8 Embedded inference client API

The embedded inference client API is part of the *<project_name>/Middlewares/ST/AI/src/<name>.h* file. All functions and macros are generated according to the provided C-network name.

8.1 Input and output x-D tensor layout

Input and output buffers are defined as a tensor with a maximum of 3 dimensions (HWC layout format, standing for Height, Width, and Channels). A batch dimension can be added to handle an array of tensors. The buffers are completely defined through the `struct ai_buffer` C structure definition.

```
/* @file: ai_platform.h */

typedef struct ai_buffer_ {
    ai_buffer_format    format;     /*!< buffer format */
    ai_u16        n_batches;  /*!< number of batches in the buffer */
    ai_u16        height;     /*!< buffer height dimension */
    ai_u16        width;      /*!< buffer width dimension */
    ai_u32        channels;   /*!< buffer number of channels */
    ai_handle          data;       /*!< pointer to buffer data */
} ai_buffer;
```

Note: Currently, only the `AI_BUFFER_FORMAT_FLOAT` format (32-bit floating-point) is supported for the input and output tensors.

This C structure can be also used to handle an opaque and specific data buffer.

8.1.1 1-D tensor

For a 1-D tensor, standard C-array type is expected to handle the input and output tensors.

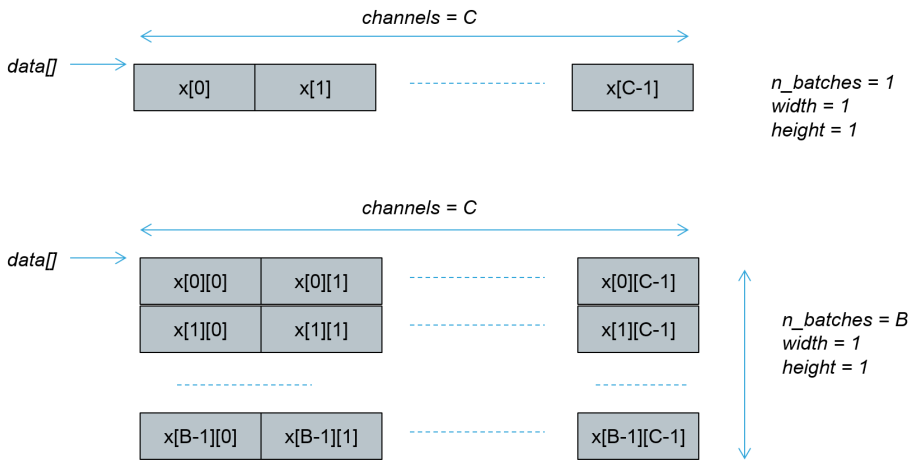
```
#include "network.h"

#define xx_SIZE VAL                 /* = H * W * C = C */

ai_float xx_data[xx_SIZE];          /* n_batch = 1, height = 1, width = 1, channels = C */

ai_float xx_data[B * xx_SIZE];      /* n_batch = B, height = 1, width = 1, channels = C */
ai_float xx_data[B][xx_SIZE];
```

Figure 40. 1-D Tensor data layout



8.1.2 2-D tensor

For a 2-D tensor, standard C-array-of-array memory arrangement is used to handle the input and output tensors. Two dimensions are mapped to each of two first dimensions of the tensor in the original toolbox representation: H and C in Keras / TensorFlow™, H and W in Lasagne.

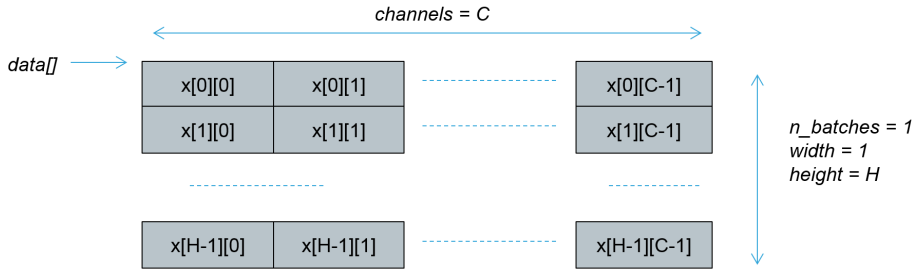
```
#include "network.h"

#define xx_SIZE VAL              /* = H * W * C = H * C */

ai_float xx_data[xx_SIZE];       /* n_batch = 1, height = H, width = 1, channels = C */
ai_float xx_data[H][C];

ai_float xx_data[B * xx_SIZE];   /* n_batch = B, height = H, width = 1, channels = C */
ai_float xx_data[B][H][C];
```

Figure 41. 2-D Tensor data layout



Note: If the dimension order in the original toolbox is different from HWC (such as Lasagne: CHW), it is the user's responsibility to properly re-arrange the elements.

8.1.3 3-D tensor

For a 3D-tensor, standard C-array-of-array-of-array memory arrangement is used to handle the input and output tensors.

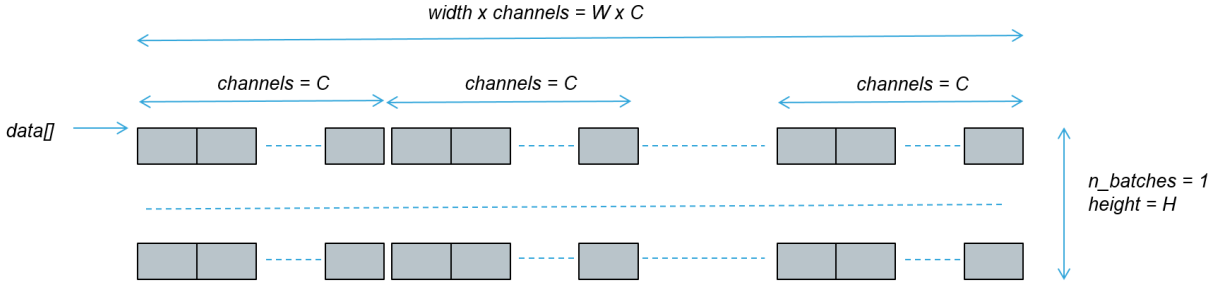
```
#include "network.h"

#define xx_SIZE VAL              /* = H * W * C */

ai_float xx_data[xx_SIZE];       /* n_batch = 1, height = H, width = W, channels = C */
ai_float xx_data[H][W][C];

ai_float xx_data[B * xx_SIZE];   /* n_batch = B, height = H, width = W, channels = C */
ai_float xx_data[B][H][W][C];
```

Figure 42. 3-D Tensor data layout



8.2 create() / destroy()

```
ai_error ai_<name>_create(ai_handle* network, const ai_buffer* network_config);
ai_handle ai_<name>_destroy(ai_handle network);
```

This mandatory function is the early function that must be called by the AI client to instantiate and initialize an NN instance. `ai_handle` references an opaque context that must be passed to the other functions.

- The `<name>_config` parameter is a specific network configuration buffer coded as an `ai_buffer`. It is generated (`AI_<NAME>_DATA_CONFIG` C-define) by the AI code generator (see the *<name>.h* file)
- When the instance is no more used by the application, the `ai_<name>_destroy()` function must be called to release the allocated resources

Typical usage

```
#include <stdio.h>
#include "network.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;
...
int aiInit(void) {
    ai_error err;
    ...
    err = ai_network_create(&network, AI_NETWORK_DATA_CONFIG);
    if (err.type != AI_ERROR_NONE) {
        /* manage the error */
        printf("E: AI error - type=%d code=%d\r\n", err.type, err.code);
        ...
    }
    ...
}
```

8.3 get_error()

```
ai_error ai_<name>_get_error(ai_handle network);
```

This function can be used by the client to retrieve the first error reported during the execution of an `ai_<name>_xxx()` function. Refer to the *ai_platform.h* file to have the list of the returned error types (`ai_error_type`) and associated codes (`ai_error_code`) .

Typical AI error function handler (for debug and log purpose)

```
#include "network.h"
...
void aiLogErr(const ai_error err, const char *fct)
{
    if (fct)
        printf("E: AI error (%s) - type=%d code=%d\r\n", fct, err.type, err.code);
    else
        printf("E: AI error - type=%d code=%d\r\n", err.type, err.code);
}
```

8.4 get_info()

```
ai_bool ai_<name>_get_info(ai_handle network, ai_network_report* report);
```

This optional function can be used by the client to retrieve the informations of the instantiated NN (for debug and log purpose).

The `network` handle must be a valid handle. Refer to the `ai_<name>_create()` function.

Typical usage

```
#include "network.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;
...
int aiInit(void) {
    ai_network_report report;
    ai_bool res;
...
    res = ai_network_get_info(network, &report)
    if (res) {
        /* display/use the reported data */
        ...
    }
...
}
```

8.5 init()

```
ai_bool ai_<name>_init(ai_handle network, const ai_network_params* params);
```

This mandatory function must be used by the client to initialize the internal run-time structure of the instantiated NN:

- The `params` parameter is a structure (`ai_network_params` type) that passes the references of the generated weights (`params` attribute), and the activation/crash memory buffer (`activations` attribute).
- The `network` handle must be a valid handle. Refer to the `ai_<name>_create()` function.

```
/* @file: ai_platform.h */
typedef struct ai_network_params_ {
  ai_buffer params;           /*! info about params buffer(required!) */
  ai_buffer activations;      /*! info about activations buffer (required!) */
} ai_network_params;
```

Multiple C-macro helpers and specific `AI_<NAME>_XX` C defines must be used to initialize this parameter:

- The `params` attribute handles the weights/bias memory buffer
- The `activations` attribute handles the activation/crash memory buffer, which is used by the forward process (refer to the `ai_<name>_run()` function)
- The sizes of associated memory blocks are respectively defined by the following C defines (refer to file *<name>_data.h*). The memory layouts of these buffers depend on the Neural Network implemented.
  - `AI_<NAME>_DATA_WEIGHTS_SIZE`
  - `AI_<NAME>_DATA_ACTIVATIONS_SIZE`

Typical usage

```
#include <stdio.h>
#include "network.h"
#include "network_data.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;

/* Global buffer to handle the activations data buffer - R/W data */
AI_ALIGNED(4)
static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];
...

int aiInit(void) {
    ai_error err;
```

```
...
    /* initialize network */
    const ai_network_params params = {
        AI_NETWORK_DATA_WEIGHTS(ai_network_data_weights_get()),
        AI_NETWORK_DATA_ACTIVATIONS(activations) };
    if (!ai_network_init(network, &params)) {
        err = ai_network_get_error(network);
        /* manage the error */
        ...
        }
...
}
```

8.6 run()

```
ai_i32 ai_<name>_run(ai_handle network, const ai_buffer* input, ai_buffer* output);
```

This function is the main function to feed the NN. The input and output buffer parameters (`ai_buffer` type) provide the input tensors and store the predicted output tensors respectively (refer to Section 8.1 Input and output x-D tensor layout):

- The returned value is the number of the input tensors processed (when `n_batches` > 1). If the returned value is negative or null, use the `ai_<name>_get_error()` function to know the error.
- `AI_<NAME>_IN_1` (respectively `AI_<NAME>_OUT_1`) must be used to initialize the input (respectively output) buffer handle
- `AI_<NAME>_IN_1_SIZE` (respectively `AI_<NAME>_OUT_1_SIZE`) is used to initialize the input data (respectively output data) buffers

Note: Two separate lists of input and output `ai_buffer` can be passed. This allows the future support of neural network with multiple inputs, outputs or both. `AI_<NAME>_IN_NUM` and `AI_<NAME>_OUT_NUM` respectively are used to know at compile-time the number of inputs and outputs. These values are also returned by the `ai_network_report` structure (refer to the `ai_<name>_get_info()` function).

Typical usage

```
#include <stdio.h>
#include "network.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;
...
static ai_buffer ai_input[AI_NETWORK_IN_NUM] = { AI_NETWORK_IN_1 };
static ai_buffer ai_output[AI_NETWORK_OUT_NUM] = { AI_NETWORK_OUT_1 };
...
int aiRun(const ai_float *in_data, ai_float *out_data,
          const ai_u16 batch_size)
{
    ai_i32 nbatch;
    ...
    /* initialize input/output buffer handlers */
    ai_input[0].n_batches = batch_size;
    ai_input[0].data = AI_HANDLE_PTR(in_data);
    ai_output[0].n_batches = batch_size;
    ai_output[0].data = AI_HANDLE_PTR(out_data);

    nbatch = ai_network_run(network, &ai_input[0], &ai_output[0]);
    if (nbatch != batch_size) {
        err = ai_network_get_error(network);
        /* manage the error */
        ...
        }
    ...
}
```