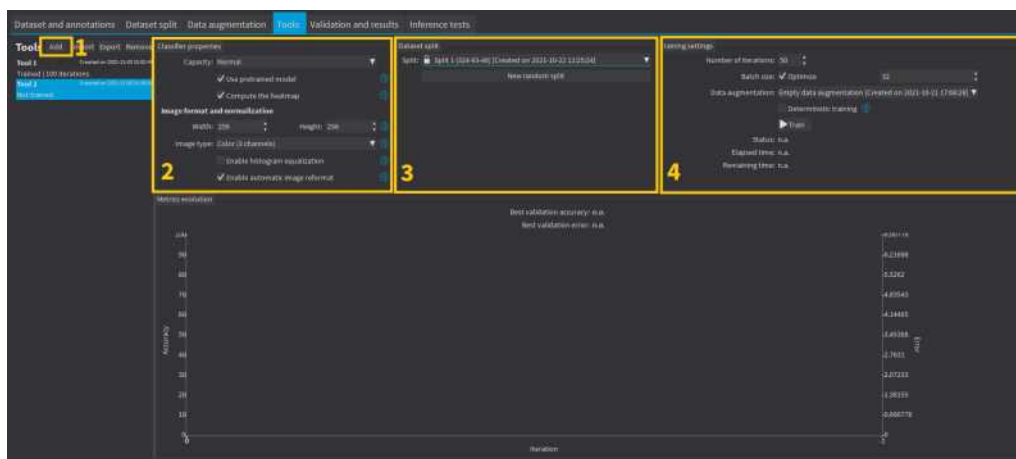
The standard deviation is expressed as a percentage of the maximum pixel value.

- 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 Deep Learning Studio

1. Create a new tool.
2. Configure the tool settings.
3. Select the dataset split to use for this tool.
4. Configure the training settings and click on `Train`.



In the API

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

The training settings

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

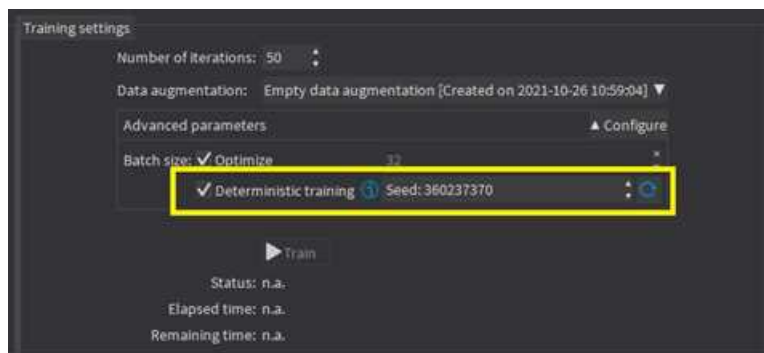


TIP

We recommend to use more iterations than the default value:

- For **EasyClassify** without pretraining, **EasyLocate** and **EasySegment**.
 - For smaller dataset because the training is harder (for example, 100 iterations for a dataset with 100 images, 200 iterations for a dataset with 50 images, 400 iterations for a dataset with 25 images...).
- 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.
 - The *Data augmentation*.
 - Whether to use *Deterministic training* or not.
 - The deterministic training allows to reproduce the exact same results when all the settings are the same (tool settings, dataset split and training settings).
 - The deterministic training fixes random seeds used in the training algorithm and uses deterministic algorithms.
 - The deterministic training is usually slower than a non-deterministic training.

- In **Deep Learning Studio**, the option to use deterministic training and the random seed are available in the advanced parameters.



- In the API, use `EDeepLearningTool::EnableDeterministicTraining.` and `EDeepLearningTool::DeterministicTrainingRandomSeed.`

Continue the training

You can continue to train a tool that is already trained.

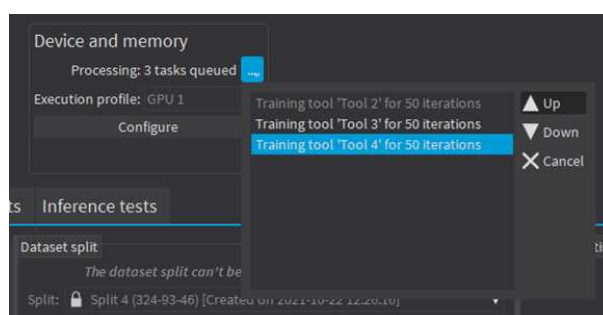
In **Deep Learning Studio**, the dataset split associated with a trained tool is locked.

- You can only continue training a tool with the same dataset split.
- You can still add new training or validation images to the split by moving test images to the training set or the validation set of that split.

Asynchronous training

The training process is *asynchronous* and performed in the background.

- In **Deep Learning Studio**:
 - The training processes are queued.
 - They are automatically executed one after the other.
 - You can manually reorder the training in the processing queue.



- In the API:
 - `EDeepLearningTool::Train` launches a new thread that does the training in the 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.

Using a Deep Learning Tool

In the API

- The API to use a **Deep Learning** tool on a new image or on a set of images is different for each tool:
 - **EasyClassify** > "Classifying New Images" on page 101
 - **EasySegment** > Unsupervised > "Applying the Tool to New Images" on page 114
 - **EasySegment** > Supervised > "Using the Supervised Segmenter" on page 120
 - **EasyLocate** > "Locating Objects" on page 135
- You can apply each tool to individual images or to sets of images.
 - With a set of images and with GPU processing enabled, the tool processes `EDeepLearningTool::BatchSize` at the same time.
 - **EasySegment** can automatically split individual images in sub-images according to the patch size, the scaling and the sampling density parameters. These sub-images are processed by batch.

NOTE: The processing of the first prediction after loading a model, changing the batch size or with a different number of images (smaller than the batch size), is slower than the previous one because the tool must perform various initializations that can be very slow.

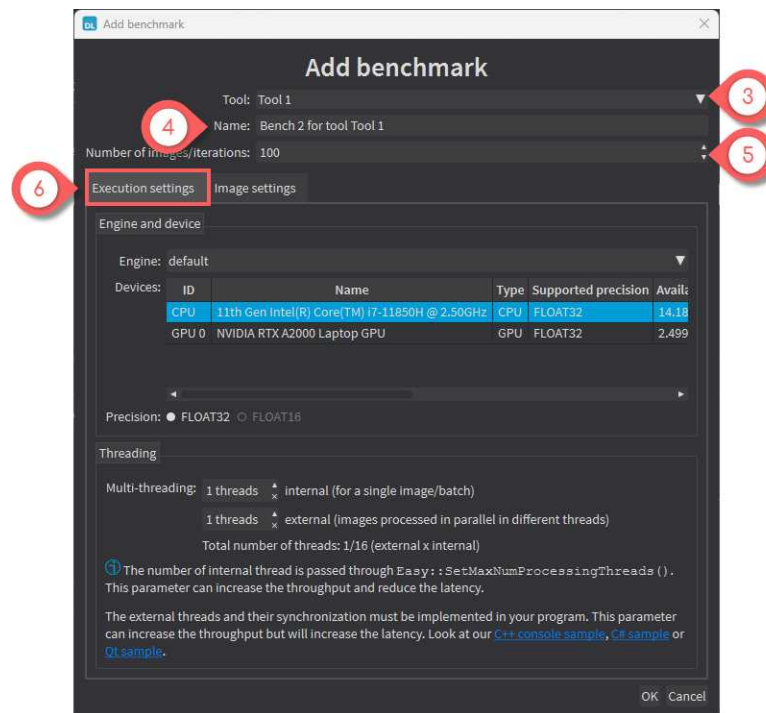
- Use the methods `EDeepLearningTool::InitializeInference` with an image or a set of images to perform these initializations before actually using the tool.

Benchmarking a Deep Learning Tool

In **Deep Learning Studio**, if you have a **Deep Learning** license, you can benchmark your trained tools to measure their speed according to various execution parameters.

The screenshot shows the **Benchmark** tab in the software interface. The top navigation bar includes tabs for **Dataset and annotations**, **Dataset split**, **Data augmentation**, **Tools**, **Validation and results**, **Inference tests**, and **Benchmark**. The **Benchmark** tab is active and highlighted. Below the navigation bar, there are three buttons: **+ Add benchmark**, **- Remove benchmark**, and **▶ Launch all benchmarks**. The **+ Add benchmark** button is circled in red with the number 2. Below the buttons is a table with the following columns: **Name**, **Engine**, **Device**, **Threading (txE)**, **Batch size**, **Camera simulation**, **Throughput (in**, **Min time (ms)**, **Mean time (ms)**, **Max time (ms)**, and **State**. The **Throughput (in** column is circled in red with the number 1. The table contains one row with the following data: **Name**: for tool Tool 1, **Engine**: default, **Device**: NVIDIA RTX A2000 Laptop GPU - 1 (1 x 1), **Threading (txE)**: 1, **Batch size**: 1, **Camera simulation**: No, **Throughput (in**: 134,986, **Min time (ms)**: 4.6873, **Mean time (ms)**: 7.39838, **Max time (ms)**: 15.2188, and **State**: Done. Below the table, there is a **Summary** section with a sub-tab **Processing time per inference call**. The summary is divided into three columns: **Execution settings**, **Image settings**, and **Speed**. The **Execution settings** column lists: Engine: default, Device(s): GPU 0 (NVIDIA RTX A2000 Laptop GPU), Precision: FLOAT32, Internal threads: 1, and External threads: 1. The **Image settings** column lists: Number of images processed: 200, Batch size: 1, Number of inference call: 200, Number of image per inference call: 1, Internal resize/rescaling disabled: No, Size of images in dataset: 256 x 256 px, and Size of images generated for benchmark: 256 x 256 px. The **Speed** column lists: Throughput: 134,986 fps, Minimum inference time per image: 4.6873 ms, Mean inference time per image: 7.39838 ms ± 0.00494823 ms, and Maximum inference time per image: 15.2188 ms.

1. Open the **Benchmark** tab.
2. To add a benchmark, click on the **Add benchmark** button.



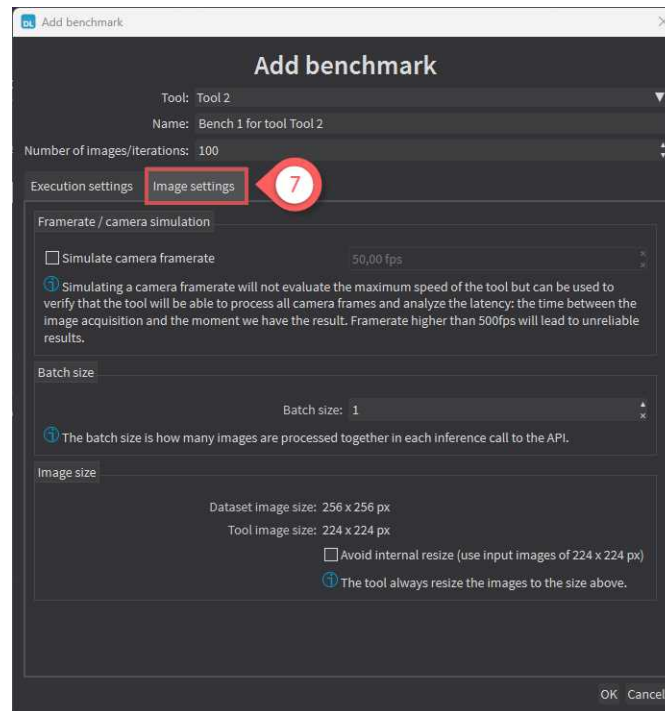
In the **Add benchmark** window:

3. Select the **Tool** that you want to benchmark. Only trained tool appear in the list.
4. **Name** your benchmark.
5. Set the **Number of images/iterations** to perform in the benchmark. The more images, the better the precision.
6. In the **Execution settings** tab, set the parameters that control how the neural network runs:
 - The engine, device and inference precision.
 - The threading parameters.



Use the threading parameters when running the neural network on the CPU:

- The internal threading can improve the processing speed for a single image. It also improves both the throughput (number of images processed by seconds) and the latency (time taken to process one image).
- The external threading involves using the same tools in different threads on different images. This can greatly improve the throughput but it decreases the latency (it means that the number of images processed each seconds increases but the time to process a single image is longer on average).
- Check the code samples referenced in the dialog for how to implement internal and external multithreading.



7. In the **Image settings** tab, set the parameters that control how the images are preprocessed and given to the neural network:
- **Simulate camera framerate**: When checked, the benchmark does not try to perform inference at the maximum speed. Instead, images are produced at the specified frame rate.
Use this setting to check if your execution settings can match the speed of your camera. Simulating a camera frame rate also gives you an estimation of the time spent between the acquisition of an image and the moment the result is available.
 - **Batch size**: Increasing the batch size can greatly improve the throughput, especially for GPU processing. However, it also increases the latency.
 - **Image size**: By default, when applying a tool on an image that has a size different from the input size of the tool, the image is automatically resized by the tool. The time spent performing this resizing is part of the inference time.
Check **Avoid internal resize** to avoid the resizing performed during inference and slightly accelerate the tool.
In a real application, you can configure your camera to capture images directly of the input size of the tool to avoid this automatic resizing step.

Benchmark results

The following benchmark results are available:

- In the **Summary** tab:
 - The settings of the benchmark.
 - The throughput (number of images processed by seconds).
 - The minimum/ mean / maximum inference time per image.

- In the **Processing time per inference call**:
 - A table and a graph with the processing times.

NOTE: An inference call means calling the inference API with a batch of images (see batch size setting).



- In addition in the **Summary** tab, if you are simulating a camera frame rate:

Camera and result latency

Simulated camera speed: 50 fps

Result: ✔ OK

Dropped: 0 ⓘ

Minimum latency: 7.0608 ms

Mean latency: 8.92305 ms ⓘ

Maximum latency: 11.7943 ms

- Whether all images could be processed in time.
- The number of images that were dropped because the frame rate was too fast compared to the inference speed.
- The minimum/ mean / maximum latency (the time between the acquisition of the image and the moment the result for this image is available).
- The camera simulation details tab that contains a table showing the various timings for each image (acquisition time, inference start time, inference end / latency, inference time).

Summary		Processing time per inference call		Camera simulation details	
# Image	Acquisition	# Inference	Inference start	Inference end / Latency	Inference time
1	0 ms	1	+ 0.0341 ms	+ 7.3899 ms	7.3558 ms
2	20.6381 ...	2	+ 0.1175 ms	+ 9.9168 ms	9.7993 ms
3	40.0852 ...	3	+ 0.1584 ms	+ 9.8796 ms	9.7212 ms
4	60.1939 ...	4	+ 0.1463 ms	+ 9.8619 ms	9.7156 ms
5	79.2444 ...	5	+ 0.0438 ms	+ 7.131 ms	7.0872 ms
6	99.9194 ...	6	+ 0.1281 ms	+ 9.504 ms	9.3759 ms
7	120.171 ...	7	+ 0.1129 ms	+ 7.7202 ms	7.6073 ms
8	139.956 ...	8	+ 0.1469 ms	+ 7.538 ms	7.3911 ms
9	159.66 ms	9	+ 0.1283 ms	+ 7.5549 ms	7.4266 ms
10	180.554 ...	10	+ 0.1316 ms	+ 9.1361 ms	9.0045 ms
11	199.644 ...	11	+ 0.1032 ms	+ 10.6869 ms	10.5837 ms
12	220.045 ...	12	+ 0.0891 ms	+ 10.921 ms	10.8319 ms
13	239.628 ...	13	+ 0.136 ms	+ 10.7854 ms	10.6494 ms
14	259.544 ...	14	+ 0.124 ms	+ 9.5091 ms	9.3851 ms
15	280.265 ...	15	+ 0.0368 ms	+ 7.0699 ms	7.0331 ms
16	299.579 ...	16	+ 0.1138 ms	+ 7.9086 ms	7.7948 ms
17	320.003 ...	17	+ 0.1234 ms	+ 9.3457 ms	9.2223 ms
18	339.995 ...	18	+ 0.1296 ms	+ 9.4134 ms	9.2838 ms
19	359.35 ms	19	+ 0.1235 ms	+ 8.8115 ms	8.688 ms

EasyClassify - Classifying Images

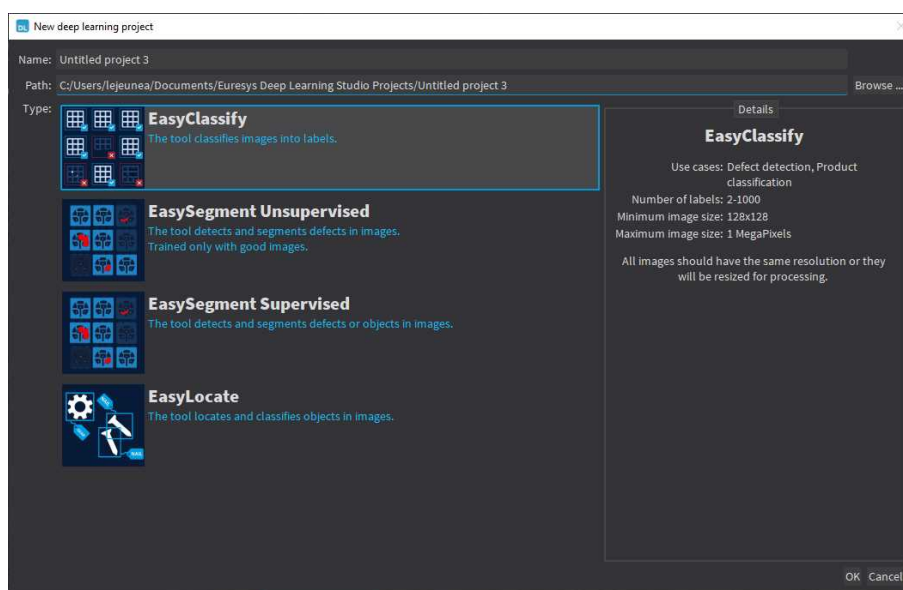
Tool and Images

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

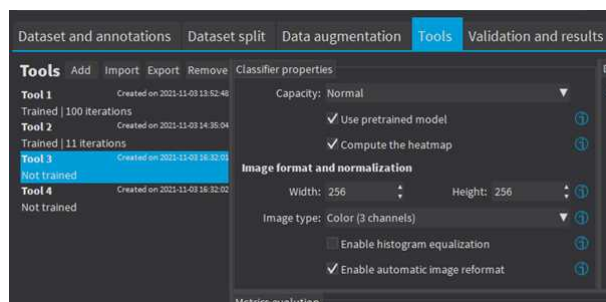
Deep Learning Studio

To create an **EasyClassify** project in **Deep Learning Studio**:

1. Start **Deep Learning Studio**.
2. Create a new project and select **EasyClassify** in the **New deep learning project** dialog.



Tool properties



Capacity

- The **Capacity** of the neural network (default: Normal) represents the quantity of information that it is capable of learning.
 - We recommend you to first train a Normal network on your dataset.
 - ▶ If the learning is working:
 - Try to use the Small or the Large network to get the performance (inference speed and accuracy) that best matches your application.
 - ▶ If the learning does not work it means either that:
 - Your dataset image resolution is too small (below 128×128): try to use the Extra Small network that specifically targets such cases.
 - Your dataset is complex and difficult: try to use the Extra Large network that specifically targets such cases.
- In the API:
 - Use the string parameter `EClassifier::ModelType` to select the model capacity.
 - Use `EClassifier::SetModelType` to set the capacity of your tool.
 - Use `EClassifier::GetAvailableModelTypes` to list the available models for training.

Use pretrained model

- A pretrained model should give faster convergence time and better accuracy for all the datasets except for extremely large datasets (> 10 000 images).
- A pretrained model works by initializing the neural network with the weights learned on a large and complex dataset.
 - This allows to transfer the knowledge of some universal features (such as edges, corners...) learned on this dataset to your dataset.
 - If you do not use a pretrained model, the neural network is initialized randomly.
- In the API, use `EClassifier::SetUsePretrainedModel(false)` to disable the user of pretrained models.

NOTE: You cannot use the histogram equalization with pretrained weights as it would lead to poor training results.

Compute the heatmap

- If this option is checked, the tool computes the heatmap along with each result.
 - The heatmap is available with `EClassificationResult::GetHeatmap` or with `EClassifier::GetHeatmap`.
 - ▶ Note that this option makes the computation of the results a bit slower but twice faster than computing the result and then computing the heatmap separately.
- In the API, use `EClassifier::SetComputeHeatmapWithResult` to enable or disable this option.
- If this option is disabled, use the method `EClassifier::GetHeatmap` to get the heatmap for an image.

Image format and normalization

Width, Height, Image type and Enable automatic image reformat

- 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.



TIP

It is recommended to use a width and height of at least 224×224 , as the performances may start to deteriorate if the height or/and the width are smaller.

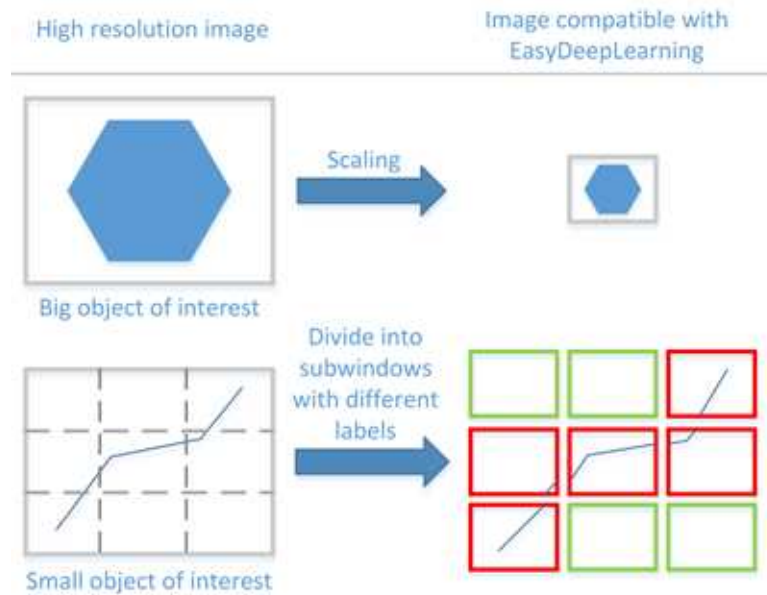
- In **Deep Learning Studio**, you can set the input image format in the **Classifier properties** of the tools that you create in the **Tools** tab.
- In the API, you can also set manually the input image format with the methods `SetWidth`, `SetHeight` and `SetChannels` (1 channel for grayscale images and 3 channels for color images).

Image resolution

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.



Enable 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.

NOTE: You cannot use the histogram equalization with pretrained weights as it would lead to poor training results.

Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 86.

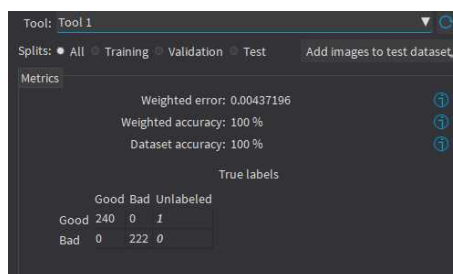
Validating the Results

In Deep Learning Studio:

- The metrics are always computed without applying data augmentation on the images.
- In the **Tools** tab, the metrics **Best validation error** and **Best validation accuracy** are computed during the training using the label weights. The evolution of several metrics during the training is also available.



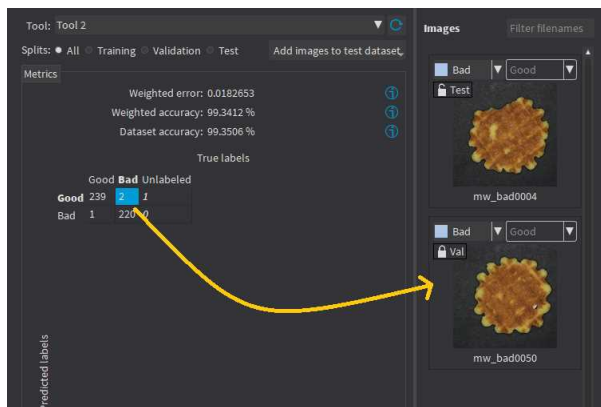
- In the **Validation and results** tab, there are 3 metrics displayed:
 - The **weighted error** and the **weighted accuracy** (normalized with respect to the label weights instead of being dependent of the number of images for each label).
 - The **dataset accuracy** (it does not use the label weights).



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 **Validation and 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 **Validation and results** tab, you can export the results using the 2 buttons below the metrics and confusion matrix.
 - The **Export heatmaps** button exports the heatmap for each image of your dataset.
 - The heatmaps are saved in the PNG format under the name ImageName_ToolName_heatmap_UniqueId.png.
 - This feature requires a **Deep Learning** license.
 - The **Export results to CSV** button exports the results for each image of your dataset as rows of a CSV file.
 - Each record contains the image filename, the ground truth label, the predicted label, the prediction probability and the probabilities for all the labels recognized by the tool.



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`.

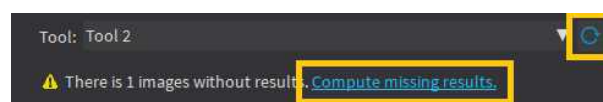


TIP

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

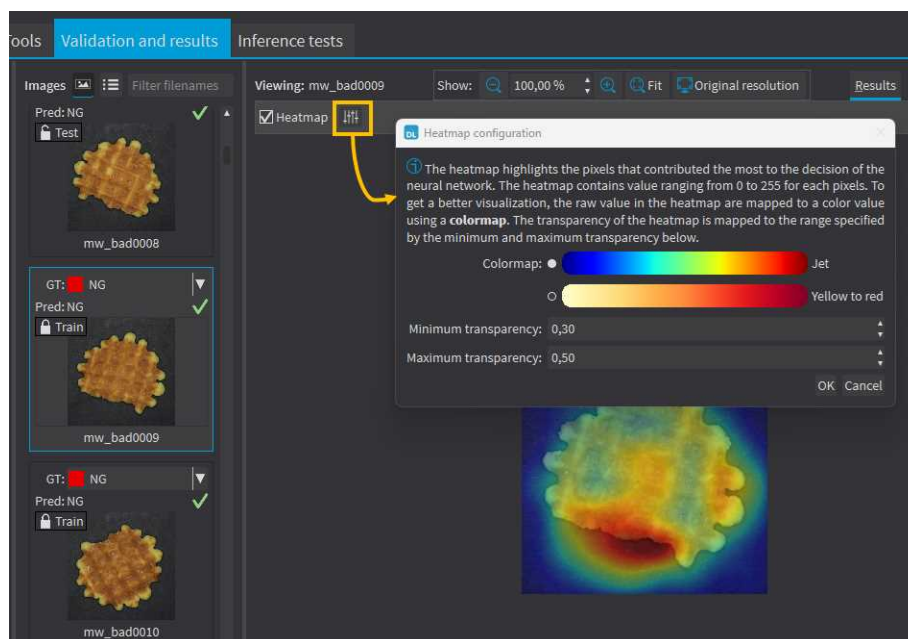
Classifying New Images

- In **Deep Learning Studio**:
 - Add new images to the dataset and refresh the results.



- Open the `Inference tests` tab to classify the new images and display the detailed results for these images.
- You can also export the results to CSV or export the heatmap from this tab.

- In the image viewer, you can enable or disable the heatmap visualization and configure how the heatmap is displayed (color map and transparency).



- 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).
 - `EClassificationResult::GetHeatmap` and `EClassificationResult::GetColorizedHeatmap` return a heatmap highlighting the pixels that have contributed the most to get the most probable label.
 - The heatmap is only available when the classifier property `EClassifier::ComputeHeatmapWithResult` is set to True.
 - Use `EClassificationResult::HasHeatmap` to check if the result contains a heatmap.

- 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 a heatmap highlighting the pixels that contribute the most to a label.
 - In some cases, this heatmap can provide a rough localization of the object corresponding to the label.
 - The heatmap is colored, and the important parts are displayed in red.

**TIP**

Since large memory allocations take a lot of time, a classification does not release its 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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- On **Windows**:
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.

Capacity Extra Small

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	2.72	1.42	15.14	19.58
	4	0.97	=	2.38	=
	16	0.72	=	1.69	=
	64	0.55	=	0.96	=
256 × 256	1	5.36	3.95	8.29	65.00
	4	2.58	=	5.27	=
	16	1.91	=	3.26	=
	64	0.54	=	2.24	=
512 × 512	1	5.10	19.20	19.76	311.00
	4	7.52	=	12.11	=
	16	2.01	=	7.49	=
	64	2.73	=	7.40	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	132	n/a
	4	139	159
	16	166	242
	64	276	571
256 × 256	1	139	n/a
	4	166	241
	16	275	569
	64	712	1 879
512 × 512	1	166	n/a
	4	275	568
	16	711	1 877
	64	2 455	7 114

Capacity Small

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	3.26	2.65	22.99	33.00
	4	1.67	=	3.81	=
	16	0.88	=	2.02	=
	64	0.53	=	1.38	=
256 × 256	1	3.89	7.40	11.85	108.00
	4	3.00	=	7.95	=
	16	1.59	=	3.80	=
	64	0.59	=	3.35	=
512 × 512	1	6.58	34.50	27.94	521.00
	4	5.84	=	14.47	=
	16	2.19	=	10.82	=
	64	2.79	=	10.02	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	137	n/a
	4	147	180
	16	188	304
	64	350	799
256 × 256	1	147	n/a
	4	187	303
	16	349	796
	64	996	2 766

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
512 × 512	1	187	n/a
	4	349	795
	16	995	2 763
	64	3 579	10 635

Capacity Normal

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	5.61	4.65	15.26	47.30
	4	4.01	=	4.45	=
	16	1.55	=	2.27	=
	64	0.52	=	1.23	=
256 × 256	1	5.74	14.40	36.33	179.00
	4	5.87	=	8.73	=
	16	2.11	=	4.54	=
	64	0.67	=	3.74	=
512 × 512	1	7.75	60.00	49.12	760.00
	4	7.63	=	17.72	=
	16	2.74	=	15.28	=
	64	2.97	=	14.44	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	142	n/a
	4	153	201
	16	199	357
	64	383	983
256 × 256	1	153	n/a
	4	199	357
	16	381	980
	64	1 110	3 473
512 × 512	1	199	n/a
	4	381	979
	16	1 109	3 470
	64	4 021	13 433

Capacity Large

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	3.75	8.50	40.42	94.00
	4	2.44	=	6.49	=
	16	1.64	=	2.61	=
	64	0.29	=	2.26	=
256 × 256	1	4.25	25.60	55.36	367.00
	4	5.45	=	8.80	=
	16	0.84	=	6.46	=
	64	0.89	=	6.53	=
512 × 512	1	6.06	128.00	32.07	1 630.00
	4	3.12	=	23.61	=
	16	3.17	=	21.72	=
	64	3.25	=	20.43	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	152	n/a
	4	178	272
	16	281	611
	64	695	1 965
256 × 256	1	178	n/a
	4	281	610
	16	693	1 960
	64	2 341	7 360
512 × 512	1	281	n/a
	4	692	1 959
	16	2 338	7 355
	64	9 075	29 094

Capacity Extra Large

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	6.05	13.30	37.56	181.50
	4	3.78	=	8.91	=
	16	1.71	=	3.42	=
	64	0.40	=	3.80	=

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
256 × 256	1	6.81	50.00	28.23	726.00
	4	6.07	=	12.64	=
	16	1.25	=	10.76	=
	64	1.57	=	11.17	=
512 × 512	1	10.24	244.00	50.34	3 233.00
	4	4.76	=	42.70	=
	16	5.88	=	39.70	=
	64	-	=	-	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	170	n/a
	4	251	442
	16	577	1 295
	64	1 879	4 705
256 × 256	1	251	n/a
	4	576	1 293
	16	1 875	4 698
	64	7 074	18 333
512 × 512	1	576	n/a
	4	1 874	4 696
	16	7 070	18 326
	64	28 004	73 212

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

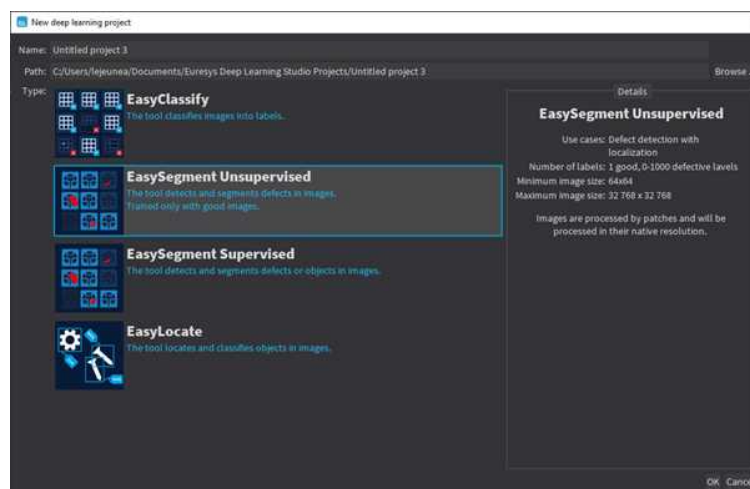
EasySegment Unsupervised is the deep learning tool part of the **EasySegment** segmentation library of **Open eVision**. It detects and segments defects in images.

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.

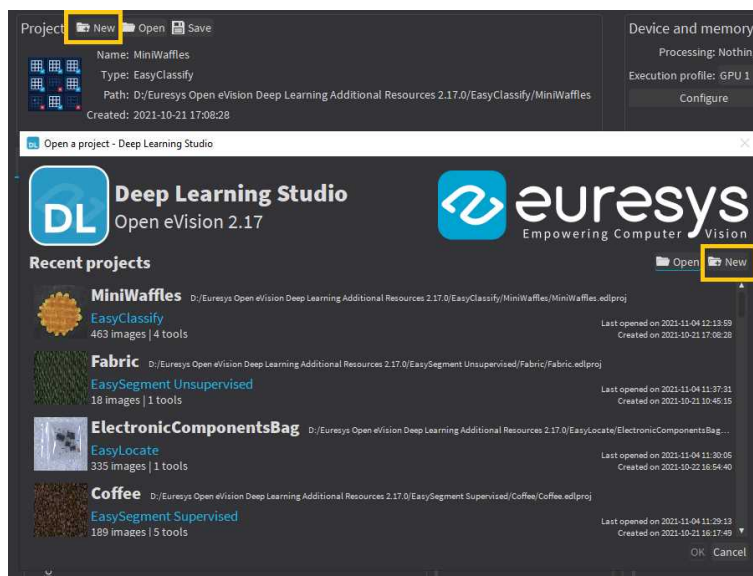
Deep Learning Studio

To create an **EasySegment Unsupervised** project in **Deep Learning Studio**:

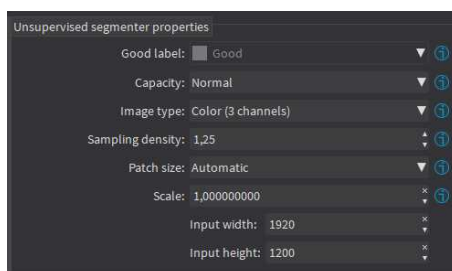
1. Start **Deep Learning Studio**.
2. Create a new project and select **EasySegment Unsupervised** in the **New deep learning project** dialog.



The following dialog is displayed when clicking on **New** in the **Open a project** dialog displayed at the start of **Deep Learning Studio** or when you click on **New** in the toolbar.



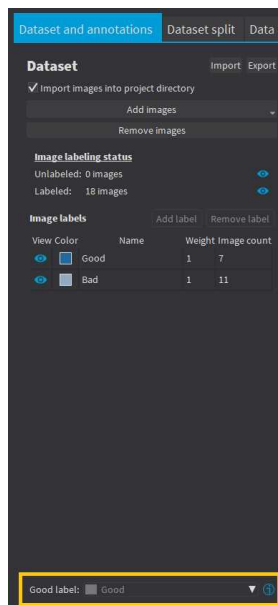
Configuration



The unsupervised segmenter tool has 6 parameters:

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

In **Deep Learning Studio**, the `Good label` is a project property that you must select in the `Dataset and annotations` tab below the list of labels.



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 (1 channel)`):

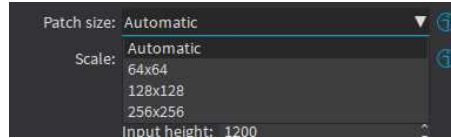
In the API:

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

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 86.

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 **Tools** 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 **Validation and 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.

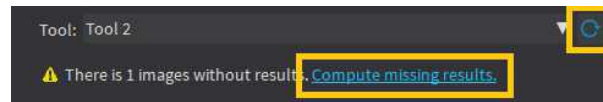
In the API:

- 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:

- Add new images to the dataset and refresh the results.



- Open the `Inference tests` tab to apply the segmenter to new images and display detailed results for these images.
- 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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- **On Windows:**
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.

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 approximately 25% of the time reported below.

Capacity Small

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	722.34	1 280	949.36	9 314
	4	237.25	=	344.92	=
	16	97.55	=	221.60	=
	64	68.11	=	191.46	=

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	289.34	1 472	619.84	6 946
	4	96.46	=	286.99	=
	16	51.83	=	201.88	=
	64	40.60	=	167.90	=
256 × 256	1	92.99	896	325.74	6 509
	4	48.62	=	214.72	=
	16	34.67	=	164.97	=
	64	38.92	=	150.79	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	5	n/a
	4	8	11
	16	13	26
	64	52	101
128 × 128	1	12	n/a
	4	23	41
	16	60	117
	64	221	430
256 × 256	1	44	n/a
	4	80	151
	16	233	454
	64	869	1 690

Capacity Normal

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	851.08	2 533	1 379.48	18 734
	4	245.62	=	532.91	=
	16	106.22	=	357.82	=
	64	70.44	=	271.87	=
128 × 128	1	312.90	3 216	1 095.83	17 763
	4	104.09	=	557.38	=
	16	55.60	=	369.82	=
	64	42.79	=	275.37	=
256 × 256	1	103.61	2 246	740.05	13 881
	4	53.91	=	421.52	=
	16	38.22	=	309.25	=
	64	46.27	=	257.49	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	3	n/a
	4	8	15
	16	26	50
	64	102	192
128 × 128	1	37	n/a
	4	53	98
	16	126	240
	64	441	834
256 × 256	1	145	n/a
	4	211	386
	16	487	935
	64	1 588	3 128

Capacity Large

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	965.58	6 178	2 183.06	40 833
	4	278.08	=	938.51	=
	16	117.13	=	622.66	=
	64	75.41	=	387.08	=
128 × 128	1	361.31	8 882	2 824.60	57 329
	4	129.73	=	1 394.44	=
	16	66.03	=	798.73	=
	64	53.80	=	470.68	=
256 × 256	1	187.35	7 254	2 969.24	38 020
	4	79.50	=	1 461.80	=
	16	51.84	=	667.86	=
	64	67.15	=	544.54	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	11	n/a
	4	20	36
	16	56	103
	64	209	383

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	133	n/a
	4	164	293
	16	300	561
	64	840	1 632
256 × 256	1	523	n/a
	4	654	1 163
	16	1 180	2 208
	64	3 664	6 768

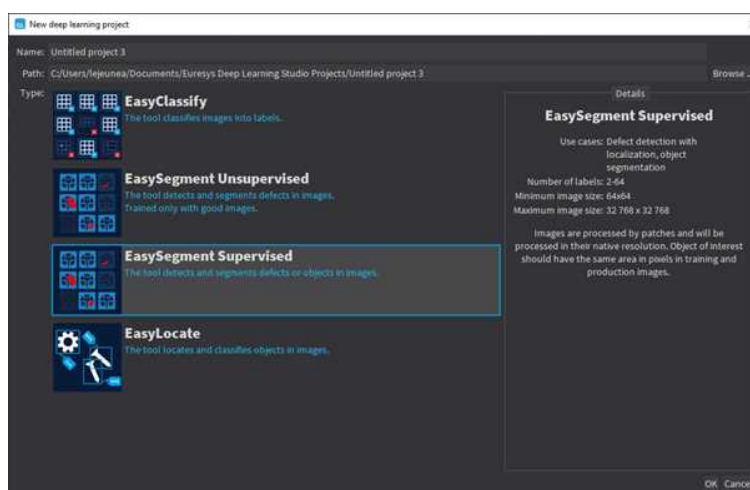
EasySegment Supervised

Tool and Configuration

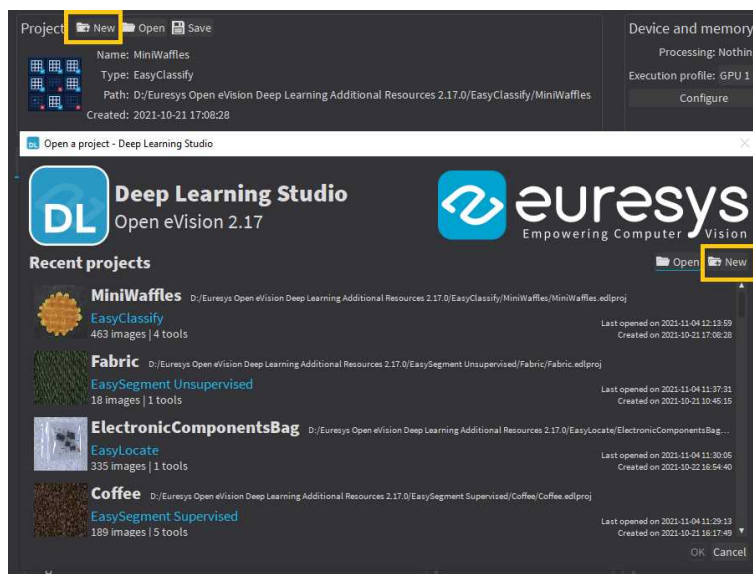
Deep Learning Studio

To create an **EasySegment Supervised** project in **Deep Learning Studio**:

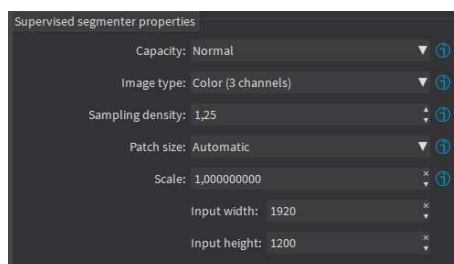
1. Start **Deep Learning Studio**.
2. Create a new project and select **EasySegment Supervised** in the **New deep learning project** dialog. .



The following dialog is displayed when clicking on **New** in the **Open a project** dialog displayed at the start of **Deep Learning Studio** or when you click on **New** in 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:

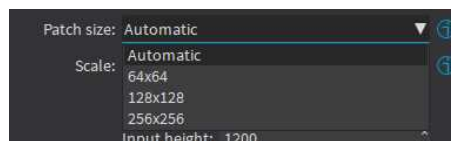
- The capacity is represented by the enumerate type `ESupervisedSegmenterCapacity`.
- `ESupervisedSegmenter::Capacity` sets the capacity of the tool.

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 86.

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.

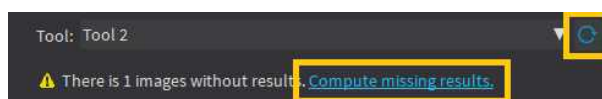


NOTE

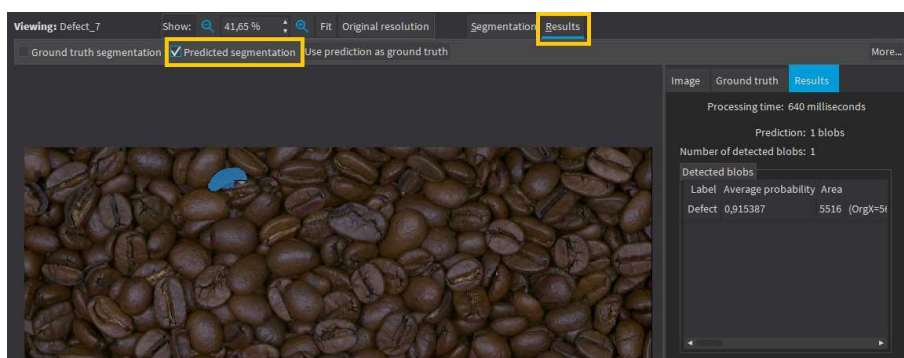
In the context of defect detection, an image is considered to be without defect when its score is below or equal to the threshold of the supervised segmenter tool.

In Deep Learning Studio:

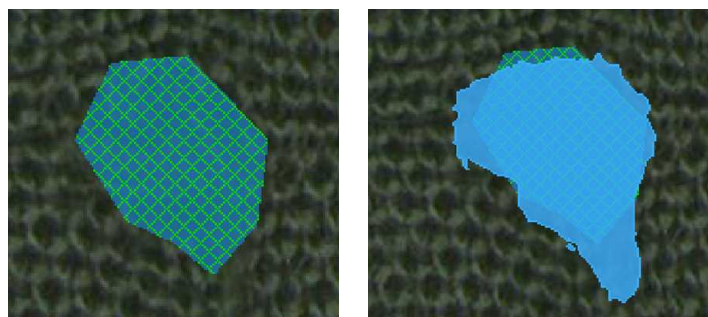
- Add new images to the dataset and refresh the results.



- Open the **Inference tests** tab to apply the segmenter to new images and display detailed results for these images.
- To visualize the segmentation of an image, check the **Predicted segmentation** option (CTRL + P) in the **Result** menu (ALT + R) of the image viewer.

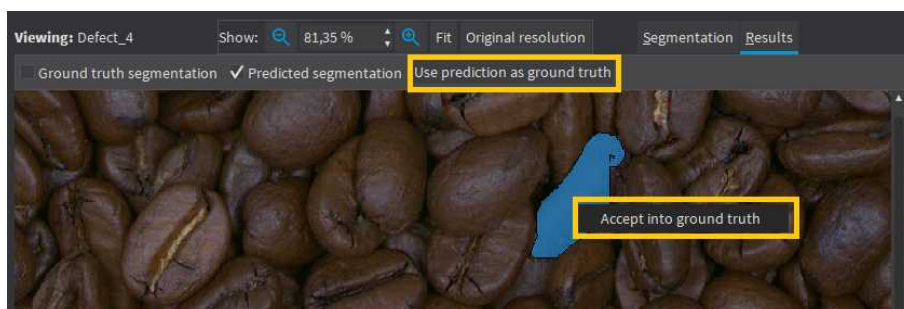


- If the image has a ground truth, check the **Ground truth segmentation** option (CTRL + G) to display it. It appears with a green pattern drawn over it.

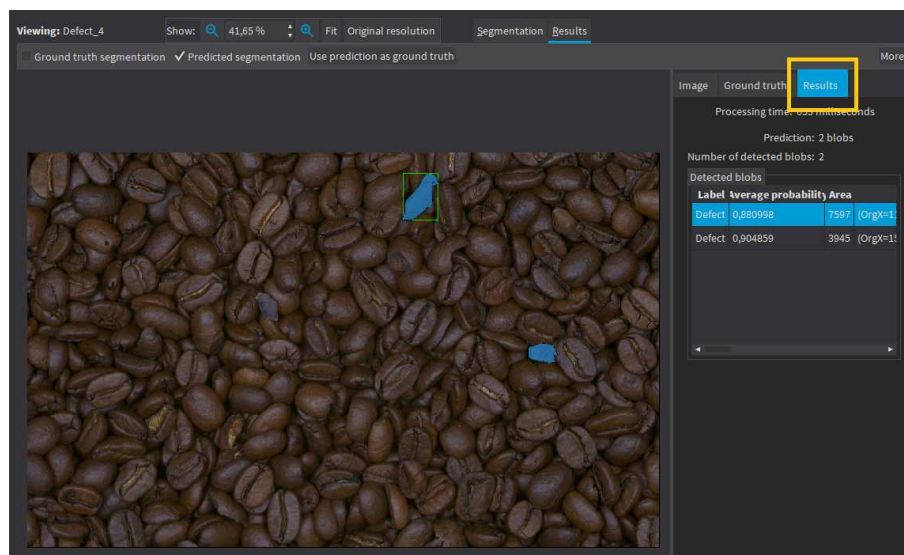


Ground truth (left) and prediction on top of ground truth (right)

- 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 `Results` tab of the image viewer.



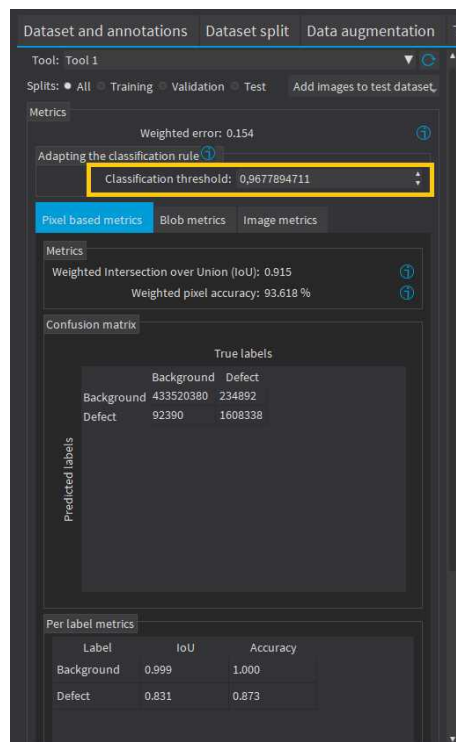
In the API:

- 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**.

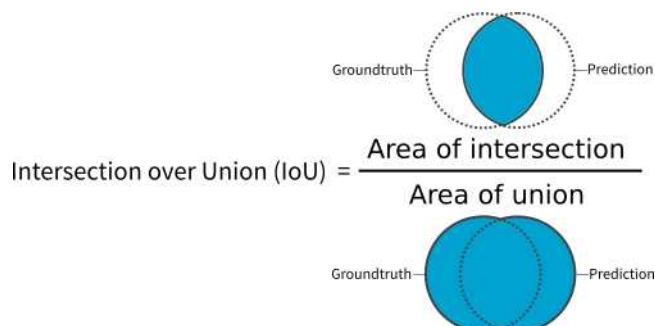


- In the API:
 - The metrics are represented by an `ESupervisedSegmenterMetrics` object.

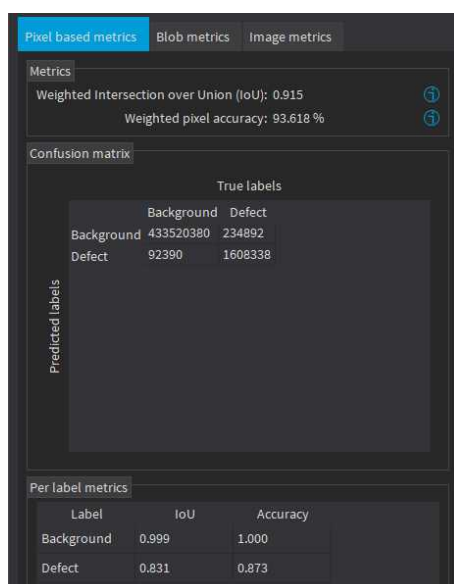
The pixel-based metrics

- 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.



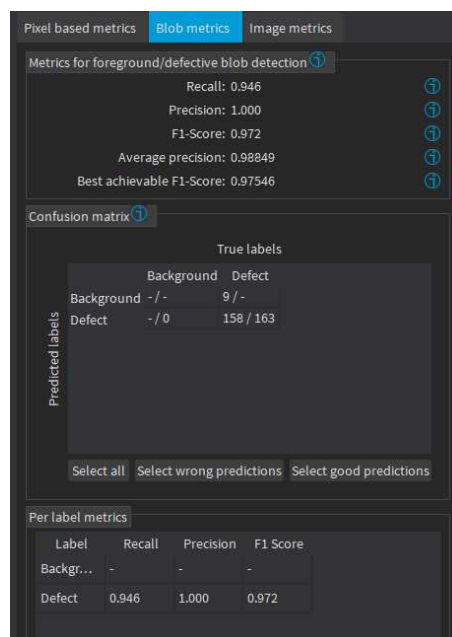
- The *accuracy* that is the proportion of the pixels of the label that are correctly predicted.



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*

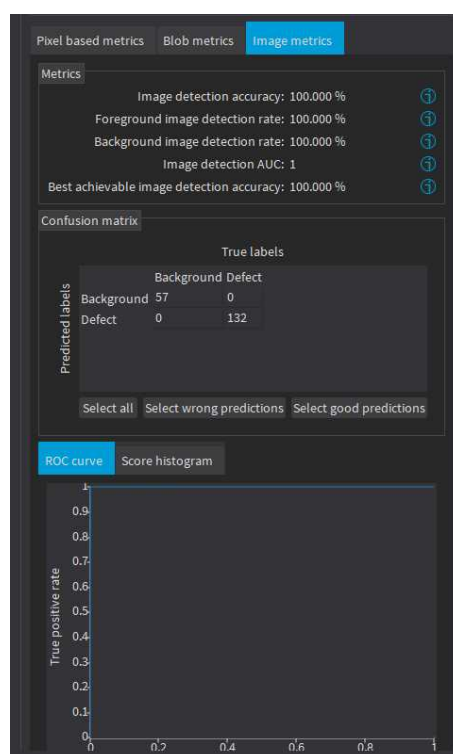


The image-based metrics

- 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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- **On Windows:**
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.

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 approximately 25% of the time reported below.

Capacity Small

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	583	2 008	669	32 443
	4	196	=	418	=
	16	90	=	354	=
	64	71	=	325	=

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	216	2 122	444	28 278
	4	92	=	359	=
	16	65	=	331	=
	64	64	=	358	=
256 × 256	1	105	2 435	435	28 811
	4	74	=	390	=
	16	66	=	365	=
	64	94	=	470	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	7	n/a
	4	22	41
	16	85	154
	64	288	560
128 × 128	1	33	n/a
	4	85	164
	16	310	604
	64	1 217	2 371
256 × 256	1	119	n/a
	4	344	669
	16	1 257	2 466
	64	5 770	10 516

Capacity Normal

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	651	5 771	1 080	67 544
	4	212	=	768	=
	16	104	=	661	=
	64	90	=	599	=
128 × 128	1	246	6 390	1 039	73 102
	4	113	=	862	=
	16	90	=	737	=
	64	89	=	815	=
256 × 256	1	125	8 120	1 127	86 212
	4	104	=	990	=
	16	95	=	873	=
	64	135	=	1 068	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	143	n/a
	4	168	209
	16	157	297
	64	604	1 145
128 × 128	1	193	n/a
	4	191	362
	16	644	1 240
	64	2 486	4 780
256 × 256	1	304	n/a
	4	750	1 453
	16	2 642	5 092
	64	10 208	19 648

Capacity Large

Patch size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
64 × 64	1	703	17 220	3 025	191 224
	4	263	=	2 367	=
	16	174	=	1 857	=
	64	164	=	1 694	=
128 × 128	1	321	22 050	3 392	264 352
	4	219	=	2 839	=
	16	188	=	2 158	=
	64	176	=	2 125	=
256 × 256	1	273	28 420	4 188	312 504
	4	219	=	3 459	=
	16	203	=	2 569	=
	64	-	=	-	=

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
64 × 64	1	57	n/a
	4	223	319
	16	422	716
	64	1 281	2 369

Patch size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	229	n/a
	4	528	929
	16	1 437	2 681
	64	5 388	10 007
256 × 256	1	915	n/a
	4	1 830	3 467
	16	5 880	10 991
	64	20 545	39 553

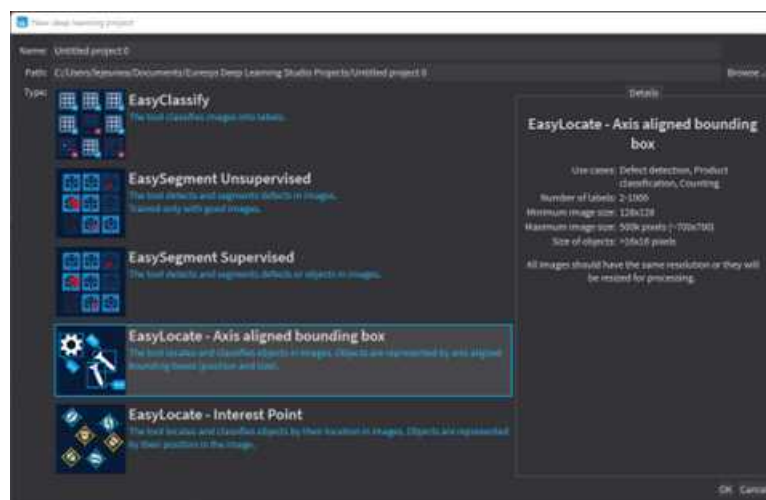
EasyLocate - Locating Objects and Defects

Tool and Configuration

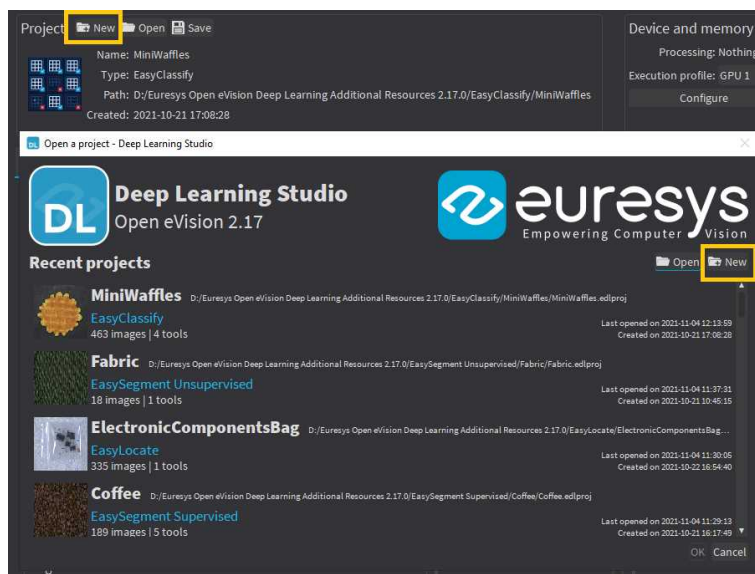
Deep Learning Studio

To create an **EasyLocate** tool in **Deep Learning Studio**:

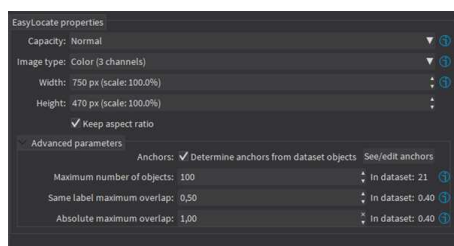
1. Start **Deep Learning Studio**.
2. Create a new project and select **EasyLocate Axis Aligned Bounding Box** or **EasyLocate Interest Point** in the **New deep learning project** dialog.



The following dialog is displayed when clicking on **New** in the **Open a project** dialog displayed at the start of **Deep Learning Studio** or when you click on **New** in 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)**):

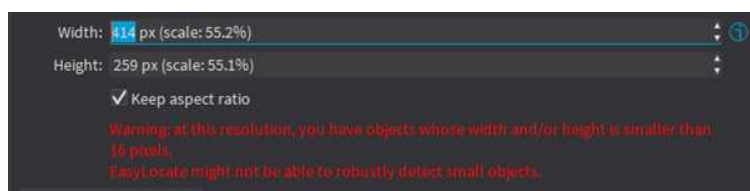
In the API:

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



In the API:

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

EasyLocate Axis Aligned Bounding Box configuration (advanced parameters)

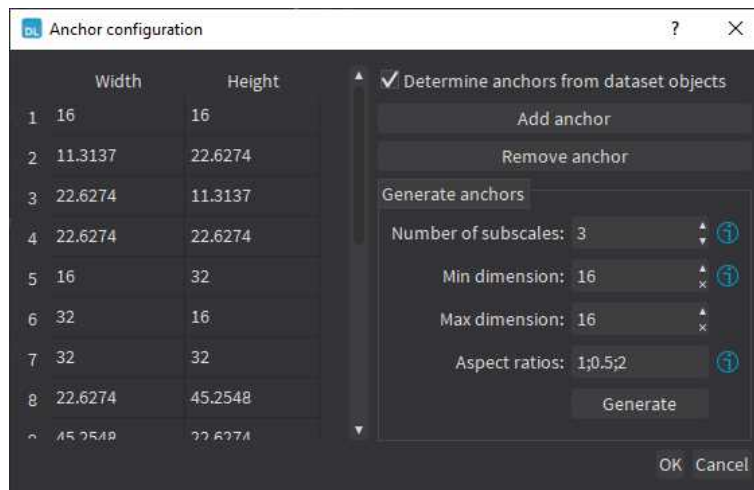
The **EasyLocate Axis Aligned Bounding Box** tool has 4 advanced parameters linked to **EasyLocate** neural network design.

- 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.



TIP

The anchors must be defined with respect to the width and height of the current tool. Thus, when specifying the anchors manually, the width and height parameters are locked.

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 `ELocatorBase::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`.

- 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.

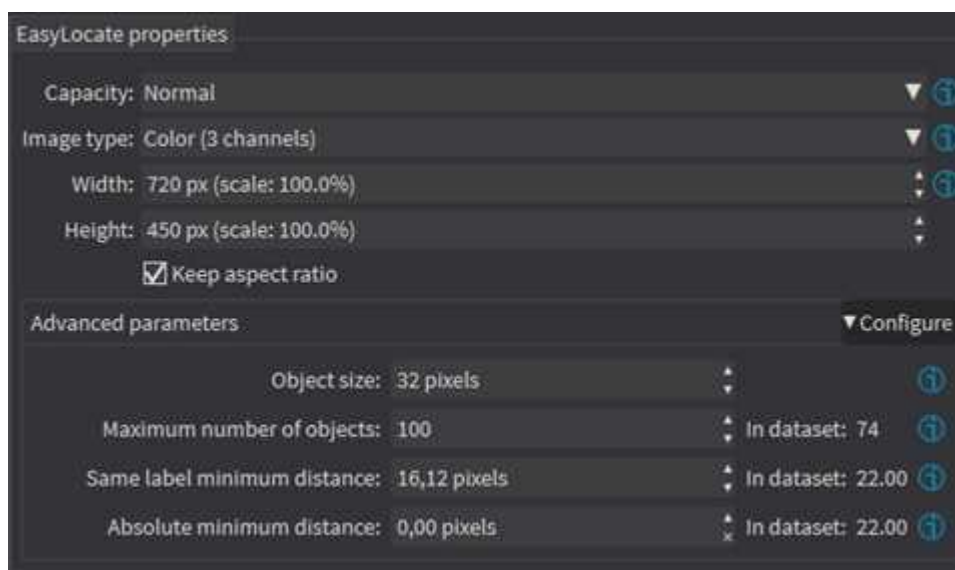
In the API:

- 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.

EasyLocate Interest Point configuration (advanced parameters)

The **EasyLocate Interest Point** tool has 4 advanced parameters.



- The **Object size** is the size of the objects that you want to detect in the image.
 - By default, the value of this parameter is the object size specified for the dataset.
 - The object size has the same role as the anchors for **EasyLocate Axis Aligned Bounding Box**.
 - The object size is defined with respect to the size of the images in the dataset. When training a tool with a different input resolution, the object size parameters should not change (they are internally adapted according to the resolution of the training images and the width and height set for the tool).

In the API:

- Use `EInterestPointLocator::ObjectSize`

- 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 `ELocatorBase::MaxNumberOfObjects`.

3. The `Same label minimum distance` is the minimum distance between objects with the same label.

- By default the value is a third of the object size.

In the API:

- Use `EInterestPointLocator::SameLabelMinDistance`

4. The `Absolute minimum distance` is the minimum distance between objects, regardless of their label.

- By default the value is 0 and it means that the tool can predict two objects with different labels but with the exact same position.

In the API:

- Use `EInterestPointLocator::AbsoluteMinDistance`

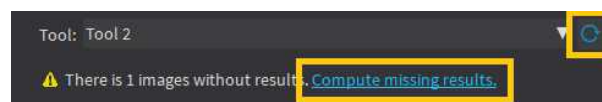
Training

To train your tool, see "[Training a Deep Learning Tool](#)" on page 86.

Locating Objects

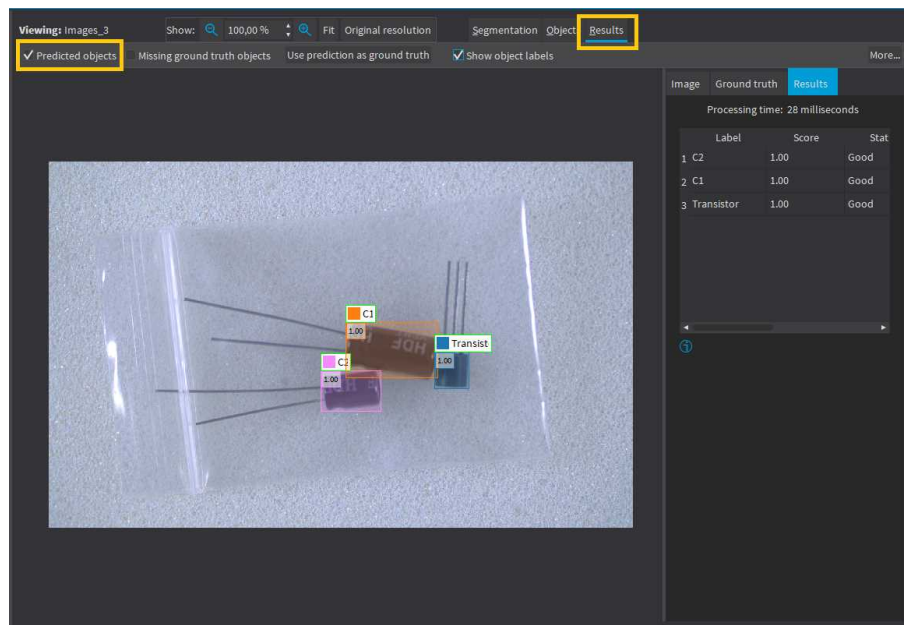
In Deep Learning Studio:

- To apply **EasyLocate** to new images:
 - Add new images to the dataset and refresh the results.



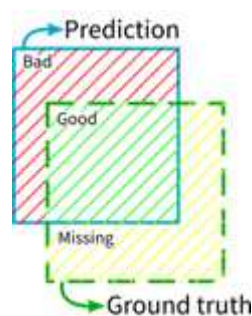
- Open the `Inference tests` tab to apply the tool to new images and display detailed results for these images.

- 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).



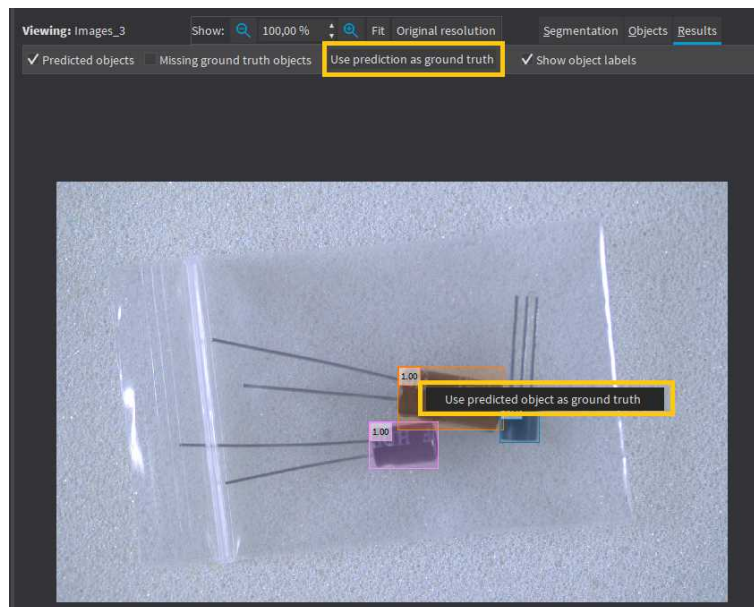
- If the image has a ground truth, check the **Missing ground truth object** option (CTRL + G) to display missing objects. They appear with a yellow pattern drawn over it.
- In the **Results** tab on the right side:
 - The list of detected objects shows their label, their score, whether they are matched to a ground truth object and their predicted bounding box or position.
 - 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.

For **EasyLocate Axis Aligned Bounding Box**, 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.



In the API

- To apply **EasyLocate** to an image, use `ELocatorBase::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.
- Use `ELocatorResult::LocatorFeatures` to check if the result was computed by **EasyLocate Axis Aligned Bounding Box** or **EasyLocate Interest Point**.

The value of `ELocatorResult::LocatorFeatures` is a combination (using the binary OR (`|`) operator) of one or more `ELocatorFeature` values. It indicates the type of information predicted by the tool:

- For **EasyLocate Interest Point**: `ELocatorFeature_Position`.
 - For **EasyLocate Axis Aligned Bounding Box**: `ELocatorFeature_Position` | `ELocatorFeature_Size`.
- With **EasyLocate Interest Point**, for each predicted object, use:
 - `ELocatorObject::PositionX` to get the X coordinate of the object.
 - `ELocatorObject::PositionY` to get the Y coordinate of the object.
 - `ELocatorObject::Label` to get its label.
 - `ELocatorPredictedObject::Probability` to get its predicted probability.
 - With **EasyLocate Axis Aligned Bounding Box**, for each predicted object, use:
 - `ELocatorObject::PositionX` to get the X coordinate of the center of the bounding box.
 - `ELocatorObject::PositionY` to get the Y coordinate of the center of the bounding box.
 - `ELocatorObject::OrgX` to get the X coordinate of the top left origin of the bounding box.

- `ELocatorObject::OrgY` to get the Y coordinate of the top left origin of the bounding box.
- `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.

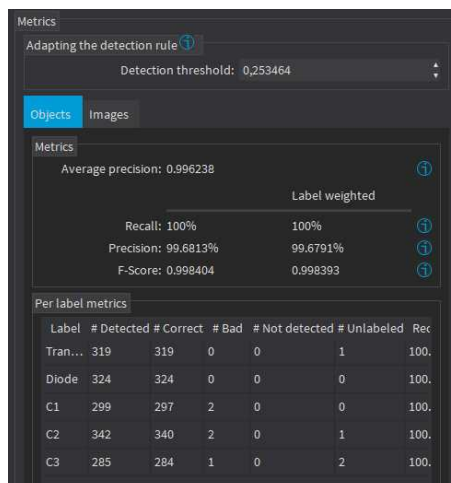
Validating the Results

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`.



Label	# Detected	# Correct	# Bad	# Not detected	# Unlabeled	Rec
Tran...	319	319	0	0	1	100.
Diode	324	324	0	0	0	100.
C1	299	297	2	0	0	100.
C2	342	340	2	0	1	100.
C3	285	284	1	0	2	100.

In the API:

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

The object-based metrics

The object-based metrics are computed by matching actual, ground truth objects to detected objects.

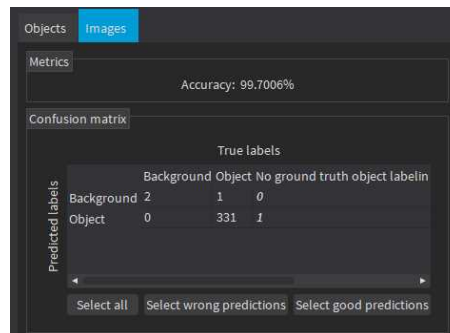
A ground truth object and a detected object are matched together if:

- They have the same label.
- With **EasyLocate Axis Aligned Bounding Box**:
 - 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.
- With **EasyLocate Interest Point**:
 - Their distance is lower than the `Same label minimum overlap` parameter of the tool
 - There is no other ground truth that has a smaller distance with the detected object and there is no other detected object that has a smaller distance 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.

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 memory requirements indicated below are approximate and can vary according to the GPU model.
 - These values were obtained for a **NVIDIA GeForce 3080 Ti** on **Windows 11**.
 - The GPU inference can be 10 to 50% faster on **Linux** for **GeForce** GPUs.
- **On Windows:**
 - When using the WDDM driver mode (always on for a **GeForce** GPU), the inference times can vary quite a lot.
 - When using the TCC mode on a **Quadro** GPU, the inference times are more stable.
- In the tables below 'n/a' means that the value could not be computed for this specific configuration (for example because there is not enough memory).
- In the tables below, a '=' means that the value is equal to the one above it.
- The benchmarks were obtained using **EasyLocate Axis Aligned Bounding Box**.

- For **EasyLocate Interest Point**, the training and inference speeds are approximately the same. The small variations (a few percent slower or faster) in the processing speed depend on the parameters of the tool.

Capacity Small

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	4.19	30.13	18.55	504
	4	4.44	=	7.43	=
	16	1.81	=	4.16	=
	64	0.41	=	3.45	=
256 × 256	1	4.85	84	32.02	1 959
	4	6.88	=	16.74	=
	16	1.45	=	13.51	=
	64	1.37	=	14.19	=
512 × 512	1	11.32	341	66.10	9 314
	4	5.70	=	60.85	=
	16	5.38	=	53.89	=
	64	-	=	56.28	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	175	n/a
	4	241	354
	16	503	879
	64	1 553	2 979
256 × 256	1	241	n/a
	4	503	879
	16	1 553	2 979
	64	5 884	11 511
512 × 512	1	503	n/a
	4	1 553	2 979
	16	5 884	11 511
	64	23 455	45 885

Capacity Normal

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	3.75	32	19.03	645
	4	2.43	=	8.14	=
	16	1.90	=	4.48	=
	64	0.42	=	3.69	=
256 × 256	1	7.00	91	32.36	2 717
	4	6.83	=	18.14	=
	16	1.59	=	14.88	=
	64	1.54	=	15.47	=
512 × 512	1	8.93	391	71.63	12 646
	4	5.62	=	66.94	=
	16	5.39	=	58.83	=
	64	-	=	61.78	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	178	n/a
	4	248	369
	16	528	929
	64	1 648	3 168
256 × 256	1	248	n/a
	4	528	929
	16	1 648	3 168
	64	6 256	12 255
512 × 512	1	528	n/a
	4	1 648	3 168
	16	6 256	12 255
	64	24 937	48 849

Capacity Large

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
128 × 128	1	6.31	76	48.57	1 194
	4	4.99	=	14.32	=
	16	1.08	=	10.05	=
	64	0.85	=	7.34	=

Image size	Batch	Inference time / image (ms)			
		GPU NVIDIA GeForce 3080Ti	CPU Intel Core i9 7900X	GPU NVIDIA Jetson Xavier NX (ARM)	CPU Raspberry Pi 4 Model B
256 × 256	1	8.80	205	65.37	5 694
	4	3.46	=	38.04	=
	16	2.25	=	30.14	=
	64	2.32	=	29.54	=
512 × 512	1	25.93	866	168.90	26 320
	4	9.11	=	128.96	=
	16	8.52	=	115.15	=
	64	-	=	-	=

Image size	Batch	GPU memory for inference (MB)	GPU memory for training (MB)
128 × 128	1	288	n/a
	4	421	714
	16	952	1 776
	64	3 075	6 023
256 × 256	1	421	n/a
	4	952	1 776
	16	3 075	6 023
	64	11 701	23 144
512 × 512	1	952	n/a
	4	3 075	6 023
	16	11 701	23 144
	64	46 450	91 874

5. Code Snippets

5.1. 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= new EImageBW8();
EImageBW8 dstImage= new EImageBW8();

// 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= new EImageBW8();

// Size of the third-party image
int sizeX = bufferSizeX;
int sizeY = bufferSizeY;

//Pointer to the third-party image buffer
IntPtr imgPtr = bufferPointer;

// ...

// 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 to access //
// the pixel values in a BW8 image.                          //
////////////////////////////////////

using System.Runtime.InteropServices;

IntPtr pixAddr;
byte pix;

//...

for(int y = 0; y < height; ++y)
{
    pixAddr = bw8Image.GetImagePtr(0,y);
    for(int x = 0; x < width; ++x)
        pix = Marshal.ReadByte(pixAddr,x);
}

```

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= new EImageBW8();

// ROI constructor
EROIBW8 myROI= new EROIBW8();

// 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= new EBW8Vector();
EBW8 bw8 = new EBW8();

```

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

// Fill the vector with increasing values
for(int i= 0; i < 128; i++)
{
    bw8.Value = (byte)i;
    ramp.AddElement(bw8);
}

// Retrieve the 10th element value
EBW8 value = ramp.GetElement(9);
```

Exception Management

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

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

try
{
    // Image constructor
    EImageC24 srcImage= new EImageC24();

    // ...

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

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

5.2. 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= new EClassificationDataset();
EClassificationDataset trainingDataset= new EClassificationDataset();
EClassificationDataset validationDataset= new EClassificationDataset();
```

```

EClassifier classifier= new EClassifier();

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

// Enabling data augmentation on the dataset
dataset.EnableDataAugmentation= true;

// Rotation of up to 90°
dataset.MaxRotationAngle= 90.0F;

// Enabling horizontal flips
dataset.EnableHorizontalFlip= 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.BestEpoch);

// Dispose of objects
dataset.Dispose();
trainingDataset.Dispose();
validationDataset.Dispose();
classifier.Dispose();

```

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= new EClassifier();
EImageBW8 srcImage= new EImageBW8();

// String and probability for the most probable result
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.BestLabel;
probability = result.BestProbability;

// Dispose of objects
classifier.Dispose();
srcImage.Dispose();

```

Using Multithreading for Classification

```

////////////////////////////////////
// This code snippet shows how to parallelize the           //
// classification of new images on the CPU.                 //
// This code snippet requires the .NET Framework 4.0       //
////////////////////////////////////

using System.Collections.Threading;
using System.Collections.Concurrent;

...

static void ClassificationLoop(Object obj)
{
    BlockingCollection<EImageC24> queue = obj as BlockingCollection<EImageC24>;

    EClassifier c = new EClassifier();
    c.Load("classifier.ecl");

    while (!queue.IsCompleted)
    {
        EImageC24 image = queue.Take();

        EClassificationResult result = c.Classify(image);
        // Get the most probable label
        string label = result.BestLabel;
        float probability = result.BestProbability;

        // Perform other actions based on the result
        ...
    }
}

...

int NUM_THREADS = 2;

// Queue holding the image to classify
BlockingCollection<EImageC24> imageQueue = new BlockingCollection<EImageC24>(new ConcurrentQueue<EImageC24>(),
2 * NUM_THREADS);

// Create and start the thread pool
Thread[] threads = new Thread[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++)
{
    threads[i] = new Thread(ClassificationLoop);
    threads[i].Start(imageQueue);
}

bool hasImage = true;
while (hasImage)
{
    EImageC24 image = new EImageC24();

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

    // Add the image to the queue
}

```

```
imageQueue.Add(image);

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

// Tell the threads that they won't have any new image coming.
imageQueue.CompleteAdding();

// Wait for the threads to finish
for (int i = 0; i < NUM_THREADS; i++)
    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 and segmenter constructor
EUnsupervisedSegmenter segmenter = new EUnsupervisedSegmenter();
EImageBW8 image = new EImageBW8();

// Load segmenter and image
segmenter.Load(...);
image.Load(...);

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

// Retrieve the segmentation map
EImageBW8 segmentationMap = result.SegmentationMap() ;

// Dispose of objects
segmenter.Dispose();
image.Dispose();
```