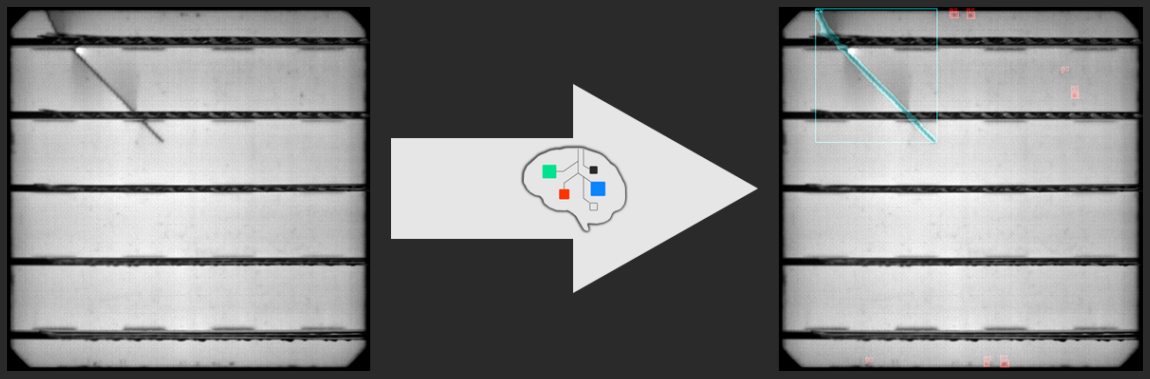
DENK API

Contents

1 Feature Set of the API



The DENK API is a set of DLLs that allow the customer to utilize DENKweit neural networks in their own software. The main features of the API include:

- Evaluation of images using DENKweit neural networks using the CPU or a GPU
- Output of the results in the form of annotated images
- Output of the results in the form of ProtoBuf serialized objects
- Filtering of the found features by certain attributes, for example a feature's area or its average gray value
- A heatmap that allows to spatially correlate the found features

2 Usage Walkthrough

2.1 General Notes

The available functions and the error codes are defined in the `denk.h` file. The `denk.h` file also contains more detailed information on the most commonly used functions. Every function will return an integer to signify whether there were any errors and data output is generally achieved using pointers. The successsful completion of a function is indicated by the return value `DE_NO_ERROR`. Every time an image is loaded, an internal dataset will be created, which will be identified by an integer index. Any later functions that work on the data will be given this index, to specify which dataset to work on. Many of the functions use `char*` pointers, which are generally compatible with the corresponding byte string format of a given programming language. For example, a python bytestring can be given to function which expects a `char*` as is. When a `char*` buffer is given to a function, it is generally succeeded by an `int*` pointer to specify its size. When calling the function, the value that this `int*` points to should contain the length of the given `char*` buffer. The API uses this information to check whether the data it wants to write fits into the buffer and it returns the error `DE_BUFFER_TOO_SMALL`, if the buffer is not sufficiently sized. If this occurs, the buffer will not be modified and the size that would have been required is written to the value pointed to by the given `int*`. If the buffer was of sufficient size and everything succeeded, the `char*` buffer will contain the written data and the value that the `int*` points to will contain the written length.

2.2 Evaluation of an Image

This section will give an example of how to evaluate an image using the API.

As a first step, the Dongle needs to be validated. This is a two step process, where first the Dongle is detected using the method:

```
int FindDongle();
```

If a Dongle was found, it needs to be checked for validty using the method:

```
int CheckDongle();
```

After that is done, the model files, which contain the neural networks, can be read. This is done using the method:

```
int ReadAllModels(
    const char* path,
    char*       proto_chars,
    int*        proto_size,
    int         device);
```

The variable `path` contains the relative or absolute path to a directory that contains the model files. All files in subdirectories and all files that do not end in `.denk` will be ignored. `proto_chars` contains a serialized protobuf object of length `proto_size`, which corresponds to the `Results` type message that is defined in the `results.proto` file. This message contains information on the networks that where read in. The variable `device` is used to specify which device is going to be used for the evaluation. The CPU always corresponds to −1 and all available GPUs are numbered successively, starting from 0.

To load an image file, one of these functions can be used:

```
int LoadImageData(
    int*  dataset_index,
    char* image_in_data,
    int   image_in_data_size);

int LoadImageDataFromRaw(
    int*  dataset_index,
    char* image_in_data,
    int   img_w,
    int   img_h,
    int   img_c);
```

Both functions create a dataset, as is described in section 2.1, and write its index into the variable pointed to by `dataset_index`. Take note that all subsequent functions that use this index do not take in an integer-pointer, but instead the integer itself. The first function takes a `char*` array of size `image_in_data_size`, which is an unparsed bytestream from an encoded image file of a supported format (PNG, JPG, TIF, BMP). The second function takes a raw bytestream of an image, with one byte per channel, in the channel format RGBRGB..., with 1, 2, 3 or 4 channels. The API assumes that the `image_in_data` array is of length `img_w` · `img_h` · `img_c`.

When the image has been loaded, it can be evaluated using the models that

were loaded in the `ReadAllModels` step. To do this, the following method needs to be called:

```
int EvaluateImage(int index);
```

After the evaluation is finished, the results can be returned in the following ways:

```
int GetResults(
    int   index,
    char* output_chars,
    int*  output_chars_size);
```

This method will write a protobuf serialized object into `output_chars` and its size into `output_chars_size`. This object can be deserialized into a `Results` message, which is defined in the file `results.proto`. Alternatively, an annotated image can be generated, using:

```
int DrawBoxes(
    int    index,
    double overlap_threshold,
    double alpha_boxes,
    double alpha_segmentations,
    char*  output_data,
    int*   output_data_size);
```

The `overlap_threshold` determines how much annotation boxes of the same feature type can overlap, before the smaller one is omitted. If a value of 1 is given, all boxes will be shown, independent of how much they overlap. `alpha_boxes` is the alpha value of the drawn annotation boxes. A value of 0 means that they will not be shown, a value of 1 means they are fully visible. Similarly, `alpha_segmentations` determines the alpha value of the pixel accurate segmentations. The final image is written into `output_data` and its size is written into the value pointed to by `output_data_size`. The written image will have the same width and height as the input image, but will always have three channels in RGBRGB... succession. Therefore, the `output_data` array needs to be of a size of at least: width $\cdot$ height $\cdot$ 3.

For any subsequent evaluations, the functions that check the Dongle and

Read the model files do not need to be rerun. The next evaluations can therefore start at the step, where the image data is loaded. After the session is finished and the calling programs is about to terminate, it is recommended to call the method:

```
int EndSession();
```

This ensures that all resources are freed properly and all sessions on the GPUs are properly terminated.

2.2.1 Minimal Summary

Upon starting the program:

```
int FindDongle();
int CheckDongle();
int ReadAllModels(
    const char* path,
    char*       proto_chars,
    int*        proto_size,
    int         device);
```

Then, for every image file to be evaluated:

```
int LoadImageData(
    int*  dataset_index,
    char* image_in_data,
    int   image_in_data_size);
int EvaluateImage(int index);
int GetResults(
    int   index,
    char* output_chars,
    int*  output_chars_size);
```

Upon closing the program:

```
int EndSession();
```

2.3 Advanced Functions

2.3.1 Automatic Index Handling

Every time an image file is loaded, a dataset is created and it is given an index. These indices will start from 0 at the beginning of the session and they will count up successively. By default, the API keeps the last 3 datasets in memory and deletes all older ones. This value can be adjusted using this method:

```
int SetCacheSize(int new_size);
```

Keep in mind that higher values will lead to a higher RAM usage, as more data is kept in memory.

2.3.2 Options JSON

To allow the customization of the API's behaviour, an internal JSON data structure is maintained by the API. At the beginning of a session, this data structure is filled with default values. Every dataset that is created will get a copy of this JSON data structure upon its creation. This structure can be read out using the method:

```
int GetJson(
    int   index,
    char* json_data,
    int*  json_size);
```

If the given index does not correspond to a valid dataset, the current default JSON is returned. The JSON currently used by a dataset can be set using the method:

```
int SetJson(
    int   index,
    char* json_data,
    int   json_size);
```

It is recommended to call this method directly after the new dataset has been created by loading an image. To change the current default JSON, that every new dataset starts with, the following method is used:

```
int SetDefaultJson(
    char* json_data,
    int   json_size);
```

When calling the `SetJson` or `SetDefaultJson` methods, any fields that are not present in the new JSON will automatically assume their default values. Keep in mind that the API does not maintain its state between terminating and opening the calling program. Therefore, this default will not be carried over between restarts of the software that is using the API.

In order to filter the found features by the criteria defined in the JSON, this method needs to be called:

```
int ProcessFeatures(int index);
```

3 Function Overview

3.1 Debug Methods

```
int BuildInfo();
int GetDebugOutputName(char* name, int* name_size);
int GetOnnxVersion(int* major, int* minor);
int GetCudaVersion(int* major, int* minor);
int PrintInt(int input);
int PrintIntFromPointer(int* input);
int GetDeviceInformation(char* proto_chars, int* proto_chars_length);
int GetImageByKey(int index, char* image_key, char* image_out_data, int*
image_out_data_size);
int GetImageDimensions(int index, char* image_key, int* w, int* h, int *c);
```

3.2 Dongle Methods

```
int FindDongle();
int CheckDongle();
int UnFindDongle();
```

3.3 Data Input

```
int LoadImageData(int* dataset_index, char* image_in_data, int
image_in_data_size);
int LoadImageDataFromRaw(int* dataset_index, char* image_in_data, int img_w, int
img_h, int img_c);
int LoadImageDataFromRawRGB(int* dataset_index, char* image_r, char* image_g,
char* image_b, int img_w, int img_h);
int LoadImageDataManualIndexHandling(int* dataset_index, char* image_in_data, int
image_in_data_size);
int DeleteDatasetManualIndexHandling(int dataset_index);
int SetCacheSize(int new_size);
int EnableImageProcessingCache(bool enable_processing_cache);
int SetDefaultJson(char* json_data, int json_size);
int SetJson(int index, char* json_data, int json_size);
int SetResults(int index, char* input_chars, int input_chars_size);
```

3.4 Data Output

```
int GetResults(int index, char* output_chars, int* output_chars_size);
int DrawBoxes(int index, double overlap_threshold, double alpha_boxes, double
alpha_segmentations, char* output_data, int* output_data_size);
int GetOriginalImageDimensions(int index, int* w, int* h, int* c);
```

```
int GetOriginalImage(int index, char* image_out_data, int* image_out_data_size);
int GetOriginalImage8UC3(int index, char* image_out_data, int*
image_out_data_size);
int GetUndrawnChangeStatus(int index, bool* undrawn_changes);
int GetCLAHEImageRaw(int index, double clip_limit, int grid_size, int mode, char*
image_out_data, int* image_out_data_size);
int SwitchBR(char* data, int size);
int GetFeatureTable(int index, char* json_data, int* json_size);
int GetHeatmapDimensions(int* w, int* h, int* c);
int GetModelHeatmap(char* model_id, int model_id_size, char* image_data, int*
image_data_size);
int GetCombinedHeatmap(char* image_data, int* image_data_size);
int ListSets();
```

3.5 Methods on Data

```
int EvaluateImage(int index);
int ProcessFeatures(int index);
int ReapplyThresholds(int index);
int UpdateHeatmaps(int index);
int ResetHeatmap();
int CalculateRLCs(int index);
```

3.6 Other Methods

```
int ReadAllModels(const char* path, char* proto_chars, int* proto_size, int
device);
int EndSession();
int PrintModelNames();
int GetJson(int index, char* json_data, int* json_size);
int CreateJsonEntries(char* json_in_data, int json_in_size, char* json_out_data,
int* json_out_size);
int SetAreaMapFromFile(char* area_map_data, int area_map_size);
int SetAreaMapFromParameters(int image_width, int image_height, int
cells_horizontal, int cells_vertical, int border_left, int border_right, int
border_top, int border_bottom);
int SetAreaMapFromColoredImage(char* area_map_data, int area_map_size);
int SetAreaMapFromNeuralNetwork(int index, const char* model_uid, double
threshold);
int CheckForAreaMap();
int WriteAreaMap(const char* filename);
int WriteColoredAreaMap(const char* filename, int colors);
int GetAreaMapSize(int* img_w, int* img_h);
int GetAreaMap(char* area_map_data, int* area_map_size);
int GetColoredAreaMap(int colors, char* area_map_data, int* area_map_size);
int GetAreaAtPosition(int x, int y, int* area);
```

```
int NumberToLetterCode(int number, char* char_code, int* char_code_len);
int LetterCodeToNumber(char* char_code, int char_code_len, int* number);
int ProcessAreaMapping(int index);
int GetAreaMapping(int index, double alpha, int colors, char* output_data, int*
output_data_size);
int SetAreaMapCorner(int corner);
int AreaToCoordinate(int area, int* x, int* y);
int MakeCoordinateMapFromGrid(int areas_horizontal, int areas_vertical, int
corner);
int MakeCoordinateMapFromEstimatedGrid(int corner);
int CheckIfSetExists(int index);
int SelectNetworkSet(char* set_name, int set_name_size);
```

3.7 Deprecated Functions

```
int ProcessCellMapping(int index);
int GetCellMapping(int index, double alpha, char* output_data, int* output_data_size);
```

4 Functions

4.1 Debug Methods

BuildInfo

```
int BuildInfo();
```

Print build information of the library file, like the build date, a build ID and the compiler flags that were active on build time.

Arguments:
–

Returns: DE_NO_ERROR on success, an error code otherwise

GetDebugOutputName

```
int GetDebugOutputName(
    char* name,
    int*  name_size);
```

When an image is evaluated, an additional debug output field is created that does not correspond to any feature class. This output will contain additional information on the evaluatin like the overall evaluation time.

Arguments:

name	Name of the debug output
name_size	Size of the content written to ”name”

Returns: DE_NO_ERROR on success, an error code otherwise

GetOnnxVersion

```
int GetOnnxVersion(
    int* major,
    int* minor);
```

The denk-API will be compiled with a specific ONNX version, which can be determined by using this function. Using networks with this ONNX version best ensures compatibility

Arguments:
major This will be set to the major part of the ONNX version
minor This will be set to the minor part of the ONNX version

Returns: DE_NO_ERROR on success, an error code otherwise

GetCudaVersion

```
int GetCudaVersion(
    int* major,
    int* minor);
```

The denk-API will be compiled with a specific CUDA version, which can be determined by using this function. The graphics card and driver need to support this CUDA version.

Arguments:
major This will be set to the major part of the CUDA version
minor This will be set to the minor part of the CUDA version

Returns: DE_NO_ERROR on success, an error code otherwise

PrintInt

```
int PrintInt(
    int input);
```

This function will print the given integer to stdout.

Arguments:
input The integer that will be printed

Returns: DE_NO_ERROR on success, an error code otherwise

PrintIntFromPointer

```
int PrintIntFromPointer(
    int* input);
```

This function will print the integer that the pointer is pointing to to stdout.

Arguments:

input	The pointer to the integer that will be printed

Returns: DE_NO_ERROR on success, an error code otherwise

GetDeviceInformation

```
int GetDeviceInformation(
    char* proto_chars,
    int*  proto_chars_length);
```

This function will give information on the devices that are available to be used with neural networks. The output is a protobuf encoded message, defined in the results.proto under the name "DeviceInformation"

Arguments:

proto_chars	The serialized protobuf object will be written into this buffer
proto_chars_length	This variable should contain the size of the buffer given with proto_chars in bytes; it will contain the amount of written data in bytes after the function has returned

Returns: DE_NO_ERROR on success, an error code otherwise

GetImageByKey

```
int GetImageByKey(
    int   index,
    char* image_key,
    char* image_out_data,
    int*  image_out_data_size);
```

Get an internal image by image_key. This function is mostly useful for debugging the DLL.

Arguments:

index	The index of the image
image_key	The key of the image
image_out_data	The buffer that the image is written to, as serialized RGB bytes
image_out_data_size	This variable should contain the size of image_out_data in bytes, to avoid buffer overruns; after the function has returned, it will contain the length of the written data

Returns: DE_NO_ERROR on success, an error code otherwise

GetImageDimensions

```
int GetImageDimensions(
    int   index,
    char* image_key,
    int*  w,
    int*  h,
    int   *c);
```

Get the dimensions of an internal image by image_key.

Arguments:

index	The index of the image
image_key	The key of the image
w	The width of the image is written to the integer pointed to by this variable
h	The height of the image is written to the integer pointed to by this variable
c	The number of channels of the image is written to the integer pointed to by this variable

Returns: DE_NO_ERROR on success, an error code otherwise

4.2 Dongle Methods

FindDongle

```
int FindDongle();
```

Find and connect to the dongle. This action needs to be performed before any of the other Dongle Methods.

Arguments:
–

Returns: DE_NO_ERROR on success, an error code otherwise

CheckDongle

```
int CheckDongle();
```

Verify the dongle. This action needs to be done once, before the first invocation of ReadAllModels.

Arguments:
–

Returns: DE_NO_ERROR on success, an error code otherwise

UnFindDongle

```
int UnFindDongle();
```

Disconnect from the dongle. Calling this function is currently not necessary.

Arguments:
–

Returns: DE_NO_ERROR on success, an error code otherwise

4.3 Data Input

LoadImageData

```
int LoadImageData(
    int*  dataset_index,
    char* image_in_data,
    int   image_in_data_size);
```

Loads the image data from a bytearray, as read directly from a PNG, JPG or TIF file. The function will give back an index that will later be used to perform operations on the image.

Arguments:

dataset_index	The dataset index will be written to the value pointed to by this pointer
image_in_data	The image data as a bytearray, as read from a PNG, JPG or TIF file
image_in_data_size	The size of image_in_data in bytes

Returns: DE_NO_ERROR on success, an error code otherwise

LoadImageDataFromRaw

```
int LoadImageDataFromRaw(
    int*  dataset_index,
    char* image_in_data,
    int   img_w,
    int   img_h,
    int   img_c);
```

Loads the image data from a raw bytearray, where every byte encodes the value of one channel, in BGR format. The dimensions of the image need to be specified to properly format the image. The size of the bytearray will be inferred from the given image dimensions as size = width * height * channels.

Arguments:

dataset_index	The dataset index will be written to the value pointed to by this pointer
image_in_data	The image data as a raw bytestream in BGR format
img_w	The width of the image
img_h	The height of the image
img_c	The number of channels of the image, currently only 3 is supported here

Returns: DE_NO_ERROR on success, an error code otherwise

LoadImageDataFromRawRGB

```
int LoadImageDataFromRawRGB(
    int*  dataset_index,
    char* image_r,
    char* image_g,
    char* image_b,
    int   img_w,
    int   img_h);
```

Loads the image data from raw bytearrays, where 3 channels of an image are given as separate bytearrays. Every color value is represented by one byte.

Arguments:

dataset_index	The dataset index will be written to the value pointed to by this pointer
image_in_data	The image data as a raw bytestream in BGR format
image_r	Raw bytearray representing the red channel of the image
image_g	Raw bytearray representing the green channel of the image
image_b	Raw bytearray representing the blue channel of the image
img_w	The width of the image
img_h	The height of the image

Returns: DE_NO_ERROR on success, an error code otherwise

LoadImageDataManualIndexHandling

```
int LoadImageDataManualIndexHandling(
    int*  dataset_index,
    char* image_in_data,
    int   image_in_data_size);
```

Loads the image data from a bytearray, as read directly from a PNG, JPG or TIF file. The function will give back an index that will later be used to perform operations on the image. The index will be negative, which indicates that it will not be counted towards the cache size and it will not be automatically deleted. It must be manually deleted using the "DeleteDatasetManualIndexHandling" function.

Arguments:

dataset_index	The dataset index will be written to the value pointed to by this pointer
image_in_data	The image data as a bytearray, as read from a PNG, JPG or TIF file
image_in_data_size	The size of image_in_data in bytes

Returns: DE_NO_ERROR on success, an error code otherwise

DeleteDatasetManualIndexHandling

```
int DeleteDatasetManualIndexHandling(
    int dataset_index);
```

This will delete a dataset created by "LoadImageDataManualIndexHandling".

Arguments:

dataset_index	The index of the dataset that is to be deleted

Returns: DE_NO_ERROR on success, an error code otherwise

SetCacheSize

```
int SetCacheSize(
    int new_size);
```

When a new dataset is created, the datasets belonging to older indices will be removed from memory. This happens if the number of retained datasets exceeds the defined cache size, which is 3 by default.

Arguments:

new_size	The new size of the cache; if the size is decreased, older datasets might be removed accordingly

Returns: DE_NO_ERROR on success, an error code otherwise

EnableImageProcessingCache

```
int EnableImageProcessingCache(
    bool enable_processing_cache);
```

Images that are created by internal processing (e.g. a rescaled version of an input image) are cached by default, so they do not need to be recalculated if they are needed again. Deactivating this functionality will reduce memory usage but potentially increase processing time.

Arguments:

enable_processing_cache	Boolean that determines whether the image caching should be activated (true, default) or deactivated (false)

Returns: DE_NO_ERROR on success, an error code otherwise

SetDefaultJson

```
int SetDefaultJson(
    char* json_data,
    int   json_size);
```

This function sets the default JSON data that will be used for all new data sets that are created via LoadImage. If this function is not called, hard coded default values will be used instead.

Arguments:

json_data	The json data as a serialized byte string
json_size	The size of json_data

Returns: DE_NO_ERROR on success, an error code otherwise

SetJson

```
int SetJson(
    int   index,
    char* json_data,
    int   json_size);
```

This function sets the JSON data specifically for the selected dataset.

Arguments:

index	The index of the dataset that will have its JSON changed
json_data	The json data as a serialized byte string
json_size	The size of json_data

Returns: DE_NO_ERROR on success, an error code otherwise

SetResults

```
int SetResults(
    int   index,
    char* input_chars,
    int   input_chars_size);
```

This function will set the results container of a given dataset. The input is a serialized protobuf messsage of the type ”Results” as defined in ”reults.proto”.

Arguments:

index	The index of the dataset that will have its results changed
input_chars	The serialized protobuf object
input_chars_size	The size of the serialized protobuf object

Returns: DE_NO_ERROR on success, an error code otherwise

4.4 Data Output

GetResults

```
int GetResults(
    int   index,
    char* output_chars,
    int*  output_chars_size);
```

Retrieve the results of the EvaluateImage function in the form of a serialized protobuf string of the ”Results” type, as defined in ”reults.proto”. The repeated ”output” field contains one ”ResultField” object for each model.

Arguments:

index	The index of the dataset that has been evaluated
output_chars	The serialized protobuf object will be written into this buffer
output_chars_size	This variable should contain the size of the buffer given with output_chars in bytes; it will contain the amount of written data in bytes after the function has returned

Returns: DE_NO_ERROR on success, an error code otherwise

DrawBoxes

```
int DrawBoxes(
    int    index,
    double overlap_threshold,
    double alpha_boxes,
    double alpha_segmentations,
    char*  output_data,
    int*   output_data_size);
```

The features that have been found during EvaluateImage will be drawn into an image in the form of boxes and/or the raw segmentation data. The image will be serialized and written into a buffer, where every byte corresponds to one channel in RGB format.

Arguments:

index	The index of the dataset that has been evaluated
overlap_threshold	This gives a value for how much boxes can overlap before the smaller box is omitted; 1.0 means that boxes are never omitted; 0.0 means that even the smallest overlap will lead to omission of the smaller box
alpha_boxes	This determines how visible the boxes will be; 1.0 means fully visible; 0.0 means invisible
alpha_segmentations	This determines how visible the segmentations will be; 1.0 means fully visible; 0.0 means invisible
output_data	The buffer that the resulting image is written to, as serialized RGB bytes
output_data_size	This variable should contain the size of output_data in bytes, to avoid buffer oerruns; after the function has returned, it will contain the length of the written data

Returns: DE_NO_ERROR on success, an error code otherwise

GetOriginalImageDimensions

```
int GetOriginalImageDimensions(
    int  index,
    int* w,
    int* h,
    int* c);
```

Get the width, height and number of channels of the original image saved under the given index.

Arguments:

index	The index of the dataset
w	The width of the image is written to the value pointed to by this variable
h	The height of the image is written to the value pointed to by this variable
c	The number of channels of the image is written to the value pointed to by this variable

Returns: DE_NO_ERROR on success, an error code otherwise

GetOriginalImage

```
int GetOriginalImage(
    int   index,
    char* image_out_data,
    int*  image_out_data_size);
```

Retrieve the original image in the form of serialized bytes, where every byte corresponds to the value of one channel in one pixel.

Arguments:

index	The index of the dataset
image_out_data	The buffer that the image is written to, as serialized bytes
image_out_data_size	This variable should contain the size of image_out_data in bytes, to avoid buffer overruns; after the function has returned, it will contain the length of the written data

Returns: DE_NO_ERROR on success, an error code otherwise

GetOriginalImage8UC3

```
int GetOriginalImage8UC3(
    int   index,
    char* image_out_data,
    int*  image_out_data_size);
```

Retrieve the original image in the form of serialized bytes, where every byte corresponds to the value of one channel of one pixel in the RGB format. This function enforces the output of the image as an 8-bit unsigned integer 3 channel image.

Arguments:

index	The index of the dataset
image_out_data	The buffer that the image is written to, as serialized RGB bytes
image_out_data_size	This variable should contain the size of image_out_data in bytes, to avoid buffer overruns; after the function has returned, it will contain the length of the written data

Returns: DE_NO_ERROR on success, an error code otherwise

GetUndrawnChangeStatus

```
int GetUndrawnChangeStatus(
    int   index,
    bool* undrawn_changes);
```

Some functions ("ProcessFeatures") can lead to a change in the image returned by "DrawBoxes". If changes occurred since the last call of "DrawBoxes", the returned bool will be true, otherwise false.

Arguments:

index	The index of the dataset
undrawn_changes	Indicates whether there were changes to the image since the last call of "DrawBoxes"

Returns: DE_NO_ERROR on success, an error code otherwise

GetCLAHEImageRaw

```
int GetCLAHEImageRaw(
    int    index,
    double clip_limit,
    int    grid_size,
    int    mode,
    char*  image_out_data,
    int*   image_out_data_size);
```

Perform a CLAHE contrast adjustment operation on the image in the target dataset with the given parameters. The output will be written to image_out_data.

Arguments:

index	Index of the target dataset
clip_limit	Clip limit of the CLAHE operation
grid_size	Grid size of the CLAHE operation
mode	The CLAHE mode; can use the header defined integer constants CLAHE_GRAY (0) for grayscale images or CLAGE_LAB (1) for RGB images
image_out_data	The output image is written into this bytearray
image_out_data_size	The size of the data written to image_out_data

Returns: DE_NO_ERROR on success, an error code otherwise

SwitchBR

```
int SwitchBR(
    char* data,
    int   size);
```

Takes an image in the form of a serialized RGBRGB... bytearray and switches the R and B channels.

Arguments:

data	The image data as a bytearray
size	The size of the given image data

Returns: DE_NO_ERROR on success, an error code otherwise

GetFeatureTable

```
int GetFeatureTable(
    int   index,
    char* json_data,
    int*  json_size);
```

Get the feature table as a serialized JSON byte string. This JSON will contain a list, where one entry corresponds to one list entry.

Arguments:

index	The index of the dataset
json_data	The json data as a serialized byte string
json_size	The size of json_data

Returns: DE_NO_ERROR on success, an error code otherwise

GetHeatmapDimensions

```
int GetHeatmapDimensions(
    int* w,
    int* h,
    int* c);
```

Get the dimensions of the heatmap. This can be used to allocate the size of the required bytearray for the "GetModelHeatmap" and "GetCombinedHeatmap" functions as their size needs to be width * height * channels.

Arguments:

w	The width of the heatmap is written to the integer pointed to by this variable
h	The height of the heatmap is written to the integer pointed to by this variable
c	The number of channels of the heatmap is written to the integer pointed to by this variable

Returns: DE_NO_ERROR on success, an error code otherwise

GetModelHeatmap

```
int GetModelHeatmap(
    char* model_id,
    int   model_id_size,
    char* image_data,
    int*  image_data_size);
```

Get the heatmap for a specific model. The output will be an RGBRGB... bytearray.

Arguments:

model_id	The model ID encoded as a bytearray
model_id_size	The size of model_id
image_data	The image data of the heatmap will be written here
image_data_size	The size of the written image_data

Returns: DE_NO_ERROR on success, an error code otherwise

GetCombinedHeatmap

```
int GetCombinedHeatmap(
    char* image_data,
    int*  image_data_size);
```

Get the combined heatmap for all models. The output will be an RGBRGB... bytearray.

Arguments:

image_data	The image data of the heatmap will be written here
image_data_size	The size of the written image_data

Returns: DE_NO_ERROR on success, an error code otherwise

ListSets

```
int ListSets();
```

This function lists the indices of all currently existing datasets to stdout.

Arguments:

–

Returns: DE_NO_ERROR on success, an error code otherwise

4.5 Methods on Data

EvaluateImage

```
int EvaluateImage(
    int index);
```

Evaluates the image under the given index using the models that were loaded during ReadAllModels. The results can be retrieved via GetResults.

Arguments:
index The index of the dataset that contains the image that is to be evaluated

Returns: DE_NO_ERROR on success, an error code otherwise

ProcessFeatures

```
int ProcessFeatures(
    int index);
```

This function will filter the defects using the parameters in the given JSON data. It will also determine the image class and prepare the feature table that can be read using GetFeatureTable.

Arguments:
index The index of the dataset that will have its features processed

Returns: DE_NO_ERROR on success, an error code otherwise

ReapplyThresholds

```
int ReapplyThresholds(
    int index);
```

When a new JSON is set for a dataset via "SetJson", it might contain different segmentation thresholds than the old one. If that is the case these new thresholds can be applied using this function.

Arguments:
index The index of the dataset that will have its thresholds reapplied

Returns: DE_NO_ERROR on success, an error code otherwise

UpdateHeatmaps

```
int UpdateHeatmaps(
    int index);
```

Calling this function will add the results of the evaluation of a dataset to the global feature heatmaps.

Arguments:

index The index of the dataset whose results will be added to the global feature heatmaps

Returns: DE_NO_ERROR on success, an error code otherwise

ResetHeatmap

```
int ResetHeatmap();
```

This will reset the internal heatmap.

Arguments:

–

Returns: DE_NO_ERROR on success, an error code otherwise

CalculateRLCs

```
int CalculateRLCs(
    int index);
```

The row_rlc field of the features in the results (see "results.proto") is not populated by default. This method encodes the segmentation map of a feature into the row_rlc field. Every element in the "row_rlc" array denotes a position on the image (in pixels) and the length of a line (in pixels) that is to be drawn from that position towards the right (positive x direction).

Arguments:

index The index of the dataset whose features will be encoded into the row_rlc format

Returns: DE_NO_ERROR on success, an error code otherwise

4.6 Other Methods

ReadAllModels

```
int ReadAllModels(
    const char* path,
    char*       proto_chars,
    int*        proto_size,
    int         device);
```

Read and load all model files in the specified directory. The function will return information on the models via a serialized protobuf object of the "Results" type, where every model has its information put into a "ResultField" object. Currently this functions treats all files in the specified directory with the extension ".denk" as model files. This function also sets the device that should be used for the later evaluations. This function needs to be called once before the first call to EvaluateImage. The evaluation via GPU is only available, if the library has been built using either the "CUDA" or the "DML" compiler flag.

Arguments:

path	The path to the directory that contains the model files
proto_chars	The serialized protobuf container with the model information will be written into this buffer
proto_size	This variable should contain the length of the proto_chars buffer in bytes, to avoid buffer overruns; it will contain the length of the data that has been written to proto_chars after the function has returned
device	The device for the image evaluation can be selected using this variable; -1 signifies the CPU, 0 the first GPU, 1 the second GPU and so on

Returns: DE_NO_ERROR on success, an error code otherwise

EndSession

```
int EndSession();
```

Properly unallocate the used objects from memory. Not calling this function before exiting the calling program can lead to a crash. This is especially likely if the GPU evaluation has been used, but calling this function before exiting is recommended in all cases where models have been loaded.

Arguments:

–

Returns: DE_NO_ERROR on success, an error code otherwise

PrintModelNames

```
int PrintModelNames();
```

Print the names of all models that are currently loaded to stdout.

Arguments:
–

Returns: DE_NO_ERROR on success, an error code otherwise

GetJson

```
int GetJson(
    int   index,
    char* json_data,
    int*  json_size);
```

Get the current internal JSON of a dataset.

Arguments:

index	The index of the dataset
json_data	The serialized data of the JSON will be written to this bytearray
json_size	The size of the data written to json_data

Returns: DE_NO_ERROR on success, an error code otherwise

CreateJsonEntries

```
int CreateJsonEntries(
    char* json_in_data,
    int   json_in_size,
    char* json_out_data,
    int*  json_out_size);
```

After models have been loaded via "ReadModels", this function can be used to create any missing feature fields in a given JSON.

Arguments:

json_in_data	The input JSON as a bytearray
json_in_size	The size of the input JSON given with json_in_data
json_out_data	The output JSON will be written to this variable
json_out_size	The size of the data written to json_out_size

Returns: DE_NO_ERROR on success, an error code otherwise

SetAreaMapFromFile

```
int SetAreaMapFromFile(
    char* area_map_data,
    int   area_map_size);
```

This function takes an unsigned 16-bit int TIFF encoded file as an input and sets it as the internal area map.

Arguments:

area_map_data	The unsigned 16-bit int TIFF encoded data of the area map
area_map_size	The size of the data given by area_map_data

Returns: DE_NO_ERROR on success, an error code otherwise

SetAreaMapFromParameters

```
int SetAreaMapFromParameters(
    int image_width,
    int image_height,
    int cells_horizontal,
    int cells_vertical,
    int border_left,
    int border_right,
    int border_top,
    int border_bottom);
```

This function creates the internal area map as a grid defined by the given parameters.

Arguments:

image_width	The width of the reference image
image_height	The height of the reference image
cells_horizontal	The number of cells in the horizontal direction
cells_vertical	The number of cells in the vertical direction
border_left	The border to the left side of the image in pixels
border_right	The border to the right side of the image in pixels
border_top	The border to the top side of the image in pixels
border_bottom	The border to the bottom side of the image in pixels

Returns: DE_NO_ERROR on success, an error code otherwise

SetAreaMapFromColoredImage

```
int SetAreaMapFromColoredImage(
    char* area_map_data,
    int   area_map_size);
```

This function takes an encoded image (PNG, JPG, TIF, ...) and tries to create the internal area map from this file. Every connected area with exactly the same color will be interpreted as an area.

Arguments:

area_map_data	The data of the input image
area_map_size	The size of the data given by area_map_data

Returns: DE_NO_ERROR on success, an error code otherwise

SetAreaMapFromNeuralNetwork

```
int SetAreaMapFromNeuralNetwork(
    int          index,
    const char* model_uid,
    double       threshold);
```

This function takes the image of a given dataset and runs the model given by model_uid on the image. The result will be interpreted as an area map.

Arguments:

index	The dataset index of the image
model_uid	The model UID of the model that is to be used
threshold	The segmentation threshold that is to be applied after the evaluation

Returns: DE_NO_ERROR on success, an error code otherwise

CheckForAreaMap

```
int CheckForAreaMap();
```

This function will return the typical ”DE_NO_ERROR” code if an area map has been defined. If that is not the case, it will return ”DE_AREA_MAP_NOT_SET”.

Arguments:

–

Returns: DE_NO_ERROR if an area map has been defined, DE_AREA_MAP_NOT_SET otherwise

WriteAreaMap

```
int WriteAreaMap(
    const char* filename);
```

Writes the current area map to a file. The file format, determined by the given file extension, must be TIF

Arguments:

filename	The name of the file to be written; the extension must be .tif

Returns: DE_NO_ERROR on success, an error code otherwise

WriteColoredAreaMap

```
int WriteColoredAreaMap(
    const char* filename,
    int         colors);
```

Writes a colored version of the current area map to a file. The format can be PNG, JPG or TIF.

Arguments:

filename	The name of the file to be written
colors	The number of colors to be used

Returns: DE_NO_ERROR on success, an error code otherwise

GetAreaMapSize

```
int GetAreaMapSize(
    int* img_w,
    int* img_h);
```

Get the size of the currently defined area map.

Arguments:

img_w	The width of the current area map will be written here
img_h	The height of the current area map will be written here

Returns: DE_NO_ERROR on success, an error code otherwise

GetAreaMap

```
int GetAreaMap(
    char* area_map_data,
    int*  area_map_size);
```

Get the area map as a 16-bit unsigned integer bytearray. The dimensions can be taken from "GetAreaMapSize" and the image will always have 1 channel. Take note that the allocated length of area_map_data needs to be width * height * 2 and that 2 elements in the char array correspond to 1 value in the final image.

Arguments:

area_map_data	The data of the area map will be written here
area_map_size	The size of the data written to area_map_size

Returns: DE_NO_ERROR on success, an error code otherwise

GetColoredAreaMap

```
int GetColoredAreaMap(
    int   colors,
    char* area_map_data,
    int*  area_map_size);
```

Get the area map as a 3 channel 8-bit unsigned integer bytearray. The dimensions can be taken from "GetAreaMapSize" and the image will always have 3 channels.

Arguments:

colors	The number of colors to be used
area_map_data	The data of the area map will be written here
area_map_size	The size of the data written to area_map_size

Returns: DE_NO_ERROR on success, an error code otherwise

GetAreaAtPosition

```
int GetAreaAtPosition(
    int  x,
    int  y,
    int* area);
```

Get the area number at a specific position in the area map.

Arguments:

x	The x position
y	The y position
area	The area number will be written here

Returns: DE_NO_ERROR on success, an error code otherwise

NumberToLetterCode

```
int NumberToLetterCode(
    int   number,
    char* char_code,
    int*  char_code_len);
```

Convert a number to the corresponding letter code (A, B, C ..., AA, AB, AC, ...). This is only defined for numbers >= 1.

Arguments:

number	The input number
char_code	The buffer that the char code will be written to; decodable as "utf-8" or "ascii"
char_code_len	The size of the data written to char_code

Returns: DE_NO_ERROR on success, an error code otherwise

LetterCodeToNumber

```
int LetterCodeToNumber(
    char* char_code,
    int   char_code_len,
    int*  number);
```

Convert a letter code (A, B, C ..., AA, AB, AC, ...) to the corresponding number.

Arguments:

char_code	The inuput letter code encoded as "utf-8" or "ascii"
char_code_len	The size of char_code
number	The output number will be written here

Returns: DE_NO_ERROR on success, an error code otherwise

ProcessAreaMapping

```
int ProcessAreaMapping(
    int index);
```

By default the section of a feature is given with -1. To map the features to the corresponding areas, this function needs to be called.

Arguments:

index	The index of the datasets that should have its features associated to the defined areas

Returns: DE_NO_ERROR on success, an error code otherwise

GetAreaMapping

```
int GetAreaMapping(
    int    index,
    double alpha,
    int    colors,
    char*  output_data,
    int*   output_data_size);
```

This method will take the image within a dataset and draw the areas over it.

Arguments:

index	The index of the dataset from which the main image should be taken
alpha	A value between 0 (invisible) and 1 (fully visibile) that defines the visibility of the drawn areas.
colors	The number of colors to use; to avoid confusion it should never be set to a value that is a divisor of the horizontal number of areas
output_data	The image will be written to this variable in the form of an RGBRGB... bytearray
output_data_size	The size of the written output_data

Returns: DE_NO_ERROR on success, an error code otherwise

SetAreaMapCorner

```
int SetAreaMapCorner(
    int corner);
```

Defines the corner that the areas should be labeled from.

Arguments:

corner	The selected corner; 0 – top left; 1 – top right; 2 – bottom left; 3 – bottom right

Returns: DE_NO_ERROR on success, an error code otherwise

AreaToCoordinate

```
int AreaToCoordinate(
    int  area,
    int* x,
    int* y);
```

Get the position on the cell grid by area number.

Arguments:

area	The area number
x	The x position is written here
y	The y position is written here

Returns: DE_NO_ERROR on success, an error code otherwise

MakeCoordinateMapFromGrid

```
int MakeCoordinateMapFromGrid(
    int areas_horizontal,
    int areas_vertical,
    int corner);
```

Create a coordinate map by assuming a given grid.

Arguments:

areas_horizontal	The number of areas in the horizontal direction
areas_vertical	The number of areas in the vertical direction
corner	The selected corner (see "SetAreaMapCorner")

Returns: DE_NO_ERROR on success, an error code otherwise

MakeCoordinateMapFromEstimatedGrid

```
int MakeCoordinateMapFromEstimatedGrid(
    int corner);
```

Upon loading an area map the program tries to automatically estimate the cell number in the horizontal and vertical direction. This function works just as "MakeCoordinateMapFromGrid", but it uses the estimated grid layout instead.

Arguments:
corner The selected corner (see "SetAreaMapCorner")

Returns: DE_NO_ERROR on success, an error code otherwise

CheckIfSetExists

```
int CheckIfSetExists(
    int index);
```

Check if the dataset with the given index exists.

Arguments:
index The index of the dataset

Returns: DE_NO_ERROR if the dataset exists, DE_NO_DATASET_WITH_GIVEN_ID if it does not exist, DE_DATASET_EXPIRED if it did exist previously but was since deleted

SelectNetworkSet

```
int SelectNetworkSet(
    char* set_name,
    int   set_name_size);
```

Select one of the networksets defined in the JSON. The default ist "DEFAULT"

Arguments:
set_name The name of the set as a bytearray, encoded as "ascii"
set_name_size The size of set_name

Returns: DE_NO_ERROR on success, an error code otherwise

4.7 Deprecated Functions

ProcessCellMapping

```
int ProcessCellMapping(
    int index);
```

Link the found features to the previously defined areas. Please use "ProcessAreaMapping" instead.

Arguments:

index	Index of the target dataset

Returns: DE_NO_ERROR on success, an error code otherwise

GetCellMapping

```
int GetCellMapping(
    int    index,
    double alpha,
    char*  output_data,
    int*   output_data_size);
```

Draw the current cell mapping into an image given by the index. Please use "GetAreaMapping" instead.

Arguments:

index	Index of the target dataset
alpha	Visibility of the drawn areas
output_data	The output data will be written here
output_data_size	Size of the data written to output_data

Returns: DE_NO_ERROR on success, an error code otherwise

5 Common Issues

- **The API fails to detect the Dongle**: Try reinserting the Dongle or try using another USB port.

6 Glossary

Segmentation
Some neural networks deliver information about the location of a feature within an image. The segmentation refers to the area of the image that has been found to include a certain feature.

Deep Feature / Deep Defect
Some features might have a “deep” version, that is defined as any feature of that type that has a mean gray value below a certain threshold. This allows for the definition of separate binning thresholds for particularly severe defects.