Zebra
Aurora™ Vision

Aurora Vision Library 5.6

Technical Issues

Created: 9/25/2025

Product version: 5.6.1.79554

Table of content:

Interfacing with Other Libraries

Aurora Vision Library contains the avl::Image class which represents an image. This article describes how to create an avl::Image object with raw data acquired from cameras, and how to convert it to image structures specific to other libraries.

Aurora Vision Library provides a set of sample converters. To use it in your program you should include a specific header file which is available in Aurora Vision Library include directory (e.g. AVLConverters/AVL_OpenCV.h). The list below presents all the available converters:

- Euresys
- MFC
- MvAcquire
- OpenCV
- Pylon
- QT
- SynView

An example of using MFC converters can be found in the Aurora Vision Library directory in My Documents (Examples\MFC Examples). Below is shown also an OpenCV converter example.

Example: Converting Between avl::Image and OpenCV Mat

It is also possible to convert avl::Image to image structures from common libraries. The example code snippets below show how to convert an avl::Image object to other structures.

```
#include <opencv2/highgui/highgui.hpp>
#include <AVLConverters/AVL_OpenCV.h>

#include <AVL.h>

avl::Image inputImage, processedImage;
cv::Mat cvImage;

int thresholdValue, rotateAngle;

//image processing
void ProcessImage()
{
 avl::Image image1;
 avl::ThresholdImage(inputImage, atl::NIL, (float)thresholdValue, atl::NIL, 0.0, image1);
 avl::RotateImage(image1, (float)rotateAngle, avl::RotationSizeMode::Fit,
  avl::InterpolationMethod::Bilinear, false, processedImage);
}

// callback
void on_trackbar(int, void*)
{
 ProcessImage();

 avl::AvlImageToCVMat_Linked(processedImage, cvImage);
 cv::imshow("CV Result Window", cvImage);
}

int main(void)
{
 // Load AVL image
 avl::Image monoImage, rgbImage;
 avl::TestImage(avl::TestImageId::Lena, rgbImage, monoImage);
 avl::DownsampleImage(monoImage, 1, inputImage);
 thresholdValue = 128;
 rotateAngle = 0;

 // Create OpenCV Gui
 cv::namedWindow("Settings Window", 1);
 cv::resizeWindow("Settings Window", 300, 80);
 cv::createTrackbar("Threshold", "Settings Window", &thresholdValue, 255, on_trackbar);
 cv::createTrackbar("Rotate", "Settings Window", &rotateAngle, 360, on_trackbar);

 // set trackbar
 on_trackbar(0, 0);

 cv::waitKey(0);
 return 0;
}
```

Example: avl::Image from pointer to image data

It is also possible to create an avl::Image object using a pointer to image data, without copying memory blocks. This, however, requires compatible memory representations of images and proper information about the image being created has to be provided.

The constructor shown below should be used for this operation:

```
Image::Image(int width, int height, int pitch, PlainType::Type type, int depth, void* data,
 atl::Optional< const avl::Region& > inRoi = atl::NIL);
```

Please note that all of the XxxToXxx_Linked functions do not copy data and the user has to take care of freeing such data. See also the usage example in OpenCV converter above. Functions AvlImageToCVMat_Linked and CVMatToAvlImage_Linked do not copy data.

Displaying Images Directly on WinAPI/MFC Device Context (HDC)

For convenience, there is also a function that directly displays an image on a WinAPI device context (HDC). This function is defined in the header "AVLConverters/AVL_Winapi.h" as:

```
void DisplayImageHDC(HDC inHdc, avl::Image& inImage, float inZoomX = 1.0, float inZoomY = 1.0);
```

For sample program showing how to use this function, please refer to the official example in the "06 WinAPI tutorial" directory.

Loading Aurora Vision Studio Files (AVDATA)

Aurora Vision Studio has its own format for storing arbitrary objects - the AVDATA format. It is used for storing elements of the program (paths, regions etc.) automatically, or manually when using "Export to AVDATA file" option or the **SaveObject** and **LoadObject** generic filters.

Aurora Vision Library can load and save several types of objects in AVDATA format. This is done using dedicated functions, two corresponding for each supported type. The functions start with **Load** and **Save** and accept two parameters - a filename and an object reference - for loading or saving.

```
void LoadRegion
(
 const File& inFilename,  //:Name of the source file
 Region& outRegion  //:Deserialized output Region
);
```

```
void SaveRegion
(
 const Region& inRegion,  //:Region to be serialized
 const File& inFilename  //:Name of the target file
);
```

The supported types include:

- Region
- Profile
- Histogram
- SpatialMap
- EdgeModel
- GrayModel
- OcrMlpModel
- OcrSvmModel
- Image*

Because the **LoadImage** function is a more general mechanism for saving and loading images into common file formats (like BMP, JPG or PNG), the functions for loading and saving avl::Image as AVDATA are different:

```
void LoadImageObject
(
 const File& inFilename,  //:Name of the source file
 Image& outImage  //:Deserialized output Image
);
```

```
void SaveImageObject
(
 const Image& inImage,  //:Image to be serialized
 const File& inFilename  //:Name of the target file
)
```

Simple types like **Integer**, **Real** or **String** can be stored in files in textual form - by setting **inStreamMode** to *Text* when using **SaveObject** - this can be read by formatted input output in C/C++ (for example using functions from the scanf family).

Working with GenICam GenTL Devices

Introduction

GenICam GenTL is a standard that defines a software interface encapsulating a transport technology and that allows applications to communicate with general vision devices without prior knowledge of its communication protocol. GenTL supporting application (a GenTL consumer) is able to load a third party dynamic link library (a GenTL provider) that is a kind of a "driver" for a vision device. GenICam standard allows to overcome differences with communication protocols and technologies, and allows to handle different devices in same common way. However application still needs to be aware of differences in device capabilities and be prepared to cooperate with specific device class or device model.

Aurora Vision Library contains a built-in GenTL subsystem that helps and simplifies usage of a GenTL device in vision application. AVL GenTL subsystem helps in loading provider libraries, enumerating GenTL infrastructure, managing acquisition engine and frame buffers, converting image formats and implements GenAPI interface.

In order to be able to use a GenTL provider it needs to be properly registered (installed) in local system. Usually this task is performed by an installer supplied by a device vendor. Please note that a 32bit application requires a 32 bit provider library and a 64 bit application requires respectively a 64 bit provider library. A registered GenTL provider is characterized by a file with ".cti" extension. Path to cti library containing folder is stored in an environmental variable named "GENICAM_GENTL32_PATH" ("GENICAM_GENTL64_PATH" for 64 bit providers).

Basic Usage

Functions designed for GenTL support can be found in GenTL and GenApi categories. A basic application will first use a GenTL_OpenDevice function to open a device instance (to establish the connection) and to request a handle for further operations on the device. This handle can be than used with GenApi functions to access device specific configuration and manage them. When the device identifiers are not fully known, or can dynamically change at runtime a GenTL_FindDevices function can be first used to enumerate available GenTL devices.

To start streaming video out of configured device a GenTL_StartAcquisition function must be executed. After this sequentially upcoming images can be retrieved with GenTL_ReceiveImage or GenTL_TryReceiveImage functions. Images will be stored in an input FIFO queue. Not retrieved images (on queue overflow) will be dropped starting from the oldest one. To stop image acquisition a GenTL_StopAcquisition function should be called. Image acquisition can be stopped and than started again multiple times for same device with eventual configuration change in between (some parameters can be locked for time of image streaming).

To release the device instance its handle need to be closed with GenTL_CloseHandle function.

Advanced Usage

When more information need to be known about GenTL environment its structure can be explored using GenTL_EnumLibraries, GenTL_GetLibraryDescriptor, GenTL_EnumLibraryInterfaces, GenTL_GetInterfaceDescriptor functions.

When extended information or configuration, specific for GenTL provider or transport technology need to be accessed, following functions can be considered: GenTL_OpenLibrarySystemModuleSettings, GenTL_OpenInterfaceModuleSettings, GenTL_OpenDeviceModuleSettings, GenTL_OpenDeviceStreamModuleSettings.

Additional Requirements

When using GenTL subsystem of Aurora Vision Library a "Genicam_Kit.dll" file is required to be in range of application. This file (selected for 32/64 bit) can be found in Aurora Vision Library SDK "bin" directory.

Processing Images in Worker Thread

Introduction to the Problem

Aurora Vision Library is a C++ library, that is designed for efficient image processing in C++ applications. A typical application uses a single primary thread for the user interface and can optionally use additional worker threads for data processing without freezing the main window of the application. Images processing can be a time-consuming task, so performing it in a separate worker thread is recommended, especially for processing performed in continuous mode.

Processing images in a worker thread is asynchronous and it means that accessing the resources by the worker thread and the main thread has to be coordinated. Otherwise, both threads could access the same resource at the same time, what would lead to unpredictable data corruption. The typical resource that has to be protected to be thread-safe is the image buffer. Typically, the worker thread of the vision application has a loop. In this loop it grabs images from a camera and does some kind of processing. Images are stored in memory of a buffer as *avl::Image* data. The main thread (UI thread) presents the results of the processing and/or images from the camera. It has to be ensured that the images are not read by the UI thread and processed by the worker thread at the same time.

Please note that the GUI controls should never be accessed directly from the worker thread. To display the results of the worker thread processing in the GUI, a resource access control has to be used.

Example Application and Image Buffer Synchronization

This article does not present the rules of multithreaded programming. It only focuses on the most typical aspects of it, that can be met when writing applications with Aurora Vision Library. An example application that uses the main thread and the worker thread can be found among the examples distributed with Aurora Vision Library. It is called *MFC Simple Streaming* and the easiest way to open it is by opening *Examples* directory of Aurora Vision Library from the *Start Menu*. The application is located in *03 GigEVision tutorial* subdirectory. It is a good template for other vision applications processing images in a separate thread. It is written using *MFC*, but the basics of multithreading stay the same for all other technologies.

There are many techniques of synchronization of a shared resources access in a multithreading environment. Each of them is good as long as it protects the resources in all states that the application can be in and as long as it properly handles thrown exceptions, application closing etc.

In the example application, the main form of the application has a private field called *m_videoWorker* that represents the worker thread:

```
class ExampleDlg : public CDialog
{
private:
 (...)
 GigEVideoWorker m_videoWorker;
 (...)
}
```

The *GigEVideoWorker* class contains the image buffer:

```
class GigEVideoWorker
{
 (...)
private:
 avl::Image m_imageBuffer;
 (...)
}
```

This is the image buffer that contains the image received from the camera that needs to be protected from parallel access from worker thread and from the main thread that displays the image in the main form. The access synchronization is internally achieved using critical section and *EnterCriticalSection* and *LeaveCriticalSection* functions of the Windows operating system. When one thread calls the *GigEVideoWorker::LockResults()* function, it enters the critical section and no other thread can access the image buffer until the thread that got the lock calls *GigEVideoWorker::UnlockResults()*. When one thread enters the critical section, other threads that try to enter the critical section will be suspended (blocked) until the one leaves the critical section.

Using functions like *GigEVideoWorker::LockResults()* and *GigEVideoWorker::UnlockResults()* is a good choice for protecting the image buffer from accessing by multiple threads, but what if due to an error in the code the resource is locked but never unlocked? It can happen for example in a situation when an exception is thrown inside the critical section and the code lacks the *try/catch* statement in the function that locks and should unlock the resource. In the example application this problem has been resolved using the *RAII* programming idiom. *RAII* stands for *Resource Acquisition Is Initialization* and in short it means that the resource is acquired by creating the synchronization object and is released by destroying it. In the example application being described here, there is the class called *VideoWorkerResultsGuard*. It exclusively calls the previously mentioned *GigEVideoWorker::LockResults()* and *GigEVideoWorker::UnlockResults()* functions in constructor and destructor. The instance of this

VideoWorkerResultsGuard class is the synchronization object. The code of the class is listed below.

```
class VideoWorkerResultsGuard
{
private:
 GigEVideoWorker& m_object;

 VideoWorkerResultsGuard( const VideoWorkerResultsGuard& ); // = delete

public:
 explicit VideoWorkerResultsGuard( GigEVideoWorker& object )
 : m_object(object)
 {
  m_object.LockResults();
 }

 ~VideoWorkerResultsGuard()
 {
  m_object.UnlockResults();
 }
};
```

It can be easily seen that when the object of *VideoWorkerResultsGuard* is created, the thread that creates it calls the *LockResults()* function and by that it enters the critical section protecting the image buffer. When the object is destroyed, the thread leaves the critical section. Please note that the destructor of every object is automatically called in C++ when the automatic variable goes out of scope. It also covers the cases, when the variable goes out of scope because of the exception thrown from within of the critical section. Using *RAII* pattern allows programmer to easily synchronize the access to shared resources from multiple threads. When a thread needs to access a shared image buffer, it has to create the *VideoWorkerResultsGuard* object and destroy it (or let it be destroyed automatically when the object goes out of scope) when the access to the image buffer is no longer needed. The example usage of this synchronization looks as follows:

```
// Retrieve the results under lock.
{
 VideoWorkerResultsGuard guard(m_videoWorker);
 (...)
 avl::AVLImageToCImage(m_videoWorker.GetLastResultData(), width, height, false, m_lastImage);
 (...)
}
```

The method *GetLastResultData()* returns the reference to the shared image buffer. It can be safely used thanks to the usage of *VideoWorkerResultsGuard* object.

Notifications about Image Ready to Display

Another issue that needs to be considered in a typical application that processes images and uses a worker thread is notifying the main thread that the image processed by the worker thread is ready to display. Such notifications can be implemented in several ways. The one that has been used in the example application is using system function *PostMessage()*. When the worker thread has the image ready for presentation, it copies it to the *m_lastResultData* buffer (this is the protected one) and posts the notification message to the main window of the application:

```
//
// TODO: Compute the result data and put them in the shared buffer (just copy the source image).
//
m_lastResultData = m_imageBuffer;

// Send notification message
if (PostMessage(m_hNotificationWindow, m_notificationMessage, 0, NULL))
{
 m_lastResultProcessed = false;
}
```

The message is received by the main (UI) thread. Once it's received, the main thread acquires the access to the shared image buffer by creating the *VideoWorkerResultsGuard* object. Then, the image can be safely displayed.

The worker thread has a flag called *m_lastResultProcessed*. The flag set to *false* indicates that the notification about image ready to display had been posted to the main thread but the main thread has not processed (displayed) the image yet. The flag is set to false just after posting the notification message. The main thread sets the flag back to *true* using *NotificationGiveFeedback()* function:

```
void GigEVideoWorker::NotificationGiveFeedback( void )
{
 VideoWorkerResultsGuard guard(*this);
 m_lastResultProcessed = true;
}
```

Once the worker thread has sent the notification message, it can acquire and perform the next frame from the camera, but there's no point in sending the next notification until the previous is performed by the UI thread. Sending the new notifications without performing the old ones could lead to cumulating them in the messages queue of the main window. This is why the worker thread of the example application checks if the previous notification message has been performed and sends the next one only if the processing of the previous is finished:

```
if (m_lastResultProcessed && NULL != m_hNotificationWindow)
{
 // Create the result in shared buffers under lock.
 VideoWorkerResultsGuard guard(*this);
 (...)
}
```

Please note that the flag is also protected by the *VideoWorkerResultsGuard* synchronization object, so the main thread cannot set it to *true* in the moment directly after the worker thread posted the notification message.

Issues of Multithreading

There are two primary issues to consider when using worker thread(s). The first one is destroying data by unsynchronized access from multiple threads and the second one is a deadlock that can appear when there are two (or more) resources to be synchronized.

Securing data integrity by the thread synchronization mechanisms has been shortly described in this article and is implemented in the example application distributed with Aurora Vision Library. As a rule of a thumb, please assume that every image that can be accessed from more then one thread should be protected by some kind of synchronization. We recommend the standard C++ *RAII* pattern as an easy to use and secure solution.

The example application described in this article contains only one resource – a critical section represented by the *VideoWorkerResultsGuard* class, but of course there may exist some applications where there is more then one resource to share. In such cases, the synchronization of the threads has to be implemented very carefully because there is a danger of deadlock that can be a result of bad implementation. If your application freezes (stops responding) and you have more then one synchronized resource, please review the synchronization code.

Troubleshooting

This article describes the most common problems that might appear when building and executing programs that use Aurora Vision Library.

Problems with Building

error LNK2019: unresolved external symbol _LoadImageA referenced in function
error C2039: 'LoadImageA' : is not a member of 'avl'

The problem is related to including the "windows.h" file. It defines a macro called *LoadImage*, which has the same name as one of the functions of Aurora Vision Library. Solution:

- Don't include both "windows.h" and "AVL.h" in a single compilation unit (cpp file).
- Use #undef LoadImage after including "windows.h".

error LNK1123: failure during conversion to COFF: file invalid or corrupt

If you encounter this problem, just disable the incremental linking (properties of the project | *Configuration Properties* | *Linker* | *General* | *Enable Incremental Linking*, set to *No (/INCREMENTAL:NO)*). This is a known issue of VS2010 and more information can be found on the Internet. Installing VS2010 Service Pack 1 is an alternative solution.

Exceptions Thrown in Run Time

Exception from the *avl* namespace is thrown

Aurora Vision Library uses exceptions to report errors in the run-time. All the exceptions are defined in *avl* namespace and derive from *avl::Error*. To solve the problem, add a *try/catch* statement and catch all *avl::Error* exceptions (or only selected derived type). Every *avl::Error* object has the *Message()* method which should provide you more detailed information about the problem. Remember that a good programming practice is catching C++ exceptions by a const reference.

```
try
{
    // your code here
}
catch (const atl::Error& er)
{
    cout << er.Message();
}
```

High CPU Usage When Running AVL Based Image Processing

When working with some AVL image processing functions it is possible that the reported CPU usage can reach 50~100% across all CPU cores even in situations when the actual workload does not justify that hight CPU utilization. This behavior is a side effect of a parallel processing back-end worker threads actively waiting for the next task. Although the CPU utilization is reported to be high those worker threads will not prevent other task to be executed when needed, so this behavior should not be a problem in most situations.

For situations when it is not desired this behavior can be changed (e.g. when profiling the application, performance testing or in any situation, when high CPU usage interfere with other system). To block the worker threads from idling for extended period of time the environment variable OMP_WAIT_POLICY must be set to the value PASSIVE, before the application is started:

```
set OMP_WAIT_POLICY=PASSIVE
```

This variable is checked when the DLLs are loaded, so setting it from the application code might not be effective.

Memory Leak Detection in Microsoft Visual Studio

When creating applications using Aurora Vision Library in Microsoft Visual Studio, it may be desirable to enable automated memory leak detection possible in Debug builds. The details of using this feature is described here: Finding Memory Leaks Using the CRT Library.

Some project types, notably MFC (Microsoft Foundation Classes) Windows application projects, have this mechanism enabled by default.

False Positives of Memory Leaks in AVL.dll

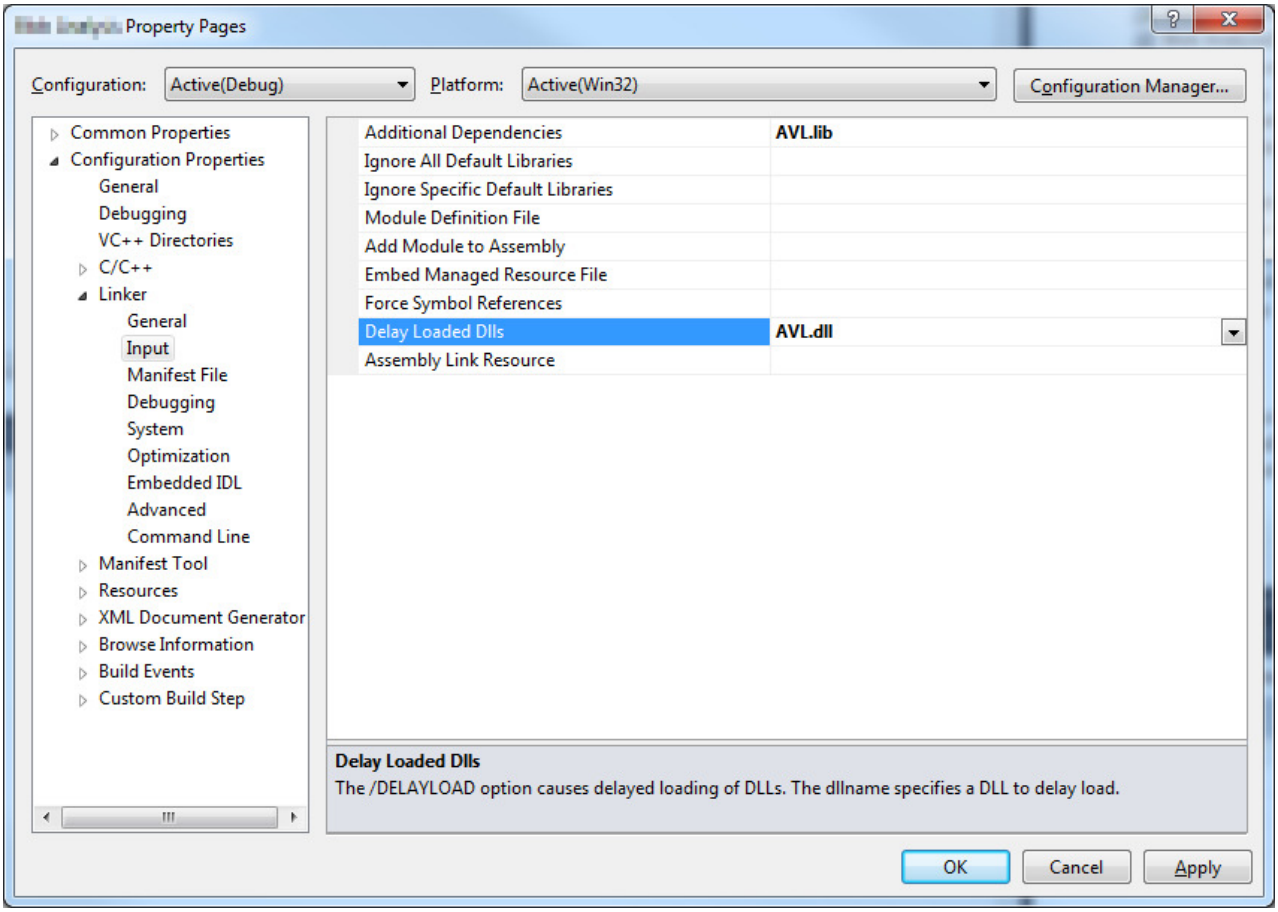
Using a default configuration, as described in Project Configuration can lead to false positives of memory leaks, which come from the AVL.dll library. The output of a finished program can look similar to the following:

```
(...)
The thread 'Win32 Thread' (0x898) has exited with code 0 (0x0).
The thread 'Win32 Thread' (0x168c) has exited with code 0 (0x0).
Detected memory leaks!
Dumping objects ->
{5573} normal block at 0x00453DB8, 8 bytes long.
 Data: <        > 01 00 00 00 00 00 00 00
{5572} normal block at 0x00453D68, 20 bytes long.
 Data: <D]NU         =E > 44 5D 4E 55 CD CD CD CD 02 00 00 00 B8 3D 45 00
{5571} normal block at 0x00453C18, 4 bytes long.
 Data: <X NU> 58 06 4E 55
(...)
```

These are not actual memory leaks, but internal resources of AVL.dll, which are not yet released when the memory leaks check is being run. Because there are many such allocated blocks reported, the actual memory leaks in your program can pass unnoticed.

Solution: Delayed Loading of AVL.dll

To avoid these false positives, AVL.dll should be configured to be delay loaded. This can be done in the Project Properties, under
Configuration Properties » Linker » Input:



Further Consequences

With this configuration, your program will not try to load AVL.dll until it uses the first function from Aurora Vision Library. This will be also connected with license checking.

The program will stop if AVL.dll is missing: if AVL.dll was not delay loaded, this would happen at start time (the program would refuse to run). This allows the program to work without AVL.dll, and use it only when it is available. The availability of AVL.dll can be checked beforehand, using LoadLibrary or LoadLibraryEx functions.

ATL Data Types Visualizers

Data Visualizers

Data visualizers present data during the debugging session in a human-friendly form. Microsoft Visual Studio allows users to write custom visualizers for C++ data. Aurora Vision Library is shipped with a set of visualizers for the most frequently used ATL data types: *atl::String*, *atl::Array*, *atl::Conditional* and *atl::Optional*.

Visualizers are automatically installed during installation of Aurora Vision Library and are ready to use, but they are also available at *atl_visualizers* subdirectory of Aurora Vision Library installation path.

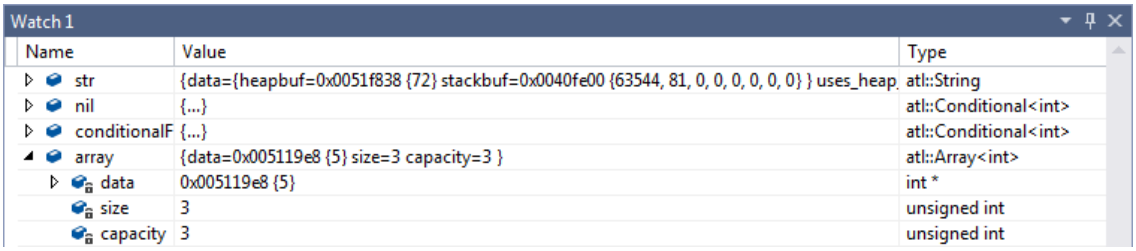
For more information about visualizers, please refer to the MSDN.

Example ATL data visualization

Please see the example variables definition below and their visualization without and with visualizers.

```
atl::String str = L"Hello world";
atl::Conditional nil = atl::NIL;
atl::Conditional conditionalFive = 5;
atl::Array array(3, 5);
```

Data preview without ATL visualizers installed:



The same data presented using AVL visualizers:

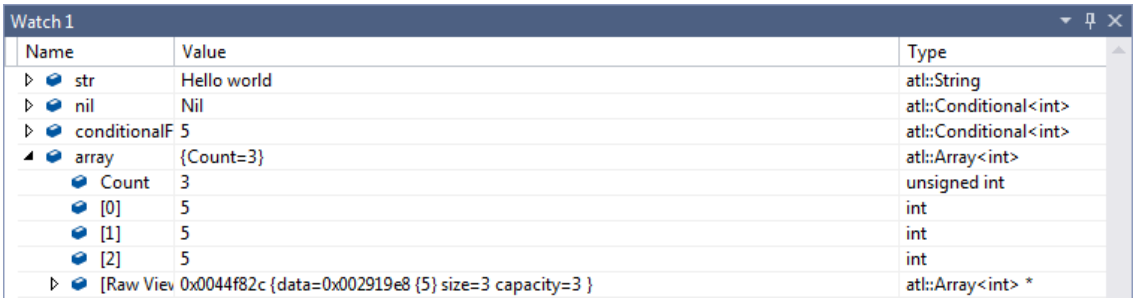


Image Watch extension

For Microsoft Visual Studio 2015, 2017 and 2019 an extension Image Watch is available. Image Watch allows to display images during debugging sessions in window similar to "Locals" or "Watch". To make Image Watch work correctly with *avl::Image* type, Aurora Vision Library installer provides avl::Image visualizer for Image Watch. If one have Image Watch extension and AVL installed, preview of images can be enabled by choosing "View->Other Windows->Image Watch" from Microsoft Visual Studio menu.

avl::Image description for Image Watch extension is included in *atl.natvis* file, which is stored in *atl_visualizers* folder in Aurora Vision Library installation directory. *atl.natvis* file is installed automatically during Aurora Vision Library installation.

When program is paused during debug session, all variables of type avl::Image can be displayed in Image Watch window, as shown below:

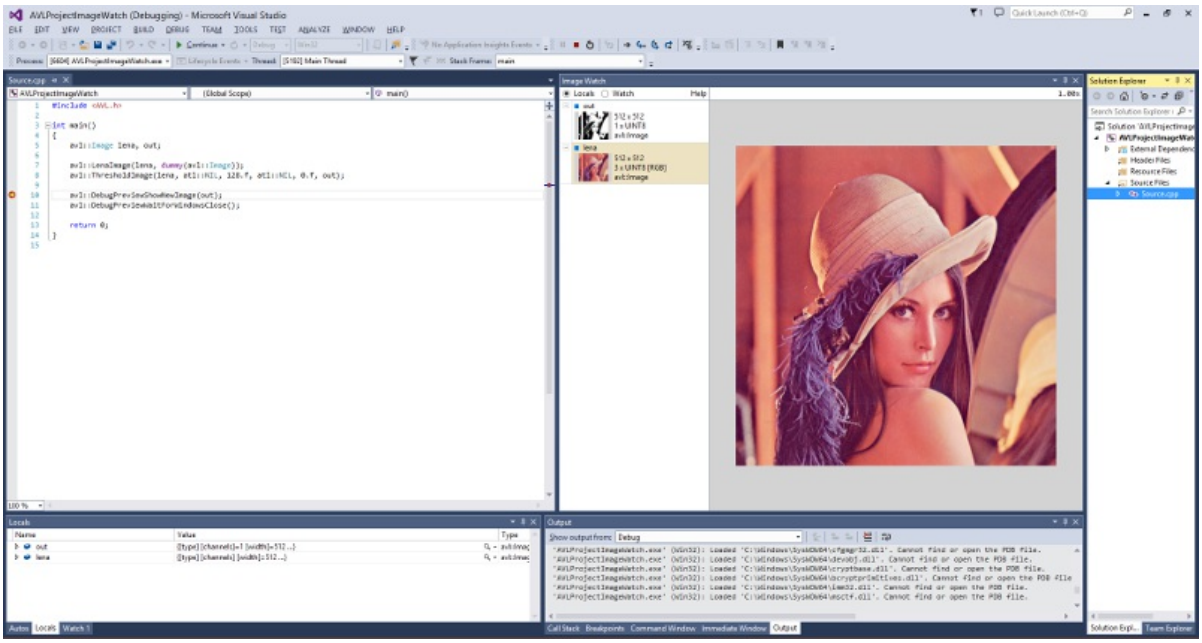


Image displayed inside Image Watch can be zoomed. When the close-up is large enough, decimal values of pixels' channel will be displayed. Hexadecimal values can be displayed instead, if appropriate option from context menu is selected.

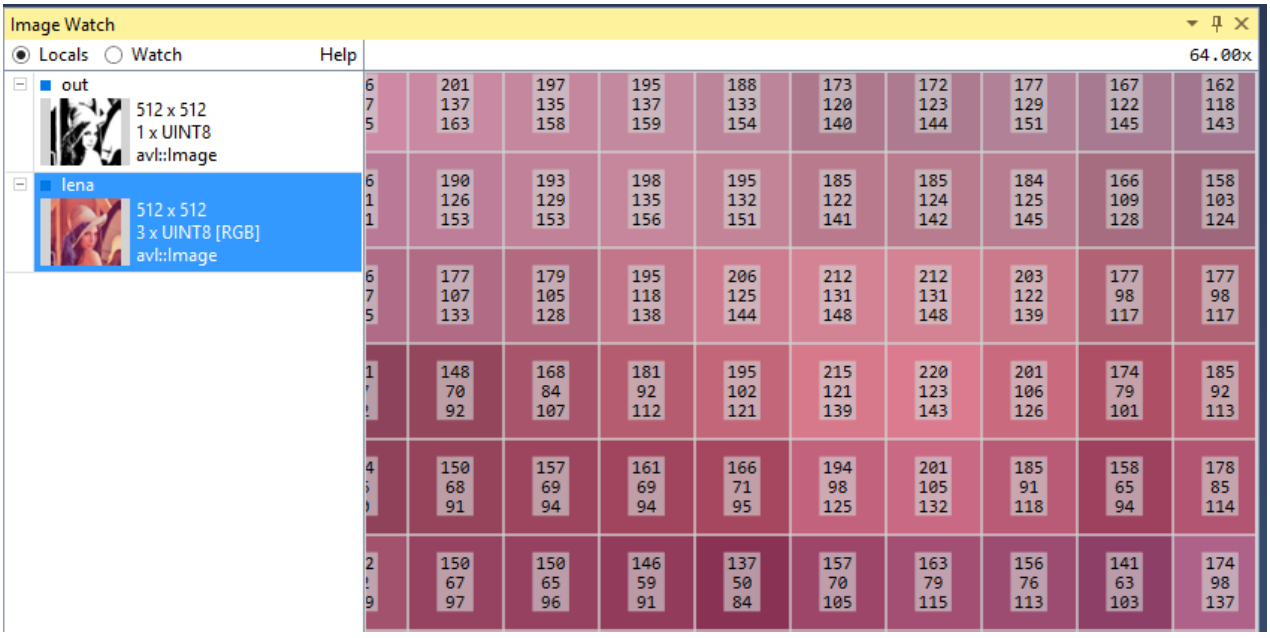


Image Watch is quite powerful tool - one can copy address of given pixel, ignore alpha channel and much more. All options are described in its documentation, which is accessible from the Image Watch site at:

- ImageWatch 2019 - for Microsoft Visual Studio 2019
- ImageWatch 2017 - for Microsoft Visual Studio 2017
- ImageWatch - for older versions of Microsoft Visual Studio

Optimizing Image Analysis for Speed

General Rules

Rule #1: Do not compute what you do not need.

- Use image resolution well fitted to the task. The higher the resolution, the slower the processing.
- Use the **inRoi** input of image processing functions to compute only the pixels that are needed in further processing steps.
- If several image processing operations occur in sequence in a confined region then it might be better to use CropImage at first.
- Do not overuse images of types other than UInt8 (8-bit).
- Do not use multi-channel images, when there is no color information being processed.
- If some computations can be done only once, move them before the main program loop, or even to a separate function.

Rule #2: Prefer simple solutions.

- Do not use Template Matching if more simple techniques as Blob Analysis or 1D Edge Detection would suffice.
- Prefer pixel-precise image analysis techniques (Region Analysis) and the *Nearest Neighbour* (instead of *Bilinear*) image interpolation.
- Consider extracting higher level information early in the program – for example it is much faster to process Regions than Images.

Rule #3: Mind the influence of the user interface.

- Note that displaying data in the user interface takes much time, regardless of the UI library used.
- Mind the Diagnostic Mode. Turn it off whenever you need to test speed. Diagnostic Mode can be turn off or on by EnableAvlDiagnosticOutputs function. One can check, if Diagnostic Mode is turned on by GetAvlDiagnosticOutputsEnabled function.
- Before optimizing the program, make sure that you know what really needs optimizing. Measure execution time or use a profiler.

Common Optimization Tips

Apart from the above general rules, there are also some common optimization tips related to specific functions and techniques. Here is a check-list:

- Template Matching: Prefer high pyramid levels, i.e. leave the **inMaxPyramidLevel** set to *atl::NIL*, or to a high value like between *4* and *6*.
- Template Matching: Prefer **inEdgePolarityMode** set not to Ignore and **inEdgeNoiseLevel** set to Low.
- Template Matching: Use as high values of the **inMinScore** input as possible.
- Template Matching: If you process high-resolution images, consider setting the **inMinPyramidLevel** to *1* or even *2*.
- Template Matching: When creating template matching models, try to limit the range of angles with the **inMinAngle** and **inMaxAngle** inputs.
- Template Matching: Consider limiting **inSearchRegion**. It might be set manually, but sometimes it also helps to use Region Analysis techniques before Template Matching.
- Do not use these functions in the main program loop: CreateEdgeModel1, CreateGrayModel, TrainOcr_MLP, TrainOcr_SVM.
- If you always transform images in the same way, consider functions from the Image Spatial Transforms Maps category instead of the ones from Image Spatial Transforms.
- Do not use image local transforms with arbitrary shaped kernels: DilateImage_AnyKernel, ErodeImage_AnyKernel, SmoothImage_Mean_AnyKernel. Consider the alternatives without the "_AnyKernel" suffix.
- SmoothImage_Median can be particularly slow. Use Gaussian or Mean smoothing instead, if possible.

Library-specific Optimizations

There are some optimization techniques that are available only in Aurora Vision Library and not in Aurora Vision Studio. These are:

In-Place Data Processing

See: In-Place Data Processing.

Re-use of Image Memory

Most image processing functions allocate memory for the output images internally. However, if the same object is provided in consecutive iterations and the dimensions of the images do not change, then the memory can be re-used without re-allocation. This is very important for the performance considerations, because re-allocation takes time which is not only significant, but also non-deterministic. Thus, it is highly advisable to move the image variable definition before the loop it is computed in:

```
// Slow code
while (...)
{
 Image image2;
 ThresholdImage(image1, atl::NIL, 128.0f, atl::NIL, 0.0f, image2);
}
```

```
// Fast code
Image image2;
while (...)
{
 ThresholdImage(image1, atl::NIL, 128.0f, atl::NIL, 0.0f, image2);
}
```

```
// Fast code (also in the first iteration)
Image image2(752, 480, PlainType::UInt8, 1, atl::NIL); // memory pre-allocation (dimensions must be known)
while (...)
{
 ThresholdImage(image1, atl::NIL, 128.0f, atl::NIL, 0.0f, image2);
}
```

Skipping Background Initialization

Almost all image processing functions of Aurora Vision Library have an optional **inRoi** parameter, which defines a region-of-interest. Outside this region the output pixels are initialized with zeros. Sometimes, when the rois are very small, the initialization might take significant time. If this is an internal operation and the consecutive operations do not read that memory, the initialization can be skipped by setting IMAGE_DIRTY_BACKGROUND flag in the output image. For example, this is how dynamic thresholding is implemented internally in AVL, where the out-of-roi pixels of the **blurred** image are not meaningful:

```
Image blurred;
blurred.AddFlags(IMAGE_DIRTY_BACKGROUND);
SmoothImage_Mean(inImage, inRoi, inSourceRoi, atl::NIL, KernelShape::Box, radiusX, radiusY, blurred);
ThresholdImage_Relative(inImage, inRoi, blurred, inMinRelativeValue, inMaxRelativeValue, inFuzziness, outMonoImage);
```

Library Initialization

Before you call any AVL function it is recommended to call the InitLibrary function first. This function is responsible for precomputing library's global data. If it is not used explicitly, it will be called within the first invocation of any other AVL function, taking some additional time.

Configuring Parallel Computing

The functions of Aurora Vision Library internally use multiple threads to utilize the full power of multi-core processors. By default they use as many threads as there are physical processors. This is the best setting for majority of applications, but in some cases another number of threads might result in faster execution. If you need maximum performance, it is advisable to experiment with the ControlParallelComputing function with both higher and lower number of threads. In particular:

- If the number of threads is **higher** than the number of physical processors, then it is possible to utilize the Hyper-Threading technology.
- If the number of threads is **lower** than the number of physical processors (e.g. 3 threads on a quad-core machine), then the system has at least one core available for background threads (like image acquisition, GUI or computations performed by other processes), which may improve its responsiveness.

Configuring Image Memory Pools

Among significant factors affecting function performance is memory allocation. Most of the functions available in Aurora Vision Library re-use their memory buffers between consecutive iterations which is highly beneficial for their performance. Some functions, however, still allocate temporary image buffers, because doing otherwise would make them less convenient in use. To overcome this limitation, there is the function ControlImageMemoryPools which can turn on a custom memory allocator for temporary images.

There is also a way to pre-allocate image memory before first iteration of the program starts. For this purpose use the InspectImageMemoryPools function at the end of the program, and – after a the program is executed – copy its **outPoolSizes** value to the input of a ChargeImageMemoryPools function executed at the beginning. In some cases this will improve performance of the first iteration of program.

Using GPGPU/OpenCL Computing

Some functions of Aurora Vision Library allow to move computations to an OpenCL capable device, like a graphics card, in order to speed up execution. After proper initialization, OpenCL processing is performed completely automatically by suitable functions without changing their use pattern. Refer to "Hardware Acceleration" section of the function documentation to find which functions support OpenCL processing and what are their requirements. Be aware that the resulting performance after switching to an OpenCL device may vary and may not always be a significant improvement relative to CPU processing. Actual performance of the functions must always be verified on the target system by proper measurements.

To use OpenCL processing in Aurora Vision Library the following is required:

- a processing device installed in the target system supporting OpenCL C language in version 1.1 or greater,
- a proper and up-to-date device driver installed in the system,
- a proper OpenCL runtime software provided by its vendor.

OpenCL processing is supported for example in the following functions: RgbToHsi, HsiToRgb, ImageCorrelationImage, DilateImage_AnyKernel.

To enable OpenCL processing in functions an AvsFilter_InitGPUProcessing function must be executed at the beginning of a program. Please refer to that function documentation for further information.

Deep Learning Training API

Note: This article is related to the C++ Deep Learning API for Feature Detection and Anomaly Detection techniques in 5.6 version only.

Table of contents:

Overview

The Deep Learning API provides a comprehensive framework for training and deploying Deep Learning models, focused on feature detection tasks. It offers an object-oriented interface that simplifies the complexities of configuring and managing Deep Learning operations. Whole API declaration is located in Api.h file. under avl namespace.

Namespaces

- avl: Main namespace containing all public-facing classes and types.

Classes and Types

avl::DetectFeaturesTraining

This is the primary class users should interact with for feature detection training. It is derived from TrainingBase and provides specialized methods and properties for configuring and managing feature detection tasks.

Constructors

```
DetectFeaturesTraining();
```

```
explicit DetectFeaturesTraining(const atl::String& url);
```

Configuration Methods

Training configuration can be performed in two ways:

1. **Via Set Methods:** Configuration can be done using methods like SetDevice, SetNetworkDepth, and other Set* methods. If a specific Set* method is not called, the default value will be used.
2. **Via JSON File:** Use the ParseConfigFromFile method to load configuration from a JSON file.

Enums

- SetType: Specifies the dataset role (Train, Valid, Test, Unknown).
- DeviceType: Defines the hardware device for training (CUDA, CPU).
- ModelTypeId: Identifies the model type (e.g., DetectFeatures, AnomalyDetection2SimilarityBased).

Functions

ParseConfigFromFile

Loads configuration from a JSON file.

```
void ParseConfigFromFile(const atl::String& jsonConfigFilePath);
```

Example of JSON file configuration below.

StartTraining

Begins the training process.

```
void StartTraining();
```

SaveModel

Saves the trained model to disk in two formats:

- A model state file (.pte) for internal use and training continuation
- A Weaver model file (.avdlmodel) for deployment in applications

Method Signatures

```
void SaveModel(const atl::Optional<atl::String>& modelDirectoryPath = atl::NIL, const bool overwritePreviousModel =
false);
void SaveModel(const char* modelDirectoryPath, const bool overwritePreviousModel = false);
```

Parameters

- modelDirectoryPath: Optional path to a directory where model files will be saved. If not provided (default), models are saved in the default directory: [current working directory]/Model/models/
- overwritePreviousModel: When set to true, any existing model files at the destination will be overwritten. When false (default), and model files exist, an error will be raised.

Helper Methods

After saving, you can retrieve the exact paths to the saved model files using:

- GetModelStateFilePath(): Returns the path to the .pte model state file
- GetModelWeaverFilePath(): Returns the path to the .avdlmodel Weaver model file

Usage Examples

```
// 1. Save to default location:
training.SaveModel();

// 2. Save to default location and overwrite existing files:
training.SaveModel(atl::NIL, true);

// 3. Save to custom location:
training.SaveModel("C:/My/Models/Path");

// 4. Save to custom location and overwrite existing files:
training.SaveModel("C:/My/Models/Path", true);

// 5. Get saved file paths:
std::cout << "Model State (.pte) saved to: " << training.GetModelStateFilePath().CStr8() << std::endl;
std::cout << "Weaver Model (.avdlmodel) saved to: " << training.GetModelWeaverFilePath().CStr8() << std::endl;
```

LoadModel

Loads a previously saved model state (.pte) file for inference.

Method Signatures

```
void LoadModel(const atl::String& modelFilePath);
void LoadModel(const char* modelFilePath);
```

Parameters

- modelFilePath: Path to a model state file (.pte) to load. The method will load the specified model file for inference operations.

Functionality

Loading a model allows you to:

- Perform inference on new images using a trained model

Usage Examples

```
// 1. Load a specific model file:
training.LoadModel("C:/My/Models/Path/model.pte");

// 2. Load using the path from a previous save operation (PTE file):
training.SaveModel();  // Save first
training.LoadModel(training.GetModelStateFilePath());  // Load the saved model
```

Important Notes

- This method loads only the model state (.pte) file used for training and inference within this API
- The Weaver model (.avdlmodel) files created by SaveModel() are for deployment in production applications
- After loading, the model is immediately ready for inference with InferAndGrade()

GetModelStateFilePath & GetModelWeaverFilePath

Helper methods to retrieve the paths of saved model files.

Method Signatures

```
atl::String GetModelStateFilePath();
atl::String GetModelWeaverFilePath();
```

Return Values

- GetModelStateFilePath(): Returns the full path to the saved model state file (.pte)
- GetModelWeaverFilePath(): Returns the full path to the saved Weaver model file (.avdlmodel)

Usage

These methods can be called only **after SaveModel()** to get the exact file paths where the models were saved:

```
training.SaveModel("./MyModels");
std::cout << "PTE model saved to: " << training.GetModelStateFilePath().CStr8() << std::endl;
std::cout << "Weaver model saved to: " << training.GetModelWeaverFilePath().CStr8() << std::endl;
```

InferAndGrade

Performs inference and grades the results. If the InferResultReceived method is overridden, it will be utilized during the inference process.

```
void InferAndGrade(
    const atl::String& imageFilePath,
    const Annotation& annotation,
    const atl::Optional<avl::Region>& roi = atl::NIL,
    const atl::Optional<atl::Array<atl::String>>& setNames = atl::NIL);
```

Parameters

- imageFilePath: Path to the image for inference.
- annotation: Annotation with class name (and optional region) used for grading or context.
- roi (optional): Region of interest. When omitted or atl::NIL, the full image is used.
- setNames (optional): Logical grouping / tag list for evaluation summary (e.g. custom test subsets).

Handling Events

To communicate with the user during training and inference, several events are available:

- TrainingProgressReceived(double progress): Called to update progress during training.
- InferResultReceived(const atl::Array<avl::Image>&): Invoked when inference results are available.

Usage Example

Below is an example demonstrating how to use the API for training a feature detection model:

```
#include "Api.h"
#include <iostream>

using namespace avl;

class MyTraining : public DetectFeaturesTraining
{
```

```
public:
 MyTraining()
 {
 }

 void TrainingProgressReceived(double progress) override
 {
  std::cout << "Progress: " << progress << std::endl;
 }

 void InferResultReceived(const atl::Array<avl::Image>& inferResultImages) override
 {
  (void)inferResultImages;
  //  for (const auto& inferResultImage : inferResultImages)
  // std::cout << "InferResult: " << "Width: " << inferResultImage.Width() <<, " Height: " <<
inferResultImage.Height() << std::endl;
 }
};

int main()
{
 MyTraining training;

 // Training dataset
 auto myTrainingSamples = atl::Array<atl::String>();
 myTrainingSamples.PushBack("Images/train/010.png");
 myTrainingSamples.PushBack("Images/train/011.png");
 myTrainingSamples.PushBack("Images/train/012.png");

 // Validation dataset
 auto myValidationSamples = atl::Array<atl::String>();
 myValidationSamples.PushBack("Images/valid/020.png");
 myValidationSamples.PushBack("Images/valid/021.png");
 myValidationSamples.PushBack("Images/valid/022.png");

 // Test dataset
 auto myTestSamples = atl::Array<atl::String>();
 myTestSamples.PushBack("Images/test/140.png");
 myTestSamples.PushBack("Images/test/141.png");
 myTestSamples.PushBack("Images/test/142.png");

 // Set names for samples used for infer and grade
 auto mySetNames = atl::Array<atl::String>();
 mySetNames.PushBack("my test set 1");
 mySetNames.PushBack("my test set 2");

 // Create a simple annotation mask for the training samples.
 atl::String myClassName = "thread";
 const int width = 648; // Example width and height, should match your training images
 const int height = 486;
 avl::Region myRegion(width, height);
 for (int y = height / 4; y < (height / 4 + height / 2); ++y)
  myRegion.Add(width / 4, y, (width / 4 + width / 2));
 auto myAnnotation = Annotation(myClassName, myRegion);

 // Set training configuration
 // MANUALLY:
 training.SetNetworkDepth(3);
 training.SetIterations(1);
 training.SetDevice(DeviceType::CUDA);
 training.SetToGrayscale(true);
 training.SetAugNoise(5.5);
 // training.SetClassNames(myClassName);  //Optional

 // Or from file:
 // training.ParseConfigFromFile("detect_features_config.json");

 // OPTIONAL:
 // training.Configure(); // Optional
 // training.GetConfig(); // Call `training.Configure();` before `training.GetConfig()` otherwise it will use default
config

 for (const auto& sample : myTrainingSamples)
  training.SetSample(sample, myAnnotation, SetType::Train);

 for (const auto& sample : myValidationSamples)
  training.SetSample(sample, myAnnotation, SetType::Valid);

 training.StartTraining();

 training.SaveModel();
 std::cout << "Model State (.pte) saved into file: " << training.GetModelStateFilePath().CStr8() << std::endl;
```

```
 std::cout << "Model Weaver (.avdlmodel) saved into file: " << training.GetModelWeaverFilePath().CStr8() << std::endl;

 // Load the model state for inference (use .pte file, not .avdlmodel)
 training.LoadModel(training.GetModelStateFilePath());

 for (const auto& sample : myTestSamples)
  training.InferAndGrade(sample, myAnnotation, atl::NIL, mySetNames);

 return 0;
}
```

JSON Configuration Example

Below is an example of JSON Configuration File for a feature detection model:

```
{
    "device": "cuda",
    "device_id": 0,
    "is_continuation": false,
    "network_depth": 3,
    "iterations": 2,
    "min_number_of_tiles": 6,
    "need_to_convert_samples": false,
    "stop.training_time_s": 0,
    "stop.validation_value": 0.0,
    "stop.stagnant_iterations": 0,
    "feature_size": 96,
    "aug.rotation": 0.0,
    "aug.scale.min": 1.0,
    "aug.scale.max": 1.0,
    "aug.shear.vertical": 0.0,
    "aug.shear.horizontal": 0.0,
    "aug.flip.vertical": false,
    "aug.flip.horizontal": false,
    "aug.noise": 2.0,
    "aug.blur": 0,
    "aug.luminance": 0.04,
    "aug.contrast": 0.0,
    "to_grayscale": false,
    "downsample": 2,
    "is_mega_tiling": false,
    "mega_tile_size": 128,
    "class_names": "thread",
    "adv.class_names_sep": ";"
}
```

Best Practices

- Use DetectFeaturesTraining for feature detection tasks instead of directly using TrainingBase.
- Extend DetectFeaturesTraining for custom behavior during training.
- Ensure balanced datasets for training and validation.
- Use callback methods to monitor training progress.

Limitations and Notes

- The ExportQuantizedModel method is not supported for DetectFeaturesTraining.
- Configuration can be done through property setters or by loading a JSON configuration file.
- SaveModel() creates two files: a .pte file for training/inference and a .avdlmodel file for deployment.
- LoadModel() only loads .pte files for inference operations within this API.
- Weaver model files (.avdlmodel) are intended for deployment in production applications, not for loading back into the training API.
- Provide an atl::Optional<avl::Region> ROI to limit inference processing area; pass atl::NIL (or omit parameter) to use the full image.
- An Annotation without a region is valid for tasks that don't require pixel masks.

Zebra
Aurora™ Vision

This article is valid for version 5.6.1